

Wykorzystaj innowacyjne metody nauki
i zacznij tworzyć dynamiczne aplikacje internetowe!

Wydanie II
zaktualizowane
zgodnie z najnowszą
wersją egzaminu c:out
z platformy JSP 1.5

Head First Servlets & JSP

Przetestuj swoją wiedzę,
odpowiadając na ponad
200 pytań egzaminu próbnego

Edycja
polska



Watch it!

Unikaj śmiertelnych
pułapek i podchwytliwych
pytań na egzaminie
w wersji 1.5



Przekazuj swoje
komunikaty światu
za pomocą
akcji c:out



Dowiedz się, jak Ted pracował
nad swoim wdziękiem, korzystając
z dynamicznych atrybutów



Poderwij dziewczynę na
bibliotekę znaczników
niestandardowych

O'REILLY®

Bryan Basham, Kathy Sierra, Bert Bates

Helion



Tytuł oryginału: Head First Servlets and JSP™. Second edition

Tłumaczenie: Mikołaj Szczepaniak

Na podstawie „Head First Servlets & JSP. Edycja polska”

w tłumaczeniu Piotra Rajcy i Mikołaja Szczepaniaka.

ISBN: 978-83-246-6057-5

© Helion S.A. 2009.

Authorized translation of the English edition of Head First Servlets and JSP, 2E

© 2008 O'Reilly Media, Inc.

This translation is published and sold by permission of O'Reilly Media, Inc.,
the owner of all rights to publish and sell the same.

All rights reserved. No part of this book may be reproduced or transmitted in any
form or by any means, electronic or mechanical, including photocopying, recording
or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości
lub fragmentu niniejszej publikacji w jakiegokolwiek postaci jest zabronione.
Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie
książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie
praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi
bądź towarowymi ich właścicieli.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte
w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej
odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne
naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo HELION
nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe
z wykorzystania informacji zawartych w książce.

Wydawnictwo HELION

ul. Kościuszki 1c, 44-100 GLIWICE

tel. 032 231 22 19, 032 230 98 63

e-mail: helion@helion.pl

WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

http://helion.pl/user/opinie?hfsjp2_ebook

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:

<ftp://ftp.helion.pl/przyklady/hfsjp2.zip>

Printed in Poland.

- [Poleć książkę na Facebook.com](#)
- [Kup w wersji papierowej](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Niniejszą książkę dedykujemy osobie — kimkolwiek jest —
która zdecydowała, że domyślny obiekt języka EL dla parametru
kontekstu powinien nosić nazwę *initParam*...

Współwinni powstania serii Head First (i tej książki)

Bert Bates



Kathy Sierra



Bryan Basham



Bert od wielu lat zajmuje się tworzeniem i projektowaniem programów, jednak wieloletnia praca nad zagadnieniami sztucznej inteligencji sprawiła, że zainteresował się teorią nauczania i nauczaniem z wykorzystaniem nowych technologii. Pierwszą dekadę swojej programistycznej kariery Bert spędził, podróżując po świecie i pomagając rozgłośniom radiowym takim jak Radio New Zealand, Weather Channel czy też Art & Entertainment Network (A & E). Aktualnie pracuje w firmie Sun, gdzie jest członkiem zespołu opracowującego kilka egzaminów serii Java Certification, w tym także nowy egzamin SCWCD.

Bert jest zapalonym graczem w Go i już od bardzo dawna pracuje na programem do gry w tę grę. Może Java okaże się językiem na tyle ekspresyjnym, że pozwoli mu skończyć ten projekt. Bert dość dobrze gra na gitarze, a obecnie próbuje swoich sił, muzykując na banjo. Jego ostatnią przygodą był zakup konia islandzkiego, co niewątpliwie będzie stanowić nowe wyzwanie dla jego talentów pedagogicznych i trenerskich...

Kathy interesowała się teorią nauczania od czasów, gdy pracowała jako projektantka gier (pisała gry dla takich firm jak Virgin, MGM oraz Amblin) i programistka rozwiązań z zakresu sztucznej inteligencji. Znaczną część pracy nad przygotowaniem tej książki Kathy wykonała podczas prowadzenia kursu New Media Interactivity w ramach programu dodatkowego studiów nad rozrywką na Uniwersytecie Kalifornijskim. Ostatnio Kathy pracowała w firmie Sun Microsystems jako główny instruktor, ucząc mniej doświadczonych instruktorów języka Java, jak należy wyklądać najnowsze technologie tego języka, oraz uczestniczyła w opracowywaniu wielu egzaminów certyfikacyjnych firmy Sun, w tym także egzaminu SCWCD. Wraz z Bertem Batesem aktywnie wykorzystywała pomysły zawarte w niniejszej książce podczas nauczania tysięcy programistów. Jest także założycielką jednej z największych witryn przeznaczonych dla społeczności programistów używających Javy — javaranch.com — która w latach 2003 oraz 2004 zdobyła nagrodę za wydajność przyznawaną przez magazyn Software Development. Kathy lubi bieganie, jazdę na nartach, konie, jazdę na deskorolce oraz różne dziwne dziedziny nauki.

Bryan ma ponad dwudziestoletnie doświadczenie w zakresie programowania, w tym także pracę w NASA, gdzie zajmował się zaawansowanym programowaniem automatów z wykorzystaniem technik sztucznej inteligencji. Pracował także dla firmy konsultingowej zajmującej się tworzeniem obiektowych aplikacji biznesowych. Aktualnie Bryan pracuje w firmie Sun, w zespole zajmującym się opracowywaniem egzaminów, gdzie koncentruje się na zagadnieniach związanych z Javą oraz zasadami projektowania obiektowego. Uczestniczył w tworzeniu wielu egzaminów firmy Sun, w tym tych poświęconych technologiom JDBC, J2EE, serwletom i JSP oraz ogólnie kwestii tworzenia oprogramowania obiektowego. Był także głównym projektantem zarówno oryginalnej, jak i najnowszej wersji egzaminu SCWCD.

Bryan jest praktykującym buddystą zen, doskonałym graczem w latające talerze i audiofilem. Uprawia również jazdę na nartach telemarkiem.

Napisz do nas:
terrapi@wickedlysmart.com
kathy@wickedlysmart.com
bryan@wickedlysmart.com

Spis treści (skrótowy)

Wprowadzenie	15
1. Do czego służą serwlety i strony JSP? <i>Wprowadzenie i przegląd najważniejszych zagadnień</i>	29
2. Architektura aplikacji internetowej. <i>Bardziej szczegółowy przegląd zagadnień</i>	65
3. Minipodręcznik MVC. <i>Omówienie MVC</i>	95
4. Być serwletem. <i>Żądanie i odpowiedź</i>	121
5. Być aplikacją internetową. <i>Atrybuty i obiekty nasłuchujące</i>	175
6. Stan konwersacyjny. <i>Zarządzanie sesjami</i>	251
7. Być stroną JSP. <i>Stosowanie technologii JSP</i>	309
8. Strony bezkryptowe. <i>Bezkryptowe strony JSP</i>	371
9. Potęga znaczników niestandardowych. <i>Stosowanie biblioteki JSTL</i>	467
10. Kiedy JSTL nie wystarcza. <i>Tworzenie znaczników niestandardowych</i>	527
11. Jak wdrożyć aplikację internetową? <i>Wdrażanie aplikacji internetowych</i>	629
12. Zachowaj to w tajemnicy, ukryj w bezpiecznym miejscu. <i>Bezpieczeństwo aplikacji internetowych</i>	677
13. Potęga filtrów. <i>Filtry i opakowania</i>	729
14. Korporacyjne wzorce projektowe. <i>Wzorce i Struts</i>	765
Dodatek A Końcowy egzamin próbny	819
Skorowidz	893

Spis treści



Wprowadzenie

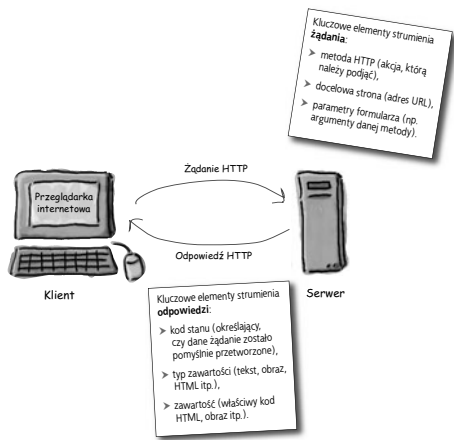
Twój mózg koncentruje się na serwletach. W tym rozdziale *Ty* próbujesz się czegoś *nauczyć*, a Twój *mózg* robi Ci przysługę i nie przykłada się do *zapamiętywania* zdobytej wiedzy. Twój mózg myśli sobie: „Lepiej zachowam miejsce na bardziej istotne informacje, na przykład: jakich dzikich zwierząt należy unikać bądź czy jazda nago na snowboardzie jest dobrym pomysłem”. Jak w takim razie można przekonać swój mózg, że nasze życie zależy od opanowania serwletów?

Dla kogo jest ta książka?	16
Wiemy, co sobie myśli Twój mózg	17
Metapoznanie	19
Zmuś swój mózg do posłuszeństwa	21
Czego potrzebujesz, aby skorzystać z tej książki?	22
Zdajemy egzamin certyfikujący	24
Redaktorzy techniczni	26
Podziękowania	27

1

Do czego służą serwlety i strony JSP?

Aplikacje internetowe są super. Ile tradycyjnych aplikacji z graficznym interfejsem użytkownika używanych przez miliony osób na całym świecie potrafisz wymienić? Jako programista aplikacji internetowych możesz uwolnić się od problemów wdrażania będących udziałem wszystkich standardowych aplikacji i udostępniać swoje aplikacje każdemu, kto dysponuje przeglądarką internetową. Będziesz jednak potrzebował serwletów i stron JSP. Będziesz ich potrzebował, ponieważ zwykłe, statyczne strony HTML były dobre... w latach 90. ubiegłego wieku. Zatem dowiedz się, jak przekształcić *witrynę WWW* w *aplikację internetową*.

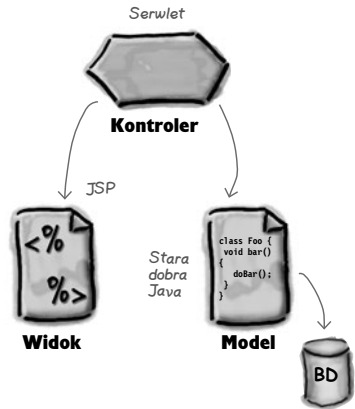


Cele egzaminu	30
Czym zajmuje się serwer WWW i klient oraz jak się ze sobą porozumiewają?	32
Dwuminutowy kurs języka HTML	35
Czym jest protokół HTTP?	38
Anatomia żądań GET i POST oraz odpowiedzi protokołu HTTP	44
Lokalizacja stron WWW przy użyciu adresów URL	48
Serwery WWW, strony statyczne i CGI	52
Serwlety bez tajemnic: pisanie, wdrażanie i uruchamianie serwletów	58
Technologia JSP jest efektem wprowadzenia języka Java do kodu HTML	62

2

Architektura aplikacji internetowej

Serwlety potrzebują pomocy. Kiedy do naszej aplikacji dociera żądanie, ktoś musi utworzyć obiekt serwletu lub przynajmniej wątek, który to żądanie obsłuży. Ktoś musi wywołać metodę doPost () lub doGet () serwletu. Ktoś musi przekazać żądanie do serwletu oraz odebrać to, co serwlet wygeneruje w odpowiedzi. Ktoś musi decydować o życiu, śmierci i zasobach niezbędnych do pracy serwletu. W tym rozdziale przyjrzymy się koncepcji kontenera i po raz pierwszy zwrócimy uwagę na wzorzec projektowy MVC.

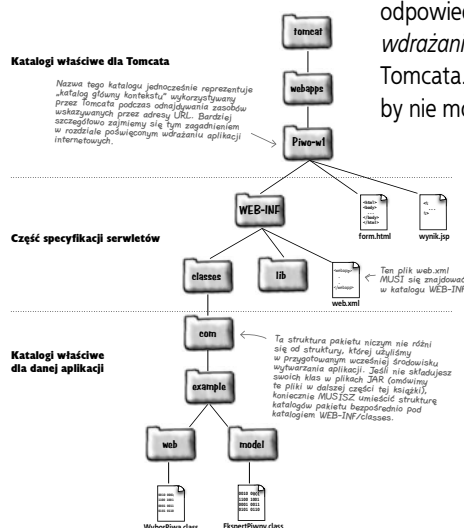


Cele egzaminu	66
Czym jest kontener oraz co nam daje?	67
Jak to wszystko wygląda w kodzie (co sprawia, że serwlet jest serwletem)?	72
Określanie nazw serwletów i kojarzenie ich z adresami URL w deskrytorze wdrożenia	74
Opowiadanie: Bob buduje witrynę swatającą (wprowadzenie do wzorca MVC)	78
Ogólne informacje i przykład wzorca model-widok-kontroler (MVC)	82
„Działający” deskrytor wdrożenia (DD)	92
Jaka w tym wszystkim jest rola platformy J2EE?	93

3

Minipodręcznik MVC

Tworzenie i wdrażanie aplikacji internetowych MVC. Nadszedł czas, aby utrudzić nasze dłonie pisaniem formularzy HTML, kontrolerów serwetów, modeli (zwykłych, tradycyjnych klas Javy), deskryptorów wdrożenia w formacie XML oraz widoków opartych na stronach JSP. Najwyższa pora zbudować, wdrożyć i przetestować taką aplikację. Najpierw jednak musimy przygotować odpowiednie środowisko *wytwarzania* aplikacji. Następnie musimy przygotować środowisko *wdrażania*, postępując przy tym zgodnie ze specyfikacją serwetów i JSP oraz wymaganiami Tomcata. Owszem... tworzymy małą aplikację, jednak niemal żadna aplikacja nie jest na tyle mała, by nie można w niej było wykorzystać wzorca MVC.

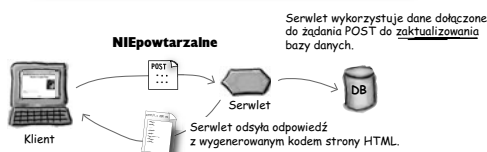


Cele egzaminu	96
Zbudujmy aplikację internetową MVC; pierwszy projekt	97
Tworzenie środowisk wytwarzania i wdrażania aplikacji	100
Tworzenie i testowanie kodu HTML początkowej strony formularza	103
Tworzenie deskryptora wdrożenia (DD)	105
Tworzenie, kompilacja, wdrażanie i testowanie serwetu kontrolera	108
Projektowanie, tworzenie i testowanie komponentu modelu	110
Rozszerzenie kontrolera o wywołania modelu	111
Tworzenie i wdrażanie komponentów widoku (to właśnie JSP)	115
Rozszerzenie serwetu o wywołanie strony JSP	116

4

Być serwetem

Serwety potrzebują pomocy. Zadaniem serwetu jest obsługa *żądań* klientów i odsyłanie do klienta właściwych *odpowiedzi*. Żądanie może być zupełnie proste, np. *prześlij mi stronę powitalną*, lub znacznie bardziej skomplikowane, np. *wygeneruj zamówienie na podstawie zawartości mojego koszyka*. **Żądanie** obejmuje kluczowe dane, a kod Twojego serwetu musi wiedzieć, jak należy te dane *odszukać* i jak ich *użyć*. Co więcej, kod serwetu musi wiedzieć, jak odesłać **odpowieź**. A jeśli nie...

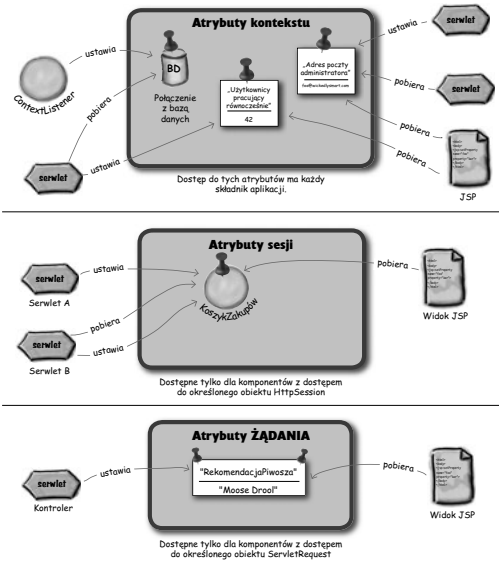


Cele egzaminu	122
Życie serwetu w kontenerze	123
Inicjalizacja i wątki serwetu	129
FAKTYCZNYM celem serwetu jest obsługa żądań GET i POST	133
Historia pewnego niepowtarzalnego żądania	140
Co sprawia, że przeglądarka wysyła albo żądanie GET, albo żądanie POST?	145
Wysyłanie i stosowanie parametrów	147
Dobrze, wiemy już, do czego służy klasa Request...	154
przyjrzymy się teraz klasie Response	161
Możesz ustawiać nagłówki odpowiedzi, możesz dodawać nagłówki odpowiedzi	164
Przekierowania kontra przydział żądań	168
Przegląd klasy HttpServletResponse	168

5

Być aplikacją internetową

Żaden serwlet nie działa samodzielnie. We współczesnych aplikacjach internetowych osiągnięcie zamierzonego celu jest możliwe dzięki współpracy wielu komponentów. Stosujemy komponenty modelu, widoku oraz kontrolera. Wykorzystujemy także rozmaite klasy pomocnicze. Jednak w jaki sposób należy łączyć wszystkie te elementy, aby tworzyły jedną aplikację internetową? W jaki sposób komponenty mogą korzystać z tych samych informacji? Jak ukrywać pewne informacje? Jak zapewniać bezpieczeństwo informacji podczas przetwarzania wielowątkowego? Od odpowiedzi na te pytania może zależeć Twoja praca.

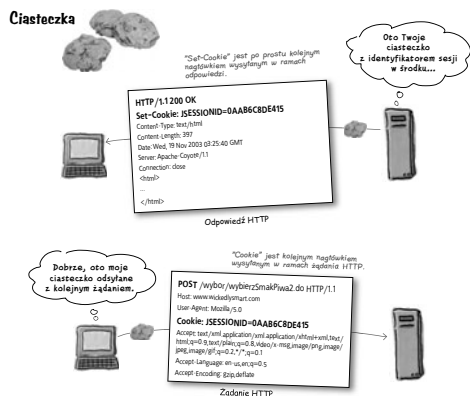


Cele egzaminu	176
Wybawieniem są parametry inicjalizacji i obiekt ServletConfig	177
Jak strona JSP może uzyskać dostęp do parametrów inicjalizacji serwletu?	183
Wybawieniem są parametry inicjalizacji kontekstu	185
Porównanie obiektów ServletConfig oraz ServletContext	187
Chcemy obiektu ServletContextListener	194
Przewodnik: prosty obiekt ServletContextListener	196
Kompilacja, wdrażanie i testowanie obiektu nasłuchującego	204
Kompletna historia — obiekt nasłuchujący kontekstu	206
Ośiem obiektów nasłuchujących nie tylko zdarzeń kontekstu	208
Czym dokładnie jest atrybut	213
Interfejs API atrybutów — ciemna strona atrybutów	217
Zasięg kontekstu nie zapewnia bezpieczeństwa wątków!	220
Analiza tego problemu w zwolnionym tempie...	221
Próba synchronizacji	223
Czy atrybuty sesji gwarantują bezpieczeństwo przetwarzania wielowątkowego?	226
Interfejs SingleThreadModel	229
Tylko atrybuty żądania i zmienne lokalne zapewniają bezpieczną wielowątkowość!	232
Atrybuty żądania i przydział zadań	233

6

Stan konwersacyjny

Serwery WWW nie mają pamięci krótkotrwałej. Zaraz po odesłaniu do nas odpowiedzi serwery WWW zapominają, kim jesteśmy. Kiedy wysyłamy kolejne żądanie, docelowy serwer WWW już nas nie rozpoznaje. Innymi słowy, serwery WWW nie pamiętają ani tego, czego żądaliśmy w przeszłości, ani tego, co do nas wysłały w ramach odpowiedzi. Zupełnie nic! Jednak czasami przechowywanie informacji o stanie konwersacji z klientem i korzystanie z nich podczas obsługi *wielu żądań* jest konieczne. Koszyk w sklepie internetowym nie mógłby działać, gdyby użytkownik musiał wybierać wszystkie towary i realizować zamówienie w ramach *jednego żądania*.

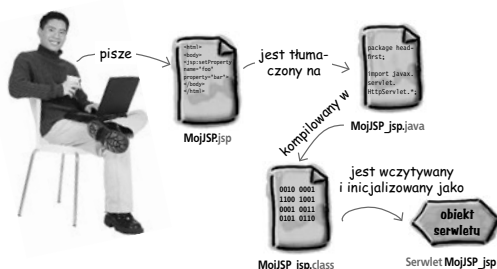


Cele egzaminu	252
To będzie dłuższa konwersacja, czyli jak działają sesje	254
Identyfikatory sesji, ciasteczka i pozostałe podstawy działania sesji	259
Przepisywanie adresów URL, sposób rozwiązywania problemu	265
Kiedy sesje stają się nieaktualne — eliminowanie zbędnych sesji	269
Czy ciasteczka mogą mieć także zastosowania inne niż obsługa sesji?	278
Najważniejsze momenty w życiu obiektu HttpSession	282
Nie zapominać o interfejsie HttpSessionBindingListener	284
Migracja sesji	285
Przykłady klas nasłuchujących	289

7

Być stroną JSP

Strona JSP staje się serwletem. Serwletem, którego *nie* musisz tworzyć. Kontener przegląda kod Twojej strony JSP, tłumaczy go na kod źródłowy języka programowania Java i kompiluje tak przetłumaczony kod na postać pełnowartościowej klasy serwletu Javy. Warto jednak wiedzieć, co dzieje się w czasie konwertowania Twojego kodu strony JSP na kod Javy. W ramach kodu JSP *można* co prawda umieszczać kod Javy, ale czy na pewno powinniśmy to robić? A jeśli w kodzie stron JSP nie umieścimy żadnych wyrażeń Javy, co *znajdzie* się w kodzie naszego serwletu? Przyjrzymy się sześciu rodzajom elementów JSP — z których każdy ma swoje przeznaczenie i, niestety, *odmienną składnię*. Dowiesz się, co i dlaczego można umieszczać w kodzie stron JSP. Dowiesz się także, czego *nie* powinieneś umieszczać w tym kodzie.

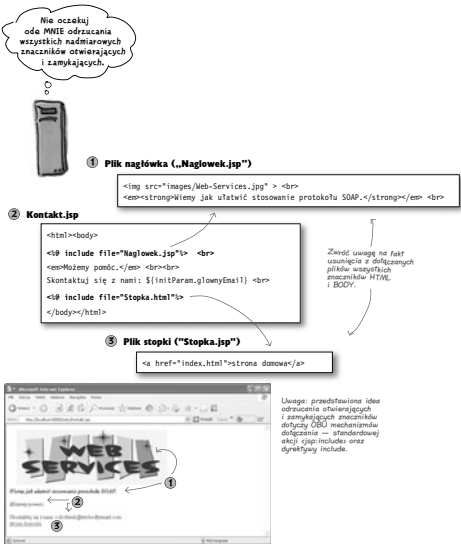


Cele egzaminu	310
Tworzymy prostą stronę JSP wykorzystującą zmienną out i dyrektywę page	311
Wyrażenia, zmienne i deklaracje JSP	316
Czas zapoznać się z wygenerowanym serwletem	324
Zmienna out nie jest jedynym obiektem domyślnym...	326
Cykl życia i inicjalizacja stron JSP	334
Skoro już poruszyliśmy ten temat... porozmawiajmy o trzech dyrektywach	342
Czy skryptlety można uznać za niebezpieczne? Oto EL	345
Ale zaczekaj... nie widzieliśmy jeszcze akcji	351

8

Strony bezskryptowe

Porzuc skrypty. Czy współpracujący z Tobą projektanci stron internetowych naprawdę muszą znać Javę? Czy sami oczekują od programistów Javy, aby byli jednocześnie np. grafikami? A jeśli nawet przyimiemy, że jesteś *jedynym* członkiem zespołu, czy rzeczywiście chciałbyś umieszczać rozbudowane fragmenty kodu Javy w swoich stronach JSP? Czyż nie nasuwa Ci się określenie „koszmar konserwacji oprogramowania”? Pisanie stron bezskryptowych jest nie tylko możliwe, ale stało się znacznie *prostsze* i bardziej elastyczne dzięki nowej specyfikacji JSP 2.0, a przede wszystkim wskutek wprowadzenia nowego języka wyrażeń (EL). Ponieważ język EL bazuje na językach JavaScript i XPATH, projektanci stron mogą go stosować bez najmniejszych problemów; zresztą także Ty go polubisz (kiedy już się do niego przyzwyczaisz). Jednak EL stwarza też pewne pułapki. Jego wyrażenia *wyglądają* co prawda jak wyrażenia Javy, jednak w rzeczywistości nimi nie są. Niektóre wyrażenia EL działają nawet inaczej niż wyrażenia Javy o identycznej składni, zatem warto mieć się na baczności!

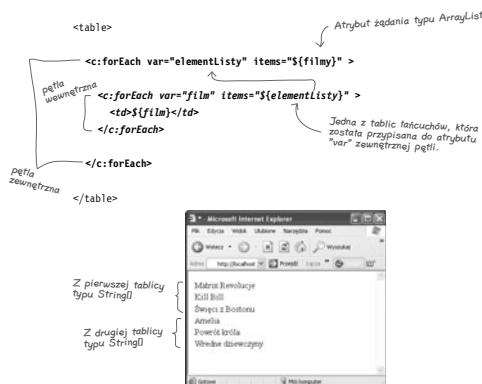


Cele egzaminu	372
Które atrybuty są <i>komponentami</i> <i>JavaBean</i> ?	373
Standardowe akcje: <i>useBean</i> , <i>getProperty</i> oraz <i>setProperty</i>	377
Czy można tworzyć polimorficzne referencje do komponentów?	382
Rozwiązaniem jest użycie atrybutu <i>param</i>	388
Konwersja właściwości	391
Uratował nas język wyrażeń (EL)	396
Stosowanie operatora kropki (.) do uzyskiwania dostępu do właściwości i map wartości	398
Operator [] stwarza dodatkowe możliwości (listy, tablice...)	400
Więcej szczegółów o operatorach kropki (.) oraz []	404
Obiekty domyślne języka EL	413
Funkcje języka EL i obsługa wartości null	420
Szablony wielokrotnego użytku — dwa rodzaje „dołączania”	430
Standardowa akcja <jsp:forward/>	444
Ona nie wie jeszcze o znacznikach JSTL (zapowiedź)	445
Przegląd standardowych akcji i wyrażeń dołączania	446

9

Potęga znaczników niestandardowych

Czasami potrzebujemy czegoś więcej niż tylko języka wyrażeń (EL) i akcji standardowych. Co będzie, jeśli zechcesz użyć pętli do przeszukania danych składowanych w tablicy i wyświetlenia po jednym elemencie w każdym wierszu generowanej dynamicznie tabeli HTML? Oczywiście doskonale *zdasz* sobie sprawę z możliwości błyskawicznego skonstruowania odpowiedniej pętli w skrypcie. Z drugiej strony, staramy się nie używać kodu skryptowego. To żaden problem. Kiedy język wyrażeń (EL) i akcje standardowe okazują się niewystarczające, zawsze można wykorzystać *znaczniki niestandardowe*. Ich stosowanie w kodzie stron JSP jest równie proste jak korzystanie z akcji standardowych. Co więcej, znaczniki, których najprawdopodobniej będziesz potrzebował, ktoś już napisał i umieścił w standardowej bibliotece znaczników JSP (ang. *JSP Standard Tag Library*, w skrócie *JSTL*). W tym rozdziale dowiesz się, jak *używać* znaczników niestandardowych, a w następnym — jak tworzyć własne znaczniki.



Cele egzaminu	468
Pętle bez skryptów, <code><c:forEach></code>	474
Kontrola warunkowa z użyciem znaczników <code><c:if></code> oraz <code><c:choose></code>	479
Zastosowanie znaczników <code><c:set></code> i <code><c:remove></code>	483
Znacznik <code><c:import></code> , czyli trzeci sposób dołączania treści	488
Modyfikowanie dołączanej zawartości	490
Realizacja tego samego zadania za pomocą znacznika <code><c:param></code>	491
Znacznik <code><c:url></code> jako narzędzie realizacji wszystkich zadań związanych z obsługą hiperłączy	493
Tworzenie własnych stron o błędach	496
Znacznik <code><c:catch></code> , który jest jak... konstrukcja try-catch	500
Co będzie, jeśli uznamy za niezbędne użycie znacznika SPOZA biblioteki JSTL?	503
Zwróć uwagę na element <code><rtexprvalue></code>	508
Co może się znaleźć w ciele znacznika	510
Klasa obsługująca znacznik, deskryptor TLD i strona JSP	511
Podelement <code><uri></code> elementu <code>taglib</code> jest tylko nazwą, nie lokalizacją	512
Kiedy strona JSP wykorzystuje więcej niż jedną bibliotekę znaczników	515

10

Kiedy JSTL nie wystarcza...

Czasami JSTL i standardowe akcje nie wystarczają. Kiedy trzeba zrobić coś niestandardowego, a nie chcesz uciekać się do stosowania skryptu, możesz stworzyć *własne* procedury obsługi znaczników. Dzięki temu projektanci stron będą mogli używać w projektowanych stronach Twoich znaczników, a całą „czarną robotę” będzie, w niewidoczny sposób, realizować Twoja *klasa* obsługująca. Z drugiej strony, takie rozwiązanie wymaga od Ciebie sporo nauki, ponieważ własne, niestandardowe znaczniki można tworzyć na aż trzy różne sposoby. Dwa spośród nich (chodzi o *znaczniki proste* oraz *pliki znaczników*) wprowadzono z myślą o naszej wygodzie dopiero w specyfikacji JSP 2.0.

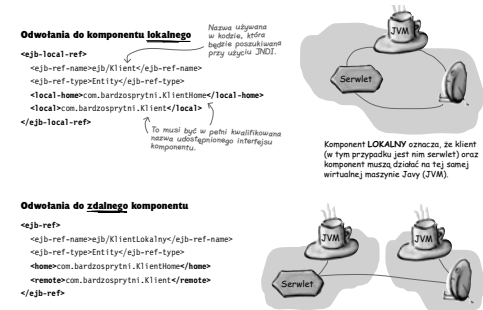


Cele egzaminu	528
Pliki znaczników — podobnie jak znacznik <code><include></code> , tylko lepiej	530
Gdzie kontener szuka plików znaczników?	537
Proste klasy obsługujące znaczniki	541
Prosty znacznik z zawartością	542
Co zrobić, jeśli w zawartości znacznika pojawia się wyrażenie?	547
Wciąż musimy znać klasyczne klasy obsługujące znaczniki niestandardowe	557
Bardzo prosta, klasyczna klasa obsługi znacznika niestandardowego	559
Cykl życia znacznika klasycznego zależy od zwracanych wartości	564
Interfejs <code>IterationTag</code> pozwala na wielokrotne przetwarzanie zawartości znacznika	565
Wartości domyślne zwracane przez metody klasy <code>TagSupport</code>	567
Interfejs <code>DynamicAttributes</code>	584
Interfejs <code>BodyTag</code> udostępnia dwie kolejne metody	591
A co, jeśli znaczniki współpracują ze sobą?	595
Interfejs programowy <code>PageContext</code> dla klas obsługi znaczników	605

11

Wdrażanie aplikacji internetowych

W końcu Twoja aplikacja jest gotowa. Strony zostały dopracowane w najdrobniejszych szczegółach, kod jest przetestowany i zoptymalizowany, a termin... minął dwa tygodnie temu. Ale gdzie to wszystko należy umieścić? Jest tyle różnych katalogów, tyle niezrozumiałych reguł. Jak *powinieneś* nazwać swoje katalogi? Jakich nazw użyłby *klient*? Do jakich zasobów tak naprawdę będzie się odwoływać klient i skąd kontener ma wiedzieć, gdzie należy ich szukać?

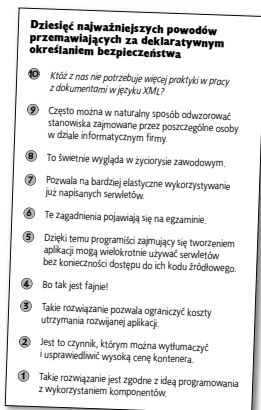


Cele egzaminu	630
Podstawowe zagadnienia związane z wdrażaniem różnych elementów aplikacji	631
Pliki WAR	640
Jak NAPRAWDĘ działają odwzorowania serwletów?	644
Konfiguracja plików powitalnych w deskrypcorze wdrożenia	650
Konfiguracja stron błędów w deskrypcorze wdrożenia	654
Konfigurowanie inicjalizacji serwletu w deskrypcorze wdrożenia	656
Tworzenie stron JSP zgodnych z zasadami konstrukcji dokumentów XML: dokumenty JSP	657

12

Zachowaj to w tajemnicy, ukryj w bezpiecznym miejscu

Twoja aplikacja internetowa jest w *niebezpieczeństwie*. Problemy czyhają w każdym zakamarku sieci. Nie chcesz chyba, aby ci źli faceci podsłuchiwali transakcje realizowane w Twoim sklepie internetowym i przechwytywali podawane numery kart kredytowych? Nie chcesz też, by byli w stanie przekonać Twój serwer, iż tak naprawdę są Bardzo Ważnymi Klientami Liczącymi Na Bardzo Duże Upusty. I w końcu nie chcesz, by *ktokolwiek* (niezależnie od zamiarów) miał dostęp do poufnych informacji o pracownikach. Czy Janek z działu marketingu naprawdę musi wiedzieć, że Lucyna z działu technicznego zarabia trzy razy więcej od niego?

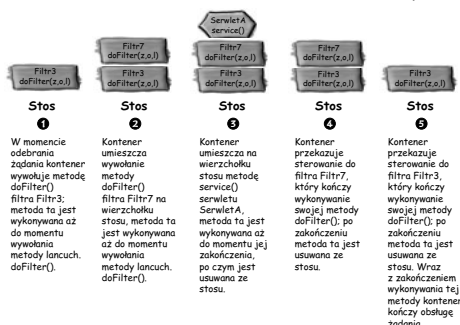


Cele egzaminu	678
Wielka Czwórka świata zabezpieczeń serwetów	681
Jak realizować uwierzytelnianie w świecie protokołu HTTP?	684
Dziesięć najważniejszych powodów przemawiających za deklaratywnym określeniem bezpieczeństwa	687
Kto implementuje zabezpieczenia aplikacji internetowej?	688
Autoryzacja: role i ograniczenia	690
CZTERY typy uwierzytelniania	705
Zabezpieczanie przesyłanych informacji — protokół HTTPS śpieszy z pomocą	710
Jak należy wybiórczo i deklaratywnie implementować poufność i integralność danych?	712

13

Potęga filtrów

Filtry umożliwiają przechwytywanie żądań. A skoro można przechwycić *żądanie*, można także kontrolować *odpowiedź*. Ale najlepsze w tym wszystkim jest to, że serwet nie ma o tym **najmniejszego pojęcia**. Serwet nigdy nie wie, czy coś się zdarzyło w czasie dzielącym odebranie żądania przez kontener od wywołania metody `service()` serwetu. Co to oznacza dla Ciebie? Dłuższe wakacje. Ponieważ czas, który musiałbyś poświęcić na modyfikowanie tylko *jednego* z istniejących serwetów, możesz poświęcić na napisanie i skonfigurowanie filtra, który będzie miał wpływ na *wszystkie* Twoje serwety. Może chciałbyś dodać opcję śledzenia żądań we *wszystkich* serwetach tworzących aplikację? Nie ma żadnego problemu. A może chciałbyś w określony sposób modyfikować wyniki generowane przez wszystkie *serwety* wchodzące w skład aplikacji? To także żaden problem. A co najlepsze — nie musisz przy tym w *żaden sposób* modyfikować kodu serwetów.

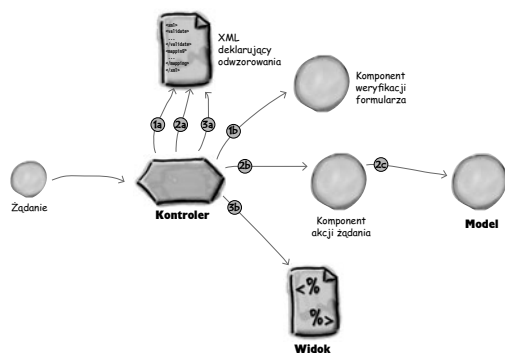


Cele egzaminu	730
Tworzenie filtra śledzącego żądania	735
Cykl życia filtrów	736
Deklarowanie i określanie kolejności filtrów	738
Kompresja wyników przy wykorzystaniu filtra operującego na odpowiedzi	741
Opakowania są świetne!	747
Prawdziwy kod filtra kompresji odpowiedzi	750
Kod opakowania kompresji odpowiedzi	752

14

Korporacyjne wzorce projektowe

Ktoś to już wcześniej zrobił. Jeśli właśnie zaczynasz tworzyć aplikacje internetowe w języku Java, masz dużo szczęścia. Możesz czerpać z wiedzy dziesiątek tysięcy programistów, którzy od dawna się tym zajmują i mają już swoje firmowe koszulki. Wykorzystując wzorce projektowe, zarówno te związane z platformą J2EE, jak i wszelkie *inne*, możesz uprościć swój kod i swoje życie. W świecie aplikacji internetowych najważniejszym wzorcem projektowym jest MVC, który zastosowano między innymi w bardzo popularnym frameworku Struts (stworzonym z myślą o wsparciu programistów tworzących elastyczne i łatwe w utrzymaniu kontrolery frontonów serwetów). Wykorzystanie pracy *innych* jesteś winien samemu sobie, dzięki temu będziesz mógł poświęcić więcej czasu na ważniejsze sprawy.



Cele egzaminu	766
Sprzętowe i programowe argumenty na rzecz wzorów projektowych	767
Przegląd zasad związanych z projektowaniem oprogramowania	772
Wzorce wspomagają zdalne komponenty modelu	773
JNDI oraz RMI — krótka prezentacja	775
Delegat biznesowy jest obiektem pośredniczącym	781
Czas poznać wzorec Transfer Object?	787
Wzorce warstwy biznesowej — krótki przegląd	789
Nasz pierwszy wzorec po raz wtóry — MVC	790
Tak! To Struts (i wzorec Front Controller) w zarysie	795
Przystosowanie aplikacji piwnej do korzystania z frameworku Struts	798
Przegląd wzorców projektowych	806

A



BAR KAWOWY

Końcowy Egzamin Próbnny Baru Kawowego. To jest to. 69 pytań. Charakter, zagadnienia i poziom trudności jest niemal taki sam jak na *prawdziwym egzaminie*. Możesz nam wierzyć.

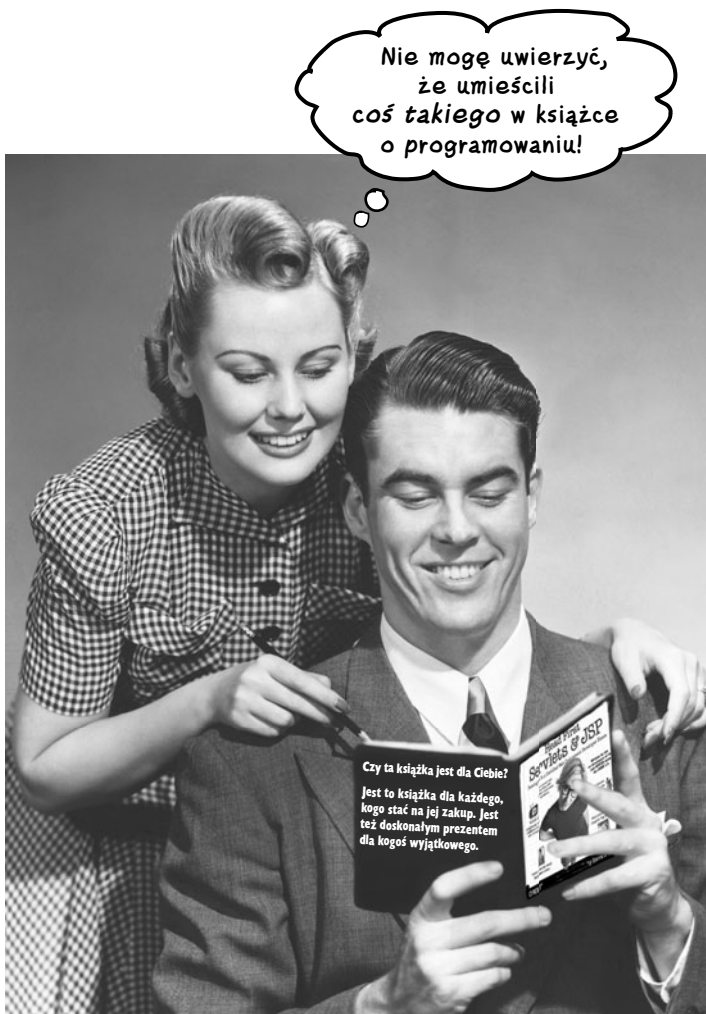
Końcowy egzamin próbny	819
Odpowiedzi	856

S

Skorowidz

Jak korzystać z niniejszej książki?

Wprowadzenie



W tej części książki odpowiadamy na palące pytanie:
„Dlaczego autorzy **UMIEŚCILI** takie rzeczy w książce
o programowaniu?”.

Dla kogo jest ta książka?

Jeśli możesz odpowiedzieć twierdząco na wszystkie poniższe pytania:

- 1 Czy potrafisz **programować w Javie** (nie musisz być ekspertem w tej dziedzinie)?
- 2 Lubisz samodzielne majstrowanie — naukę przez wykonywanie eksperymentów zamiast biernej lektury? Chcesz **opanować, zrozumieć i zapamiętać** technologie serwletów i stron JSP, a być może także **przystąpić do egzaminu SCWCD z wiedzy o platformie J2EE 1.5**?
- 3 Czy wolisz **wciągające dyskusje przy posiłku** od **drętwych i nudnych publikacji akademickich**?

— ta książka jest właśnie dla Ciebie.

Kto raczej nie powinien sięgać po tę książkę?

Jeśli możesz odpowiedzieć twierdząco na którekolwiek z poniższych pytań:

- 1 Nie masz **żadnego doświadczenia w programowaniu w Javie**? Nie musisz co prawda dysponować naprawdę zaawansowaną wiedzą; jeśli jednak brakuje Ci choćby podstawowej znajomości tego języka, sięgnij raczej, najlepiej od razu, po książkę **Head First Java**¹, by następnie wrócić do lektury niniejszej pracy.
- 2 Jesteś **bardzo doświadczonym programistą Javy** poszukującym leksykonu?
- 3 Jesteś **weteranem technologii Java EE** zainteresowanym ultrazaawansowanymi technikami serwerowymi, precyzyjną prezentacją procedur obowiązujących na różnych serwerach, architekturami korporacyjnymi i długimi, skomplikowanymi przykładami zaczerpniętymi z rzeczywistych aplikacji?
- 4 Boisz się **spróbować czegoś nowego**? Wolałbyś raczej poddać się leczeniu kanałowemu zęba niż zdecydować na połączenie pasków ze szkodłą kratą? Czy naprawdę uważasz, że nie można traktować poważnie książki technicznej, w której spersonifikowano komponenty?

— niniejsza książka nie jest dla Ciebie.



[notatka z działu marketingu: ta książka jest dla wszystkich, którzy mogą sobie pozwolić na jej kupno.]

¹ Polskie wydanie: *Head First Java. Edycja polska*, Helion, 2004 — przyp. tłum.

Wiemy, co sobie myślisz

„To coś ma być poważną książką o programowaniu w Javie?”

„Po co te wszystkie obrazki?”

„Czy w taki sposób można się czegośkolwiek *nauczyć*?”

Wiemy także, co sobie myśli Twój mózg

Twój mózg pragnie nowości. Zawsze szuka, przegląda i *wyczekuje* na coś niezwykłego. Tak został stworzony i właśnie to pomaga mu przetrwać.

Co w takim razie Twój mózg robi z tymi wszystkimi rutynowymi, zwyczajnymi, normalnymi informacjami, które do niego docierają? Otóż robi wszystko, co w jego mocy, aby nie przeszkadzały w jego *najważniejszym* zadaniu — zapamiętywaniu rzeczy, które mają *prawdziwe* znaczenie. Twój mózg nie traci czasu i energii na zapamiętywanie nudnych informacji, które nigdy nie przechodzą przez swoisty filtr: „to jest oczywiście całkowicie nieważne”.

Skąd Twój mózg *wie*, co jest istotne? Przypuśćmy, że jesteś na codziennej przechadzce i nagle staje przed Tobą tygrys. Co się wówczas dzieje w Twojej głowie?

Neurony płoną. Emocje szaleją. *Adrenalina napływa falami*.

I właśnie stąd Twój mózg wie, że...

To musi być ważne! Nie zapominaj o tym!

Wyobraź sobie teraz, że siedzisz w domu lub w bibliotece. Jesteś w bezpiecznym miejscu — przytulnym i pozbawionym tygrysów. Uczysz się. Przygotowujesz się do egzaminu. Albo poznajesz jakiś trudny problem techniczny, którego rozwiązanie, według szefa, powinno zająć Ci tydzień, a najwyżej dziesięć dni.

Jest tylko jeden drobny problem. Twój mózg stara się pomóc Ci. Próbuje zagwarantować, że te w *oczywisty sposób* nieistotne informacje nie zajmą cennych zasobów w Twojej głowie. Zasobów, które powinny zostać wykorzystane na zapamiętanie *naprawdę ważnych* rzeczy. Takich jak tygrysy. Takich jak zagrożenie, jakie niesie ze sobą pożar. Takich jak to, że już nigdy w życiu nie powinieneś jeździć na snowboardzie w krótkich spodenkach.

Co gorsza, nie można po prostu powiedzieć mózgowi: „Hej, mózgu, dziękuję ci bardzo, ale niezależnie od tego, jak nudna jest ta książka i jak mizerne są wskazania na emocjonalnej skali Richtera, naprawdę chciałbym zapamiętać wszystkie te informacje”.

Twój mózg myśli, że właśnie *TO* jest istotne



Wspaniale.
Pozostało jeszcze
tylko 900 głupich, nudnych
i dętnych stron.

Twój mózg uważa,
że tego nie warto
zapamiętywać.



Wyobrażamy sobie, że Czytelnik tej książki jest uczniem

A zatem chcesz się *czegoś nauczyć*? W pierwszej kolejności powinieneś więc to *poznać*, by następnie spróbować tego nie *zapomnieć*. Nie chodzi tylko o wtłoczenie do głowy suchych faktów. Najnowsze badania z zakresu przyswajania informacji, neurobiologii i psychologii nauczania pokazują, że *uczenie się* wymaga czegoś więcej niż tylko czytania tekstu. My wiemy, co potrafi pobudzić nasze mózgi do działania.

Oto wybrane zasady obowiązujące w serii Head First:

Wyobraź to sobie wizualnie. Rysunki są znacznie łatwiejsze do zapamiętania niż same słowa i sprawiają, że nauka staje się dużo bardziej efektywna (studia nad przypominaniem sobie i przekazywaniem informacji dowodzą, że użycie rysunków poprawia efektywność zapamiętywania o 89%). Co więcej, rysunki sprawiają, że informacje stają się znacznie bardziej zrozumiałe. Wystarczy **umieścić słowa bezpośrednio na lub w okolicach rysunku**, do którego się odnoszą, a nie na następnej stronie, a prawdopodobieństwo, że osoby uczące się będą w stanie rozwiązać problem, którego te słowa dotyczą, wzrośnie niemal dwukrotnie.

Stosuj styl konwersacji i personifikacji. Według najnowszych badań w testach końcowych studenci uzyskiwali wyniki o 40% lepsze, jeśli treść była przekazywana w sposób bezpośredni, w pierwszej osobie i w konwencji rozmowy, a nie w sposób formalny. Zamiast wykładania opowiadaj historyjki. Używaj zwyczajnego języka. Nie traktuj samego siebie zbyt poważnie.

Kiedy jesteś bardziej zainteresowany tematem: podczas ożywionej dyskusji przy obiedzie czy w czasie wykładu?

Bycie metodą abstrakcyjną to naprawdę nic miłego. Wyobraź sobie, że jesteś zupełnie pusty w środku.

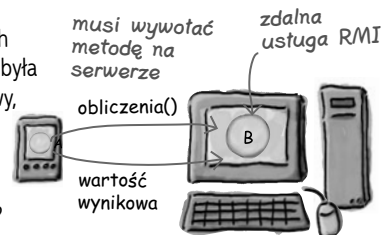
abstract void roam();
Metoda nie ma ciała! Kończy się średnikiem.

Zmusz uczniów do głębszych przemyśleń. Innymi słowy, jeśli nie zmusisz neuronów do aktywnego wysiłku, w Twojej głowie nie zdarzy się nic wielkiego. Czytelnik musi być zmotywowany, zaangażowany, zaciekawiony i podekscytowany rozwiązywaniem problemów, wyciąganiem wniosków i zdobywaniem nowej wiedzy. A osiągnięcie tego wszystkiego jest możliwe poprzez stawianie wyzwań, wykonywanie ćwiczeń i zadawanie pytań zmuszających do zastanowienia oraz poprzez zmuszanie do działań, które wymagają zaangażowania obu półkul mózgowych i wielu zmysłów.

Zdobądź — i zachowaj na dłużej — uwagę i zainteresowanie czytelnika.

Każdy znalazł się kiedyś w sytuacji, gdy bardzo chciał się czegoś nauczyć, lecz zasypiał po przeczytaniu pierwszej strony. Mózg zwraca uwagę na rzeczy niezwykłe, interesujące, dziwne, przykuwające wzrok, nieoczekiwane. Okazuje się, że nawet poznawanie nowych zagadnień technicznych wcale nie musi być nudne. Jeśli będzie to zagadnienie interesujące, Twój mózg przyswoi je sobie znacznie szybciej.

Wyzwól emocje. Jak już wiemy, zdolności do zapamiętywania informacji są w dużej mierze zależne od ich ładunku emocjonalnego. Zapamiętujemy to, na czym nam zależy. Zapamiętujemy w sytuacjach, w których coś odczuwamy. Oczywiście nie mamy tu na myśli wzruszających historii o chłopcu i jego psie. Chodzi nam o emocje takie jak zaskoczenie, ciekawość, radość, podekscytowanie, „a niech to...” i uczucie satysfakcji — „jestem wielki!” — jakie odczuwamy po poprawnym rozwiązaniu zagadki, nauczaniu się czegoś, co powszechnie uchodzi za trudne, lub zdaniu sobie sprawy, że znamy więcej szczegółów technicznych niż Zenek z działu inżynierii.



Metapoznanie — myślenie o myśleniu

Kiedy naprawdę chcesz się czegoś nauczyć i chcesz to zrobić szybciej i dokładniej, zwróć uwagę na to, jak zwracasz swoją uwagę. Myśl o tym, jak myślisz. Poznawaj sposób, w jaki się uczysz.

Większość z nas w dzieciństwie nie uczestniczyła w zajęciach z metapoznania ani teorii nauczania. Oczekiwano od nas, że będziemy się *uczyć*, jednak nie uczono nas, *jak* mamy to robić.

Przyjmujemy jednak, że skoro trzymasz w ręku tę książkę, naprawdę chcesz się nauczyć pisania aplikacji internetowych w Javie, a być może także przygotować się do egzaminu SCWCD. I prawdopodobnie nie chcesz na to stracić zbyt wiele czasu. Jeśli to, co tutaj przeczytasz, chcesz potem w ten czy inny sposób wykorzystać, konieczne musisz to *zapamiętać*. Warunkiem zapamiętania tej wiedzy jest *zrozumienie* prezentowanych zagadnień. Aby w jak największym stopniu skorzystać z tej książki, ale też *każdej* innej i wszelkich prób uczenia się czegokolwiek, musisz wziąć odpowiedzialność za swój mózg. Myśl o *ty*m, czego się uczysz.

Sztuczka polega na tym, aby przekonać mózg, że poznawany materiał jest Naprawdę Ważny. Kluczowy dla Twojego dobrego samopoczucia. Nie mniej ważny od tygrysa. W przeciwnym razie będziesz prowadzić nieustającą wojnę z własnym mózgiem, który ze wszystkich sił będzie unikał utrwalania nowej wiedzy.

Jak w takim razie zmusić mózg do traktowania Javy jak głodnego tygrysa?

Można to zrobić w sposób wolny i męczący lub szybki i bardziej efektywny. Sposób wolny polega na wielokrotnym powtarzaniu.

Oczywiście wiesz, że jesteś w stanie zapamiętać nawet najnudniejsze zagadnienie, mozołnie je „wkuwając”.

Po odpowiedniej liczbie powtórzeń Twój mózg stwierdzi: „*Wydaje się, że to nie jest dla niego szczególnie ważne, lecz w kółko to czyta i powtarza, więc przypuszczam, że jakąś wartość musi to jednak mieć*”.

Szybszy sposób polega na zrobieniu **czegokolwiek, co zwiększy aktywność mózgu**, zwłaszcza jeśli czynność ta wyzwoli kilka różnych *typów* aktywności. Wszystkie zagadnienia, o których pisaliśmy na poprzedniej stronie, są kluczowymi elementami rozwiązania i udowodniono, że wszystkie potrafią pomóc zmusić mózg, aby pracował na Twoją korzyść. Badania wykazują, że na przykład umieszczenie słów *na* opisywanych rysunkach (a nie w innych miejscach tekstu na stronie, na przykład w nagłówku lub wewnątrz akapitu) sprawia, że mózg stara się zrozumieć relację pomiędzy słowami a rysunkiem, co z kolei może rozgrzać nasze neurony do czerwoności. Większa aktywność neuronów to z kolei większa szansa, że mózg uzna informacje za warte uwagi i, ewentualnie, zapamiętania.

Prezentowanie informacji w formie konwersacji pomaga, ponieważ Czytelnicy zdają się wykazywać większe zainteresowanie w sytuacjach, gdy mają wrażenie udziału w dyskusji — gdy czują, że oczekuje się od nich śledzenia jej przebiegu i brania w niej czynnego udziału. Zadziwiające jest to, iż mózg zdaje się nie zważać na to, że rozmowa jest prowadzona z książką! Z drugiej strony, jeśli sposób przedstawiania informacji jest formalny i suchy, mózg postrzega to tak samo jak w sytuacji, gdy uczestniczysz w wykładzie na sali pełnej sennych słuchaczy. Nie ma potrzeby wykazywania jakiegokolwiek aktywności.

Okazuje się jednak, że rysunki i przedstawianie informacji w formie rozmowy to dopiero początek.

Zastanawiam się, jak zmusić mózg do zapamiętania tych informacji...



Oto, co zrobiliśmy:

Wprowadziliśmy mnóstwo **rysunków**, ponieważ Twój mózg zwraca większą uwagę na obrazy niż na tekst. Jeśli chodzi o mózg, to faktycznie jeden obraz jest wart 1024 słowa. Wszędzie tam, gdzie prezentowany tekst można było zilustrować rysunkiem, umieszczaliśmy tekst *na* rysunku, ponieważ mózg działa bardziej efektywnie, gdy tekst jest *wewnątrz* tego, co opisuje, niż kiedy jest umieszczony w innym miejscu i stanowi część większego fragmentu tekstu.

Stosowaliśmy **powtórzenia**, wielokrotnie podając tę samą informację na różne sposoby i przy wykorzystaniu różnych środków przekazu oraz odwołując się do *różnych zmysłów*. Wszystko po to, aby zwiększyć szansę, że informacja zostanie zakodowana w większej liczbie obszarów Twojego mózgu.

Korzystaliśmy z pomysłów i rysunków w **nieoczekiwany** sposób, ponieważ Twój mózg oczekuje i pragnie nowości; poza tym staraliśmy się zawrzeć w nich *choć trochę* **emocji**, gdyż mózg jest skonstruowany w taki sposób, iż zwraca uwagę na biochemię związaną z emocjami. Prawdopodobieństwo zapamiętania czegoś jest większe, jeśli „to coś” sprawia, że cokolwiek *poczujemy*, nawet jeśli to uczucie nie jest niczym więcej niż lekkim **rozbawieniem**, **zaskoczeniem** lub **zainteresowaniem**.

Używaliśmy bezpośrednich zwrotów i przekazywaliśmy treści w **formie konwersacji**, gdyż mózg zwraca większą uwagę, jeśli uważa, że prowadzisz rozmowę, niż gdy jesteś jedynie biernym słuchaczem prezentacji. Mózg działa w ten sposób, nawet gdy *czytasz* rozmowę.

Zamieściliśmy w książce ponad 40 **ćwiczeń**, ponieważ mózg uczy się i zapamiętuje zdecydowanie lepiej to, co **robimy**, niż to, o czym *czytamy*. Poza tym podane ćwiczenia stanowią wyzwania, choć nie są przesadnie trudne, gdyż właśnie takie preferuje większość osób.

Zastosowaliśmy **wiele stylów nauczania**, ponieważ tak jak *Ty* możesz preferować instrukcje opisujące sposób postępowania krok po kroku, tak ktoś inny może woleć ogólną analizę danego zagadnienia, a ktoś jeszcze inny — przejrzenie przykładowego fragmentu kodu. Jednak niezależnie od ulubionego sposobu nauki *każdy* skorzysta na tym, że te same informacje będą przedstawiane kilkakrotnie w różny sposób.

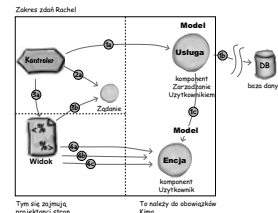
Podaliśmy informacje przeznaczone dla **obu półkul Twojego mózgu**, gdyż im bardziej mózg będzie zaangażowany, tym większe jest prawdopodobieństwo nauczania się i zapamiętania podawanych informacji i tym dłużej możesz koncentrować się na nauce. Ponieważ angażowanie tylko jednej półkuli mózgu często oznacza, że druga będzie mogła odpocząć, będziesz mógł uczyć się bardziej produktywnie przez dłuższy czas.

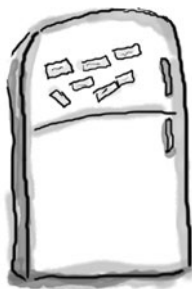
Dodatkowo zamieściliśmy **opowiadania** i ćwiczenia prezentujące **więcej niż jeden punkt widzenia**, ponieważ mózg uczy się dokładniej, gdy jest zmuszony do przetwarzania i podawania własnej opinii.

Postawiliśmy przed Tobą **wyzwania**, zarówno poprzez włączenie ćwiczeń, jak i stawiając **pytania**, na które nie zawsze można odpowiedzieć w prosty sposób; a to dlatego, że mózg uczy się i pamięta, gdy musi *popracować* nad czymś (podobnie — nie możemy zdobyć dobrej *formy fizycznej*, *obserwując* ćwiczenia w telewizji). Jednak dołożyliśmy wszelkich starań, aby zapewnić, że gdy pracujesz, robisz *dokładnie to, czego trzeba*, aby **ani jeden dendryt nie musiał** przetwarzać trudnego przykładu bądź analizować tekstu zbyt lapidarnego lub napisanego trudnym żargonem.

Personifikowaliśmy tekst. W opowiadaniach, przykładach, rysunkach i wszelkich innych możliwych miejscach tekstu staraliśmy się personifikować tekst, gdyż *jesteś* osobą, a Twój mózg zwraca większą uwagę na *osoby* niż na *rzeczy*.

Zastosowaliśmy metodę **80/20**, zakładamy bowiem, że to nie jest książka dla osób, które mają zamiar pisać doktorat na temat stron JSP. Zatem nie zajmujemy się w niej *wszelkimi możliwymi* zagadnieniami, a jedynie tymi, z którymi faktycznie możesz się *zetknąć*.





Oto, co TY możesz zrobić, aby zmusić swój mózg do posłuszeństwa

A zatem zrobiliśmy, co w naszej mocy. Reszta zależy od Ciebie. Możesz zacząć od poniższych porad. Posłuchaj swojego mózgu i określ, które sprawdzają się w Twoim przypadku, a które nie przynoszą pozytywnych rezultatów. Spróbuj czegoś nowego.

*Wytnij te porady
i przyklej na lodówce.*

❶ **Zwolnij. Im więcej rozumiesz, tym mniej musisz zapamiętać.**

Nie ograniczaj się jedynie do *czytania*. Przerwij na chwilę lekturę i pomyśl. Kiedy znajdziesz w tekście pytanie, nie zaglądaj od razu na stronę odpowiedzi. Wyobraź sobie, że ktoś *faktycznie* zadaje Ci pytanie. Im bardziej zmusisz swój mózg do myślenia, tym większa będzie szansa, że się nauczysz i zapamiętasz dane zagadnienie.

❷ **Wykonuj ćwiczenia. Rób notatki.**

Umieszczaliśmy je w tekście, jednak jeśli zrobilibyśmy je za Ciebie, to niczym nie różniłoby się to od sytuacji, w której ktoś za Ciebie wykonywałby ćwiczenia fizyczne. I nie ograniczaj się jedynie do *czytania* ćwiczeń. **Używaj ołówka.** Można znaleźć wiele dowodów na to, że fizyczna aktywność *podczas* nauki może poprawić jej wyniki.

❸ **Czytaj fragmenty oznaczone jako „Nie ma niemądrych pytań”.**

Nie należy ich traktować jak fragmenty opcjonalne — *stanowią część podstawowej zawartości książki!* Nie pomijaj ich.

❹ **Niech lektura tej książki będzie ostatnią rzeczą, jaką robisz przed pójściem spać. A przynajmniej ostatnią rzeczą stanowiącą wyzwanie intelektualne.**

Pewne elementy procesu uczenia się (a w szczególności przenoszenie informacji do pamięci długotrwałej) mają miejsce *po* odłożeniu książki. Twój mózg potrzebuje trochę czasu dla siebie i musi dodatkowo przetworzyć dostarczone informacje. Jeśli w czasie niezbędnym do tego dodatkowego „przetwarzania” zmusisz go do innej działalności, część z przyswojonych informacji może zostać utracona.

❺ **Pij wodę. Dużo wody.**

Twój mózg pracuje najlepiej, gdy dostarczasz mu dużo płynów. Odwodnienie (które może następować nawet zanim poczujesz pragnienie) obniża zdolność percepcji.

❻ **Rozmawiaj o zdobywanych informacjach. Na głos.**

Mówienie aktywuje odmienne fragmenty mózgu. Jeśli próbujesz coś zrozumieć lub zwiększyć szanse zapamiętania informacji na dłużej, powtarzaj je na głos. Jeszcze lepiej, gdy będziesz starał się je na głos komuś wytłumaczyć. W ten sposób nie tylko będziesz się uczył szybciej, ale też przy odrobinie szczęścia odkryjesz to, czego nie dostrzegłeś, czytając treść książki.

❼ **Słuchaj swojego mózgu.**

Uważaj, kiedy Twój mózg staje się przeciążony. Jeśli zauważysz, że zaczynasz czytać pobieżnie i zapominać to, o czym przeczytałeś przed chwilą, najwyższy czas na przerwę. Po przekroczeniu pewnego punktu, nie będziesz się uczył szybciej, „wciskając” do głowy więcej informacji. Co gorsza, może to zaszkodzić całemu procesowi przyswajania wiedzy.

❽ **Poczuj coś.**

Twój mózg musi wiedzieć, że to, czego się uczysz, *ma znaczenie*. Z zaangażowaniem śledź zamieszczane w tekście opowiadania. Nadawaj własne tytuły zdjęciom. Zalewanie się łzami ze śmiechu po przeczytaniu głupiego dowcipu *i tak jest lepsze* od braku jakiegokolwiek reakcji.

❾ **Wykonaj Końcowy Egzamin Próbnny Baru Kawowego dopiero PO przeczytaniu całej książki.**

Jeśli przystąpisz do niego zbyt wcześnie, nie będziesz w stanie precyzyjnie określić, w jakim stopniu jesteś przygotowany do zdawania prawdziwego egzaminu. Poczekaj do momentu, gdy uznasz, że jesteś już niemal gotowy, i dopiero wtedy spróbuj go wykonać, dając sobie dokładnie 135 minut — czyli dokładnie tyle, ile trwa prawdziwy egzamin SCWCD.

Czego potrzebujesz, aby skorzystać z tej książki?

Oczywiście oprócz mózgu i ołówka **powinieneś mieć także Javę, Tomcata 5 i komputer.**

Nie potrzebujesz żadnych dodatkowych narzędzi, jak zintegrowane środowiska programistyczne (ang. *Integrated Development Environment, IDE*). Gorąco zachęcamy do rezygnacji na czas lektury niniejszej książki ze wszelkich narzędzi z wyjątkiem prostego edytora. Środowiska programistyczne oferujące możliwość tworzenia serwetów i stron JSP mogą ukrywać przed programistą wiele ważnych (i sprawdzanych na egzaminie) szczegółów, stąd najlepszym rozwiązaniem jest samodzielne, ręczne tworzenie całego kodu. Kiedy już zrozumiesz, co się dzieje, będziesz mógł zacząć korzystać z narzędzi, które częściowo automatyzują proces tworzenia serwetów i stron JSP oraz ich wdrażania. Jeśli wiesz już, jak korzystać z programu Ant, po przeczytaniu rozdziału 3. możesz zacząć wykorzystywać to narzędzie w procesie wdrażania aplikacji. Z drugiej strony, odradzamy stosowanie Anta do czasu dobrego opanowania i zapamiętania kompletnej struktury wdrażanych aplikacji.



POBIERANIE TOMCATA

- Jeśli jeszcze nie dysponujesz platformą **Java SE 1.5**, koniecznie musisz się w nią zaopatrzyć.
- Jeśli nie masz jeszcze Tomcata 5, pobierz go ze strony internetowej:
<http://tomcat.apache.org/>.
Kliknij łącze *Tomcat 5.5* w części *Download* w lewej części przytoczonej strony domowej.
- Przewiń stronę w dół, do części *Binary Distributions* i pobierz właściwą wersję. Jeśli nie wiesz, która wersja będzie dla Ciebie najlepsza, wybierz jedną z dystrybucji w punkcie *Core* — to powinno wystarczyć.
- Zapisz pobrany plik w katalogu tymczasowym.
- Zainstaluj Tomcata.
W systemie Windows wystarczy dwukrotnie kliknąć plik wykonywalny programu instalacyjnego i postępować zgodnie z instrukcjami kreatora instalacji.
W pozostałych systemach należy rozpakować pobrany plik, umieszczając jego zawartość w katalogu docelowym Tomcata.
- Aby ułatwić sobie wykonywanie ćwiczeń zamieszczonych w niniejszej książce, sugerujemy umieszczenie Tomcata w katalogu *tomcat* (lub przynajmniej utworzenie aliasu *tomcat* wskazującego na faktyczny katalog domowy zainstalowanego serwera Tomcat).
- Zdefiniuj zmienne środowiskowe **JAVA_HOME** oraz **TOMCAT_HOME** w sposób właściwy dla używanego systemu operacyjnego.
- Powinieneś także pobrać kopie następujących specyfikacji; choć nie są one konieczne, by podchodzić do egzaminu. W chwili publikowania niniejszej książki aktualne były następujące specyfikacje:
Servlet 2.4 (JSR #154) <http://jcp.org/en/jsr/detail?id=154>
JSP 2.0 (JSR #152) <http://jcp.org/en/jsr/detail?id=152>
JSTL 1.1 (JSR #52) <http://jcp.org/en/jsr/detail?id=52>
Po wyświetleniu każdej z wymienionych stron należy kliknąć łącze *Download Page* właściwe dla wersji ostatecznej (*Final Release*).
- Przetestuj Tomcata, uruchamiając skrypt *tomcat/bin/startup* (w systemach Linux, Unix i OSX nazwany *startup.sh*). W przeglądarce spróbuj wyświetlić stronę <http://localhost:8080/> — powinna się pojawić strona powitalna Tomcata.

Java Standard Edition 1.5

Tomcat 5

Egzamin obejmuje następujące specyfikacje:

- Servlets 2.4
- JSP 2.0
- JSTL 1.1

Kilka rzeczy, o których musisz wiedzieć

To książka do nauki, a nie encyklopedia. Celowo usunęliśmy wszystko, co mogłoby Ci przeszkadzać w *nauce*, niezależnie od tego, nad czym pracujesz w danym miejscu książki. Podczas pierwszej lektury książki należy zacząć od jej początku, gdyż kolejne rozdziały bazują na tym, co wiedziałeś i czego się dowiedziałeś wcześniej.

Używamy zmodyfikowanego i uproszczonego zapisu przypominającego UML.

Używamy prostych diagramów wzorowanych na UML-u.

Chociaż najprawdopodobniej znasz już język UML, egzamin nie obejmuje tego rodzaju zagadnień, stąd znajomość UML-a nie jest niezbędna czytania ze zrozumieniem także tej książki. Dzięki temu nie będziesz musiał rozpraszać uwagi na jednoczesne poznawanie serwletów, JSP, JSTL i języka UML.

Reżyser
<pre>getFilmy() getOskary() getStopienNaukowyKBacona()</pre>

Nie opisujemy wszystkich możliwych szczegółów podanych w specyfikacji.

Egzamin *jest* dosyć szczegółowy i my także. Jeśli jednak istnieje jakiś detal specyfikacji, który nie jest tematem pytań egzaminacyjnych, to my także nie będziemy go opisywać, chyba że ma on znaczenie dla większości osób tworzących komponenty. Zasób wiedzy koniecznej do tworzenia komponentów aplikacji internetowych (serwletów i stron JSP) pokrywa się w 85 procentach z informacjami niezbędnymi do zdania egzaminu. W książce zawarto szereg informacji, które nie należą do zagadnień egzaminacyjnych — za każdym razem staraliśmy się wyróżnić je w taki sposób, abyś wiedział, że ich zapamiętanie nie jest konieczne. To my tworzyliśmy *prawdziwy* egzamin, zatem doskonale wiemy, na czym powinieneś skoncentrować swoje wysiłki! Mogło się zdarzyć, że jakiś drobny szczegół, który będzie co prawda tematem jednego z pytań egzaminacyjnych, ale którego przyswojenie nie jest warte poświęcanego czasu, został przez nas pominięty, opisany bardzo pobieżnie bądź zawarty tylko w egzaminie próbnym.

Ćwiczenia SĄ obowiązkowe.

Ćwiczenia oraz wszelkie dodatkowe polecenia nie są jedynie dodatkami — stanowią integralną część podstawowej treści książki. Niektóre z nich zostały umieszczone po to, by pomóc w zapamiętaniu informacji; inne, by pomóc w zrozumieniu opisywanego materiału, a jeszcze inne — by pomóc Ci w praktycznym zastosowaniu zdobytej wiedzy. *Niczego nie pomijaj.*

Powtórzenia są celowe i ważne.

Jedną z cech, która wyróżnia serię książek *Head First*, jest to, iż *naprawdę bardzo, bardzo, bardzo zależy nam na tym, abyś wszystko zrozumiał i przyswoił*. Chcielibyśmy także, abyś zakończył lekturę tej książki, *pamiętając* zawarte w niej informacje. Autorzy większości książek informacyjnych i leksykonów nie stawiają sobie za cel Twojego przyswojenia i zapamiętania prezentowanej treści; w tej książce jest inaczej, stąd wiele pojęć będzie się pojawiało się kilka razy.

Przykładowe kody są możliwie zwięzłe.

Nasi Czytelnicy często opowiadają, jak frustrujące bywa przeglądanie 200 wierszy kodu w poszukiwaniu zaledwie dwóch wierszy ważnych z punktu widzenia omawianego zagadnienia. W większości przykładów zamieszczonych w tej książce dodatkowy kod, który nie jest bezpośrednio związany z omawianymi zagadnieniami, został w jak największym stopniu skrócony, aby fragmenty, których naprawdę musisz się nauczyć, były przejrzyste i proste. Nie należy zatem oczekiwać, że podawane przykłady będą solidne, ani nawet że będą kompletne. To *Twoim* zadaniem po zakończeniu lektury będzie ich uzupełnienie i dopracowanie. Przykłady zamieszczone w książce zostały opracowane wyłącznie w celach *dydaktycznych* i jako takie nie zawsze oferują pełną funkcjonalność. Niektóre przykłady udostępniono na witrynie internetowej www.wickedlysmart.com.

Słowo o egzaminie SCWCD (dla Java EE 1.5)

Zmodyfikowany egzamin SCWCD nazwano co prawda *Sun Certified Web Component Developer for the Java Platform, Enterprise Edition 5* (CX-310-083), jednak nie należy przeceniać znaczenia samego tytułu. Egzamin w nowej formie wciąż dotyczy zagadnień związanych z platformą Java EE 1.4 oraz specyfikacjami Servlet 2.4 i JSP 2.0.

Czy w pierwszej kolejności muszę zdać egzamin SCJP?

Tak. Egzaminy Web Component Developer, Business Component Developer, Mobile Application Developer, Web Services Developer oraz Developer wymagają wcześniejszego zdania egzaminu Sun Certified Java Programmer¹.

Z ilu pytań składa się egzamin?

Podczas egzaminu będziesz musiał odpowiedzieć na 69 pytań. Nie wszyscy otrzymują zestaw tych samych 69 pytań, istnieje wiele różnych zestawów. Niemniej jednak stopień trudności oraz zakres zagadnień tematycznych zawsze są takie same. Możesz oczekiwać, że na prawdziwym egzaminie każdemu z jego celów będzie odpowiadać jedno pytanie, a w niektórych przypadkach — *kilka* pytań.

Ile jest czasu na podanie odpowiedzi?

Masz trzy godziny (180 minut). Dla większości zdających czas nie stanowi problemu, ponieważ pytania egzaminacyjne nie są skomplikowane, długie czy podchwytliwe. Zdecydowana większość pytań jest bardzo krótka i daje zdającemu możliwość wyboru jednej lub wielu spośród kilku podanych odpowiedzi — albo znasz tę odpowiedź, albo nie.

Jakich pytań należy oczekiwać?

Niemal identycznych jak te na naszym egzaminie próbnym; z jedną zasadniczą różnicą — na *prawdziwym* egzaminie z góry wiadomo, ile odpowiedzi jest poprawnych; my tego rodzaju informacji nie podajemy. Na egzaminie zdarzają się wygodne pytania typu „przeciągnij i upuść”, których z natury rzeczy nie można wykorzystać w książce. Z drugiej strony, tego rodzaju pytania są jedynie interaktywnym sposobem na łączenie odpowiedzi.

Na ile pytań muszę odpowiedzieć poprawnie?

Aby zdać egzamin, musisz poprawnie odpowiedzieć na 49 pytań (70 procent wszystkich pytań). Kiedy odpowiesz na wszystkie pytania, umieść wskaźnik myszy nad przyciskiem kończącym egzamin, aż zbierzesz w sobie odwagę, by go kliknąć. Gdy to zrobisz, w ciągu (około) sześciu nanosekund dowiesz się, czy zdałeś (na pewno Ci się *uda*).

Dlaczego egzaminy próbne zamieszczone w tej książce nie podają, ile odpowiedzi trzeba zaznaczyć?

Chcemy, aby nasz egzamin był nieco trudniejszy od prawdziwego, żeby dać Ci jak najbardziej realistyczne wyobrażenie tego, co Cię czeka. Dużo osób uzyskuje na egzaminach próbnych lepsze wyniki, gdyż podchodzi do nich więcej niż jeden raz. Nie chcemy, abyś wyrobił sobie błędne zdanie na temat stopnia przygotowania do egzaminu. Czytelnicy sygnalizowali nam, iż na prawdziwym egzaminie uzyskiwali wyniki bardzo zbliżone do tych, jakie uzyskali na naszym egzaminie próbnym.

¹ W nomenklaturze przyjętej przez polski oddział firmy Sun egzaminy te noszą nazwy, odpowiednio: autoryzowany developer komponentów internetowych w zakresie platformy J2EE, autoryzowany developer komponentów biznesowych w zakresie platformy J2EE, autoryzowany developer w zakresie języka Java oraz autoryzowany programista w zakresie platformy Java. Polskojęzyczna witryna firmy Sun nie podaje odpowiedników egzaminów: Mobile Application Developer oraz Web Services Developer.

Co otrzymam po zdaniu egzaminu?

Opuszczając centrum egzaminacyjne, nie zapomnij wziąć swojego świadectwa. Zawiera ono uzyskany wynik w każdym z podstawowych działów tematycznych oraz informację o tym, czy egzamin został zdany, czy też nie. *Zachowaj ten dokument!* To Twój pierwszy dowód uzyskania certyfikatu. Po kilku tygodniach otrzymasz z centrum edukacyjnego firmy Sun (ang. *Sun Educational Center*) niewielką przesyłkę zawierającą *prawdziwy*, wydrukowany certyfikat, list gratulacyjny oraz prześliczną odznakę z napisem Sun Certified Web Component Developer. Użyta czcionka jest tak mała, iż z powodzeniem możesz utrzymywać, że jesteś certyfikowany w czymkolwiek byś zechciał, a i tak nikt nie będzie w stanie zauważyć różnicy. W przesyłce nie znajdziesz jednak butelki z trunkiem, którym zapewne zechcesz uczcić zdanie egzaminu.

Ile kosztuje egzamin i gdzie można się zarejestrować?

Egzamin certyfikujący kosztuje 750 PLN. Właśnie dlatego powstała niniejsza książka... abyś mógł zdać go już za pierwszym razem. Dokonanie rezerwacji wymaga skorzystania z Serwisu Edukacyjnego Sun Microsystems i podania numeru swojej karty kredytowej, by w zamian otrzymać numer *vouchera* uprawniającego do przystąpienia do egzaminu w najbliższym centrum egzaminacyjnym Prometric Testing Center.

Wszelkie informacje na temat rejestracji i miejsc, gdzie są przeprowadzane egzaminy, można znaleźć na stronie internetowej firmy Sun pod adresem <https://www.suntrainingcatalogue.com/eduserv/client/learningPath.do?p=/training/certification/index.html>.

Jak wygląda oprogramowanie używane do przeprowadzenia egzaminu?

Jest wyjątkowo proste — na ekranie jest wyświetlane pytanie, a Ty musisz tylko wskazać odpowiedź. Jeśli nie chcesz odpowiadać na dane pytanie, możesz je pominąć, by wrócić do niego później. Jeśli odpowiedziałeś na jakieś pytanie, ale nie jesteś pewny udzielonej odpowiedzi, możesz to pytanie odpowiednio „oznaczyć” i wrócić do niego później, jeśli oczywiście starczy Ci na to czasu. Kiedy skończysz, na ekranie zostaną wyświetlone wszystkie pytania, na które nie udzieliłeś odpowiedzi lub które zostały oznaczone, dzięki czemu będziesz mógł do nich powrócić.

Na samym początku egzaminu zostanie przeprowadzone krótkie wprowadzenie dotyczące sposobów obsługi oprogramowania, podczas którego będziesz mógł rozwiązać krótki test próbny (który jednak nie będzie związany z serwetami). Czas poświęcony na ten próbny test nie będzie uwzględniany jako czas egzaminu SCWCD. Odmierzanie czasu egzaminu rozpocznie się dopiero wtedy, gdy zakończysz pracę z przykładowym egzaminem i zasygnalizujesz swoją gotowość do przystąpienia do właściwego testu.

Gdzie można znaleźć grupę dyskusyjną związaną z tym egzaminem i jak długo trzeba się do niego przygotowywać?

Tak się składa, że najlepsza internetowa grupa dyskusyjna poświęcona temu egzaminowi jest prowadzona przez autorów niniejszej książki. (O rany, a co to szkodzi?). Zajrzyj na witrynę www.javaranch.com, do jej działu *Big Moose Saloon*. Nie sposób przegapić grup dyskusyjnych dotyczących certyfikacji. Zawsze znajdzie się na nich *ktoś*, kto udzieli odpowiedzi na Twoje pytania, może nawet będziemy to *my*. Użytkownicy witryny JavaRanch należą do najbardziej przyjaznych społeczności w całym internecie, zatem będziesz tam mile widziany niezależnie od poziomu znajomości języka Java. Możesz liczyć na naszą pomoc, nawet jeśli dopiero przygotowujesz się do egzaminu SCJP (autoryzowany programista platformy Java), to także możemy Ci pomóc.

Czas przygotowań do egzaminu zależy w dużej mierze od posiadanych doświadczeń w dziedzinie korzystania z technologii związanych z serwetami i JSP. Jeśli dopiero *zaczynasz* je poznawać, przygotowania mogą Ci zająć od sześciu do dwunastu tygodni (w zależności od czasu poświęcanego codziennej nauce). Najbardziej doświadczeni programiści potrafią opanować niezbędną wiedzę w zaledwie trzy tygodnie.

Betatesterzy i korektorzy techniczni



Dave Wood



Joe Konior

Bear
Bibeault



Neeraj Singhal



Dwa nowe
siwe włosy,
do których
przyczyniła
się ta
książka.



Johannes deJong

Andrew Monkhouse



Jason Menard



Jef Cumps



Dirk Schreckmann

Ulf Dittmer



Oliver Roell



Theodore Casser



Preetish Madalia



Collins Tchoumba



Nie ma go na żadnym
zdjęciu (choć jest
równie uroczy jak
pozostali):
Amit Londhe

którym należy się uznanie

Inne osoby, które ~~należy~~ winić

W wydawnictwie O'Reilly:

Bardzo dziękujemy **Mike'owi Loukidesowi** z wydawnictwa O'Reilly za to, że zapoczątkował pomysł i potrafił przekształcić go w całą serię książek *Head First*. To wspaniałe współpracować z redaktorem będącym Prawdziwym Programistą Javy. Dziękujemy także „sile sprawczej” stojącej za całą serią książek *Head First* — **Timowi O'Reilly**. Na szczęście dla nas, Tim zawsze myśli o przyszłości i doskonale się bawi, realizując swoje destrukcyjne koncepcje. Dziękujemy także „matce” całej serii książek *Head First* — **Kyle Hart** — za wskazanie sposobu wprowadzenia tych publikacji do świata książek informatycznych.

Naszym nieustraszonym recenzentom:

No dobrze, napisanie tej książki rzeczywiście zajęło nam nieco więcej czasu, niż początkowo planowaliśmy. Jednak bez pomocy **Johannesa deJonga** — kierownika recenzentów witryny JavaRanch — trwałoby to *jeszcze* dłużej. Johanessie — jesteś naszym bohaterem. Chcielibyśmy wyrazić specjalne podziękowania dla **Joego Koniora**, którego opinie na temat każdego z rozdziałów były niemal równie obszerne co *treść* samego rozdziału. Jesteśmy też wdzięczni **Phillipowi Maquetowi** za ogromny wysiłek i wiedzę fachową (a także poczucie humoru). Cała trójka autorów niniejszej książki pokochała go do tego stopnia, że byłaby skłonna stanąć z nim na ślubnym kobiercu... choć byłoby to raczej dziwaczne. Jesteśmy także bardzo wdzięczni **Andrew Monkousowi** zarówno za opinie techniczne, *jak również* za subtelne sugestie dotyczące przekładu niniejszej książki z angielskiego na australijski angielski. **Jefie Cumps** — przerobiona przez Ciebie piosenka „setHeader” była rewelacyjna (choć może zbyt *emocjonalna*), a Twoje komentarze techniczne były *niezwykle* przydatne.

Dave Wood zwracał nam uwagę na *wszystko*, a szczególnie upodobał sobie wskazywanie czegoś na pierwszych stronach książki i twierdzenie: „to raczej nie jest w stylu serii *Head First*”. Zawsze mogliśmy też liczyć na cenne uwagi moderatorów witryny JavaRanch: **Jasona Menarda**, **Dirka „Rybiej Twarzy” Schreckmanna**, **Roba Rossa**, **Ernesta Friedman-Hilla** oraz **Thomasa Paula**. Na koniec chcielibyśmy wyrazić ogromną wdzięczność głównemu pastuchowi witryny *javaranch.com* — **Paulowi Wheatonowi**.

Za pytania do egzaminu próbnego

Jeśli bez końca głowisz się nad jakiś szczególnie zakręconym lub zwodniczym pytaniem testowym poświęconym stronom JSP, nie zrzucaj winy na nas — skieruj swoją złość na Marca Peabody'ego! Dziękujemy Marcowi za wzięcie na siebie odpowiedzialności przed wszystkimi kandydatami do certyfikatu SCWCD. Marc poświęca znaczną część swojego wolnego czasu na moderowanie dyskusji na witrynie JavaRanch, gdzie słynie z konstruowania najdziwniejszych, zupełnie niespodziewanych rozwiązań na podstawie niewinnych technologii Java EE.



Marc Peabody



Kolejne osoby*

Podziękowania od Bryana Bashama

Mógłbym zacząć od Mamy, ale to takie oklepane... Moja znajomość tworzenia aplikacji internetowych w Javie bazuje na kilku aplikacjach średniej wielkości, które napisałem, jednak baza ta została udoskonalona i wyostzona poprzez lata debat na temat Javy z instruktorami tego języka zatrudnionymi w firmie Sun. W szczególności chciałbym podziękować: Steve'owi Sheltingowi, Victorowi Petersowi, Lisie Morris, Jean Tordelli, Michaelowi Juddowi, Evanowi Troyka oraz Keithowi Ratliffowi. Wiele osób „rzeźbiło” moją wiedzę, jednak właśnie wymienioną siódemkę można by porównać do „dłut”, które ukształtowały ją w największym stopniu.

Jak to zazwyczaj bywa w przypadku pisania książek, ostatnie trzy miesiące były dosyć trudne. Chciałbym podziękować mojej narzeczonej — Kathy Collinie — za okazaną mi cierpliwość. Chciałbym także podziękować naszym kotom — Karmie i Kiwi — za długie nocne siedzenie na kolanach i zabawy z klawiaturą.

Na koniec chciałem wyrazić najważniejsze podziękowania dla Kathy i Berta za pomysł wspólnego podjęcia tego wyzwania. Kathy Sierra jest osobą absolutnie niepowtarzalną. Jej wiedza na temat metapoznania i projektów dydaktycznych może się równać wyłącznie z jej kreatywnością tryskającą wprost z jej książek *Head First*. Już od pięciu lat zajmujemy się edukacją i niemal całą swoją wiedzę w tej dziedzinie zawdzięczam właśnie Kathy... Och, nie martwcie się o moją Mamę, w kolejnej książce serii *Head First* doczeka się długiej dedykacji specjalnie dla niej. Kocham Cię, Mamo!

Od Kathy i Berta

Bryan, to było zbyt słodkie. (Nie należy przez to rozumieć, że Kathy nie lubi takiego podlizywania się). Z pewnością zgadzamy się jednak w ocenie Twojej narzeczonej. I nie chodzi o to, jak za Tobą *tęskniła* całe lato, grając na komputerze, podczas gdy *my* pracowaliśmy jak przysłowiowe woły. Z drugiej strony, sprawiłeś, że wspólne pisanie niniejszej książki było wspaniałym doświadczeniem — jesteś najlepszym² współautorem, jakiego kiedykolwiek mieliśmy! Czasem przeraża nas Twój *nieustający* spokój i radość z życia.

Wszyscy doceniamy zapracowany zespół odpowiedzialny za certyfikaty firmy Sun, a zwłaszcza jego szefową — Evelyn Cartagenę. Dziękujemy także wszystkim, którzy przyczynili się do opracowania dokumentów JSR (Java Specification Request) dla specyfikacji serwletów i JSP.

* Duża liczba zamieszczonych podziękowań wynika z faktu, iż testujemy prawdziwość teorii, według której każda osoba wymieniona w podziękowaniach kupuje przynajmniej jeden egzemplarz książki, a może i więcej — dla rodziny, znajomych itp. Jeśli więc chciałbyś zostać wymieniony w podziękowaniach w naszej *kolejnej* książce i jeśli masz dużą rodzinę, po prostu do nas napisz.

² Wyjaśnienie: Bryan jest *jedynym* współautorem, jakiego kiedykolwiek mieliśmy, co jednak w najmniejszym stopniu nie umniejsza wagi tych podziękowań.

1. Wprowadzenie i przegląd najważniejszych zagadnień

Do czego służą serwlety i strony JSP?

Ha! Znam CGI.
Moja witryna będzie
rządzić światem.

Głupcze! Powinieneś
używać serwletów i stron JSP.
Jeśli nadal będziesz pisał skrypty
w Perlu, zniszczę Ciebie
i Twoją witrynę!



Aplikacje internetowe są cudowne. Aplikacje z interfejsem GUI mogą co prawda wykorzystywać egzotyczne formanty Swing, warto się jednak zastanowić, ile znamy aplikacji GUI, które są stosowane przez miliony użytkowników na całym świecie. Jako programista aplikacji internetowych możesz się uwolnić od wiecznych problemów z wdrażaniem samodzielnych programów i udostępniać swoje rozwiązania wszystkim użytkownikom, którzy dysponują przeglądarką internetową. Konstruowanie naprawdę potężnych aplikacji tego typu wymaga jednak stosowania języka programowania Java. Będziesz potrzebował serwletów oraz stron JSP — stare, statyczne strony HTML były być może interesujące w roku 1999, ale na pewno nie zaskoczą dzisiejszych użytkowników, którzy oczekują od współczesnych witryn dynamiki, interaktywności oraz elastyczności. Z tej książki dowiesz się, jak można przekształcić *witrynę* internetową w prawdziwą *aplikację* internetową.



Przegląd technologii serwletów i stron JSP

1.1. Dla każdej z metod przesyłania żądań protokołu HTTP (takich jak GET, POST, HEAD itp.):

- * Opisz korzyści wynikające ze stosowania danej metody.
- * Opisz funkcjonalność danej metody.
- * Wymień zdarzenia, które mogą spowodować, że klient (zazwyczaj przeglądarka WWW) użyje danej metody.

Poniższe zadanie jest częścią celu 1.1, mimo że nie zostało omówione w tym rozdziale:

- * Zidentyfikuj metodę `HttpServlet`, która odpowiada danej metodzie protokołu HTTP.

Uwagi wyjaśniające:

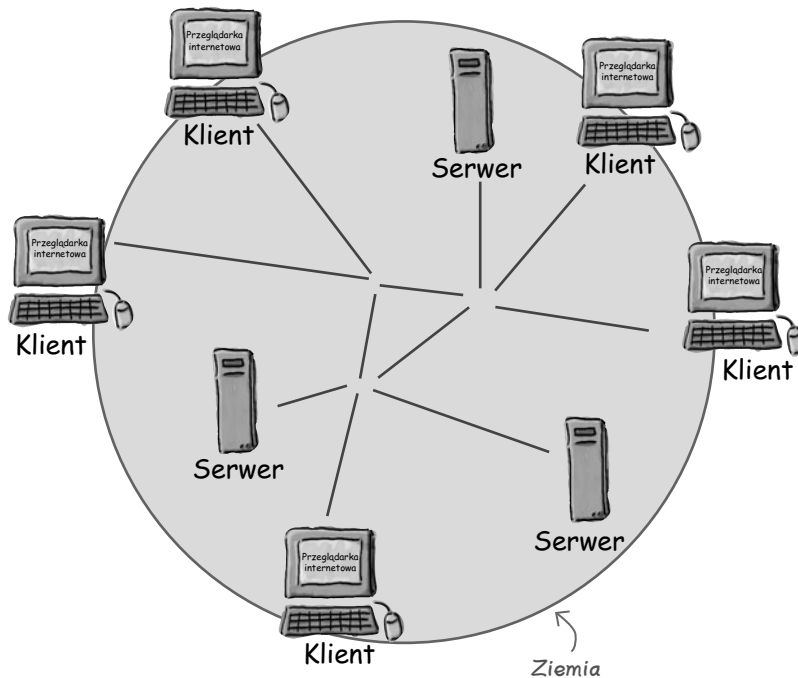
Cele zdefiniowane w tym podrozdziale zastaną w pełni zrealizowane w innym rozdziale, a zatem niniejszy rozdział należy traktować jak wstępne omówienie pewnych podstawowych zagadnień, które będą nam potrzebne w przyszłości. Innymi słowy, nie powinienes się przejmować, jeśli po przeczytaniu tego rozdziału nie będziesz pamiętał szczegółowych celów egzaminu — ich prezentacja w tym miejscu ma jedynie charakter omówienia zakresu tematycznego tego rozdziału. Jeśli znasz już wymienione zagadnienia, możesz od razu przejść do rozdziału 2. Na końcu tego rozdziału nie będziemy omawiali żadnych przykładowych pytań egzaminacyjnych, ponieważ ich analiza będzie możliwa dopiero po opanowaniu bardziej szczegółowego materiału zawartego w kolejnych rozdziałach.

Wszyscy chcą mieć swoje witryny internetowe

Masz już pomysł, jak skutecznie dobić swoją konkurencję w internecie. Aby ostatecznie rozstrzygnąć rywalizację na swoją korzyść, będziesz potrzebował elastycznej i skalowalnej architektury — będziesz potrzebował serwetów i stron JSP.

Zanim przystąpimy do budowania naszej pierwszej aplikacji internetowej, musimy spojrzeć na sieć WWW z wysokości około dwudziestu tysięcy metrów. Tym, co szczególnie nas interesuje w tym rozdziale, jest sposób, w jaki klienci WWW komunikują się z serwerami WWW.

Najprawdopodobniej materiał zawarty na kolejnych kilku stronach będzie jedynie przypomnieniem znanych Ci faktów, szczególnie jeśli już jesteś programistą aplikacji internetowych. Z drugiej strony, być może warto się zapoznać z naszym wyjaśnieniem terminologii, którą będziemy stosować niemal we wszystkich dalszych rozdziałach tej książki.



Sieć WWW składa się z niewyobrażalnej liczby klientów (wykorzystujących takie przeglądarki jak Mozilla lub Safari) oraz serwerów (z pracującymi aplikacjami serwerów WWW, np. serwerem Apache) połączonych ze sobą za pomocą przewodowych i bezprzewodowych sieci komputerowych. Naszym celem jest budowa aplikacji internetowej, z której będą mogli korzystać klienci na całym świecie. I oczywiście osiągnięcie nieprzyzwoitego bogactwa.

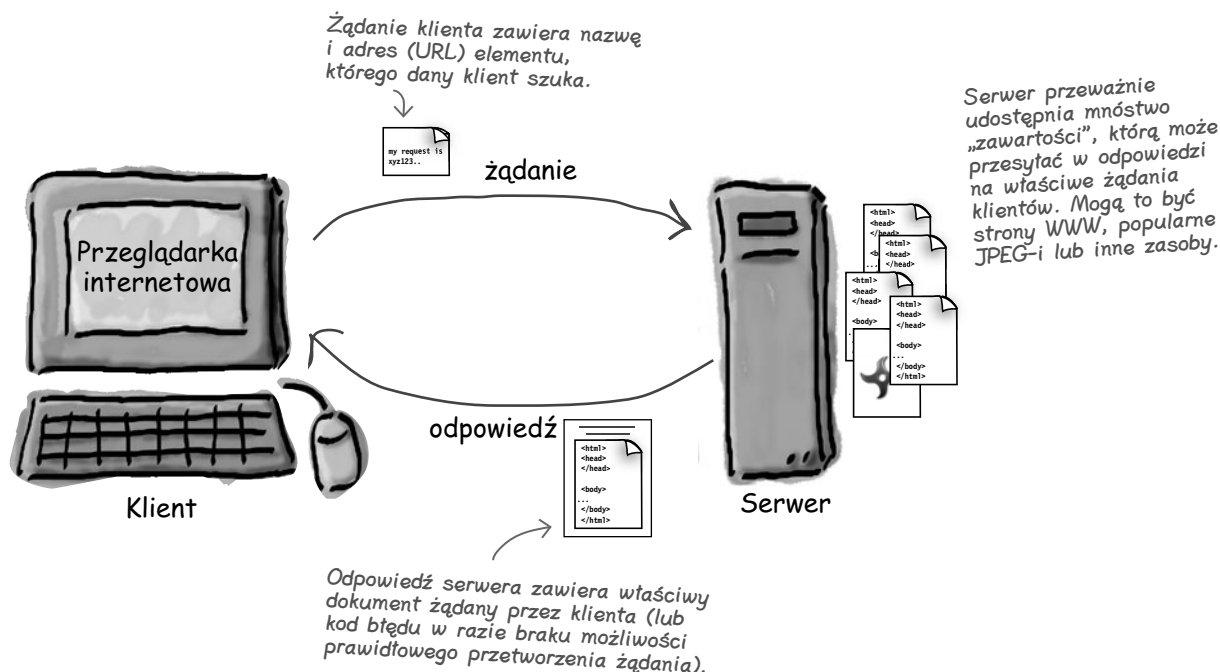
Co tak naprawdę robi Twój serwer WWW?

Serwer WWW otrzymuje żądanie klienta i coś temu klientowi odsyła.

Przeglądarka internetowa umożliwia użytkownikowi tworzenie żądań odnośnie zasobów. Serwer WWW otrzymuje to żądanie, znajduje odpowiednie zasoby i odsyła do użytkownika właściwą odpowiedź. Niekiedy żądanym zasobem jest *strona HTML*, czasami jest nim *obraz*, *plik dźwiękowy* lub nawet *dokument PDF*. Nie ma to jednak większego znaczenia — klient prosi o określony element (zasób) i serwer w miarę możliwości ten element odsyła.

Jeśli interesujących elementów nie ma tam, gdzie ich szukamy, lub jeśli serwer szuka zasobów w niewłaściwy sposób, w oknie przeglądarki zostanie wyświetlona wiadomość "Błąd 404: Strony nie znaleziono" — znany wszystkim komunikat oznaczający, że serwer nie może znaleźć strony żądanej przez klienta.

Mówiąc o „serwerze”, mamy na myśli albo fizyczny komputer (sprzęt), albo aplikację serwera WWW (oprogramowanie). Jeśli gdziekolwiek w tej książce rozróżnienie obu znaczeń słowa „serwer” będzie miało jakieś znaczenie, jasno określimy, czy mamy na myśli sprzęt, czy oprogramowanie.



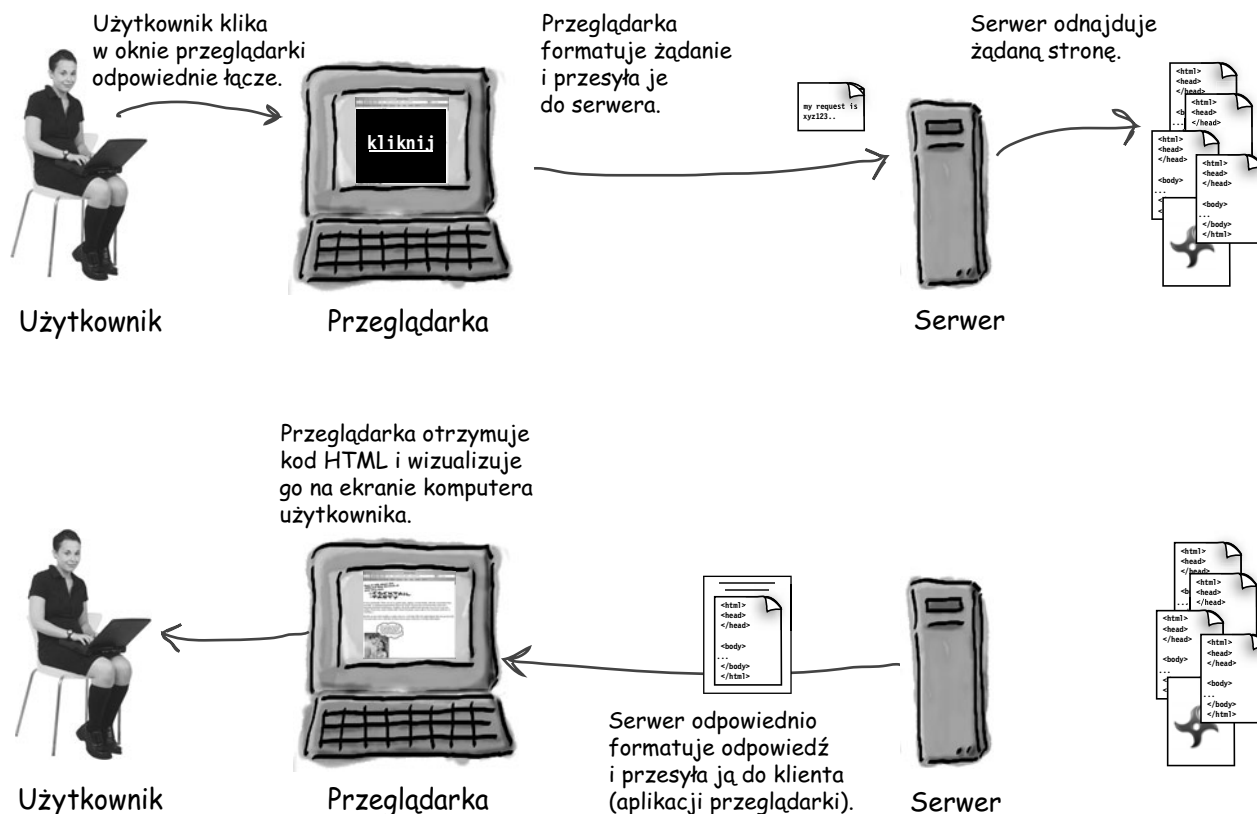
Co tak naprawdę robi klient WWW?

Klient WWW z jednej strony umożliwia użytkownikowi definiowanie zadań kierowanych do serwera, z drugiej strony prezentuje temu samemu użytkownikowi wyniki otrzymane dla tych zadań.

Warto jednak pamiętać, że kiedy mówimy o *kliencie*, mamy zwykle na myśli zarówno człowieka występującego w roli użytkownika systemu, jak i *aplikację* przeglądarki.

Przeglądarka internetowa to program (jak choćby Netscape lub Mozilla), który „wie”, jak komunikować się z serwerem. Innym ważnym zadaniem przeglądarki jest interpretowanie kodu HTML i *wizualizowanie* strony internetowej dla użytkownika.

Kiedy więc od tej chwili będziemy używali terminu *klient*, przeważnie nie będzie miało znaczenia, czy mamy na myśli użytkownika, czy aplikację przeglądarki. Innymi słowy, *klient* jest *aplikacją* przeglądarki realizującą zadania zdefiniowane przez użytkownika.



Klienci i serwery dobrze znają język HTML i protokół HTTP

HTML

Generowana przez serwer odpowiedź na otrzymane żądanie zwykle ma postać pewnego rodzaju treści dla przeglądarki internetowej, którą ta przeglądarka będzie mogła wyświetlić przed użytkownikiem. Serwery WWW często wysyłają do przeglądarek zbiory instrukcji zapisanych w języku HTML (od ang. *HyperText Markup Language*). Język HTML określa sposób, w jaki przeglądarka ma zaprezentować treść odpowiedzi na ekranie komputera.

Wszystkie przeglądarki internetowe doskonale wiedzą, co należy robić z otrzymywanym kodem języka HTML, chociaż często okazuje się, że *starsze* przeglądarki mogą nie rozumieć pewnych fragmentów stron napisanych z wykorzystaniem *nowszych* wersji języka HTML.

HTTP

Znakomita większość konwersacji utrzymywanych pomiędzy klientami a serwerami WWW opiera się na popularnym protokole HTTP, który umożliwia wymianę prostych komunikatów żądań i odpowiedzi. Klient wysyła do serwera WWW żądanie protokołu HTTP, natomiast serwer WWW odsyła do klienta odpowiedź protokołu HTTP. *Jeśli jesteś serwerem WWW, musisz używać protokołu HTTP.*

Kiedy serwer WWW wysyła do klienta stronę HTML, w rzeczywistości wysyła odpowiedni komunikat protokołu HTTP (szczegóły tego typu działań zostaną wyjaśnione w dalszej części tego rozdziału).

(Do Twojej wiadomości: HTTP jest akronimem angielskiego określenia *HyperText Transport Protocol*).



HTML mówi przeglądarce, jak należy wyświetlać na ekranie użytkownika zawartość strony.

HTTP jest protokołem wykorzystywanym do komunikacji klientów i serwerów sieci Web.

Serwer wykorzystuje protokół HTTP do odsyłania klientom kodu HTML.

Dwuminurowy kurs języka HTML

Kiedy tworzymy stronę internetową, w języku HTML opisujemy, jak dana strona ma wyglądać i jak powinna się zachowywać.

Język HTML składa się z dziesiątek **znaczników** i setek **atrybutów** tych znaczników. Celem języka HTML jest wzbogacenie dokumentu tekstowego o znaczniki określające sposób formatowania tego tekstu w oknie przeglądarki internetowej. Poniżej przedstawiono znaczniki, które będziemy wykorzystywali w kilku kolejnych rozdziałach tej książki.

Jeśli szukasz pełniejszego przeglądu znaczników i atrybutów języka HTML, zalecamy zapoznanie się z treścią książki *HTML i XHTML*.
Przewodnik encyklopedyczny.

Znacznik

Opis

<!-- -->

tu umieszczamy *komentarze*

<a>

kotwica — wykorzystywana zazwyczaj do umieszczania hiperłącza

<align>

wyrównanie zawartości do lewej lub prawej strony,
do środka lub justowanie

<body>

definiuje zakres *ciała* dokumentu

podział wiersza

<center>

wyśrodkowanie zawartości

<form>

definiuje *formularz* (który zwykle zawiera pola wejściowe dla danych wpisywanych przez użytkownika)

<h1>

pierwszy poziom *nagłówka*

<head>

definiuje zakres *nagłówka* dokumentu

<html>

definiuje zakres samego *dokumentu* HTML

<input type>

definiuje *formant wejściowy* dla formularza HTML

<p>

nowy akapit (*paragraf*)

<title>

tytuł dokumentu HTML

(Z technicznego punktu widzenia znaczniki <center> i <align> w wersji 4.0 standardu HTML zostały uznane za przestarzałe; my jednak używamy tych znaczników w niektórych z prezentowanych przykładów, ponieważ kod HTML opracowany z ich wykorzystaniem jest łatwiejszy do analizy niż odpowiednie rozwiązania alternatywne, a celem tej książki nie jest przecież nauka HTML-a.)

Co piszemy... (kod HTML)

Wyobraź sobie, że tworzysz stronę logowania. Kod takiej prostej strony HTML może mieć następującą postać:

```
<html>
<!-- Trochę prostego kodu HTML -->
<head>
  <title>Strona logowania</title>
</head>
<body>
  <h1 align="center">Strona logowania Skywalker</h1>

  <p align="right">

  </p>

  <form action="dane2">
    Nazwa: <input type="text" name="param1"/><br/>
    Hasło: <input type="text" name="param2"/><br/><br/><br/>

    <center>
      <input type="SUBMIT"/>
    </center>
  </form>

</body>
</html>
```

Komentarz HTML

Znacznik został zagnieżdżony w ramach znacznika akapitu <p>, aby nasz obraz znajdował się mniej więcej tam, gdzie go potrzebujemy. (Pamiętaj, że stosowanie atrybutu <align> nie jest obecnie zalecane — używamy go tutaj wyłącznie z uwagi na jego prostotę).

Serwlet do którego należy wysłać żądanie.

*Znaczniki
 powodują przejście do nowego wiersza.*

Formularze HTML omówimy bardziej szczegółowo w dalszej części tego rozdziału; krótko mówiąc, przeglądarka może za pośrednictwem formularza pobierać od użytkownika dane wejściowe i przekazywać je do serwera WWW.

Przycisk Wyślij zapytanie tego formularza HTML.



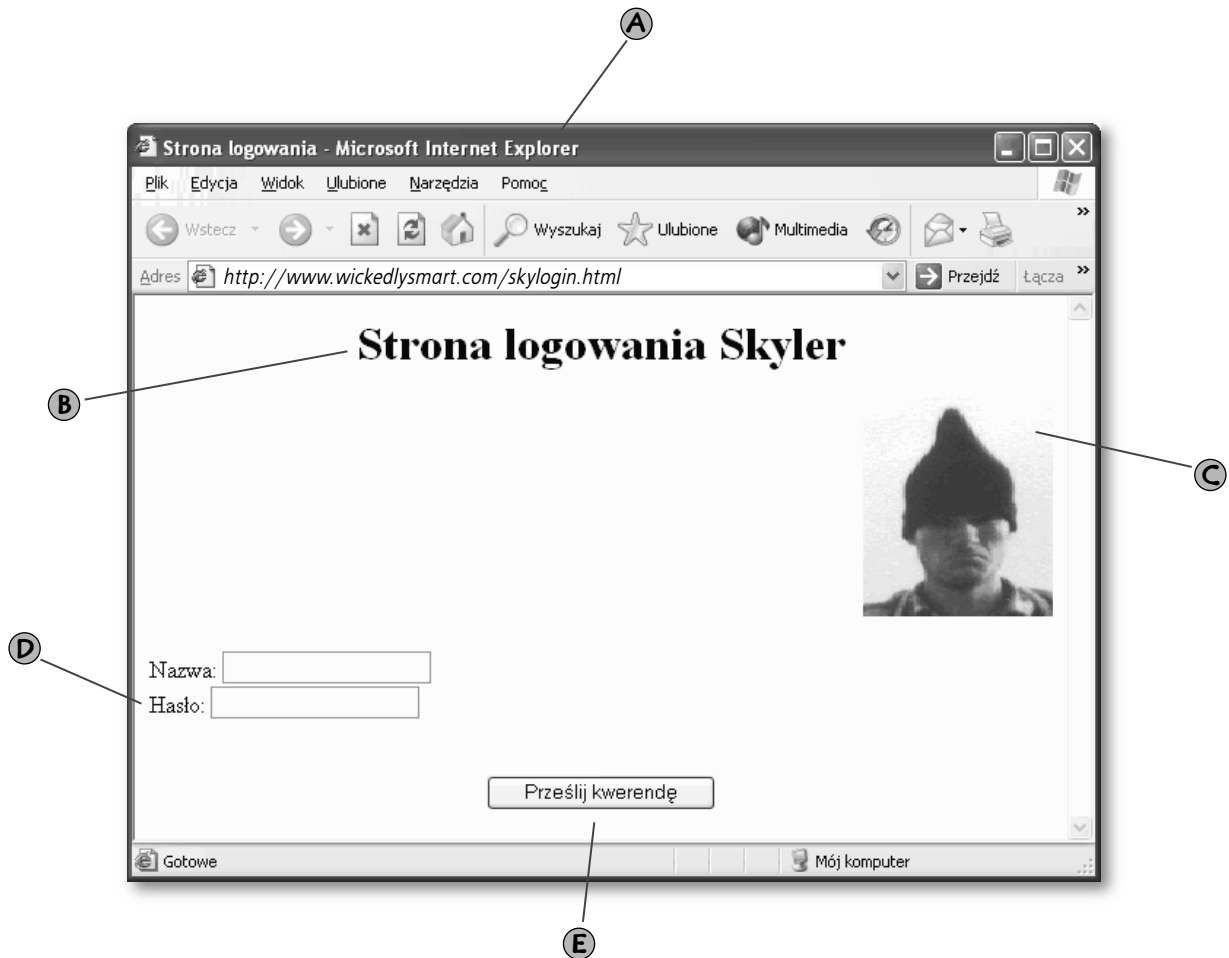
Relax

Musisz dysponować jedynie podstawową wiedzą na temat języka HTML

Język HTML przewija się w rozmaitych pytaniach egzaminacyjnych. Nie oznacza to jednak, że na tym egzaminie ktokolwiek *testuje* Twoją wiedzę o HTML-u. Z drugiej strony, ponieważ kod tego języka będzie występował w kontekście znacznej części pytań, warto opanować przynajmniej podstawy umożliwiające prawidłową interpretację prostych fragmentów kodu języka HTML.

Co tworzy przeglądarka internetowa...

Przeglądarka odczytuje kod HTML, tworzy stronę internetową i wizualizuje przetworzony kod na ekranie użytkownika.

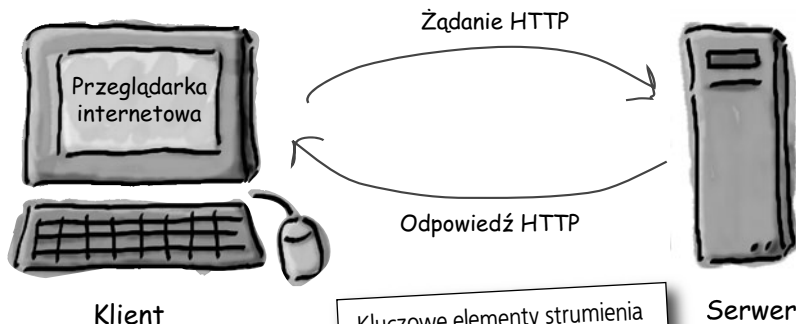


Czym jest protokół HTTP?

Protokół HTTP działa na szczycie stosu TCP/IP. Jeśli nie znasz się na tych wszystkich protokołach sieciowych, proponujemy Ci przejście błyskawicznego kursu — protokół TCP odpowiada za zapewnianie poprawności przesyłania pliku pomiędzy węzłami sieci komputerowej (protokół musi gwarantować kompletność pliku w węźle docelowym także w sytuacjach, gdy podczas transmisji plik ten został podzielony na mniejsze fragmenty). Działający na niższym poziomie protokół IP odpowiada za transmisję i znajdowanie tras dla fragmentów danych (pakietów) podczas ich przesyłania z komputera źródłowego do komputera docelowego. Kolejnym wykorzystywanym protokołem sieciowym jest HTTP, który zawiera co prawda rozwiązania właściwe tylko dla przesyłania stron WWW, ale jednocześnie bazuje na uniwersalnym protokole TCP/IP (odpowiedzialnym za dostarczanie do miejsc przeznaczenia kompletnych żądań i odpowiedzi). Struktura konwersacji HTTP jest prostą sekwencją **żądań-odpowiedzi**, gdzie przeglądarka generuje *żądania*, a serwer generuje *odpowiedzi*.

Kluczowe elementy strumienia **żądania**:

- metoda HTTP (akcja, którą należy podjąć),
- docelowa strona (adres URL),
- parametry formularza (np. argumenty danej metody).



Kluczowe elementy strumienia **odpowiedzi**:

- kod stanu (określający, czy dane żądanie zostało pomyślnie przetworzone),
- typ zawartości (tekst, obraz, HTML itp.),
- zawartość (właściwy kod HTML, obraz itp.).



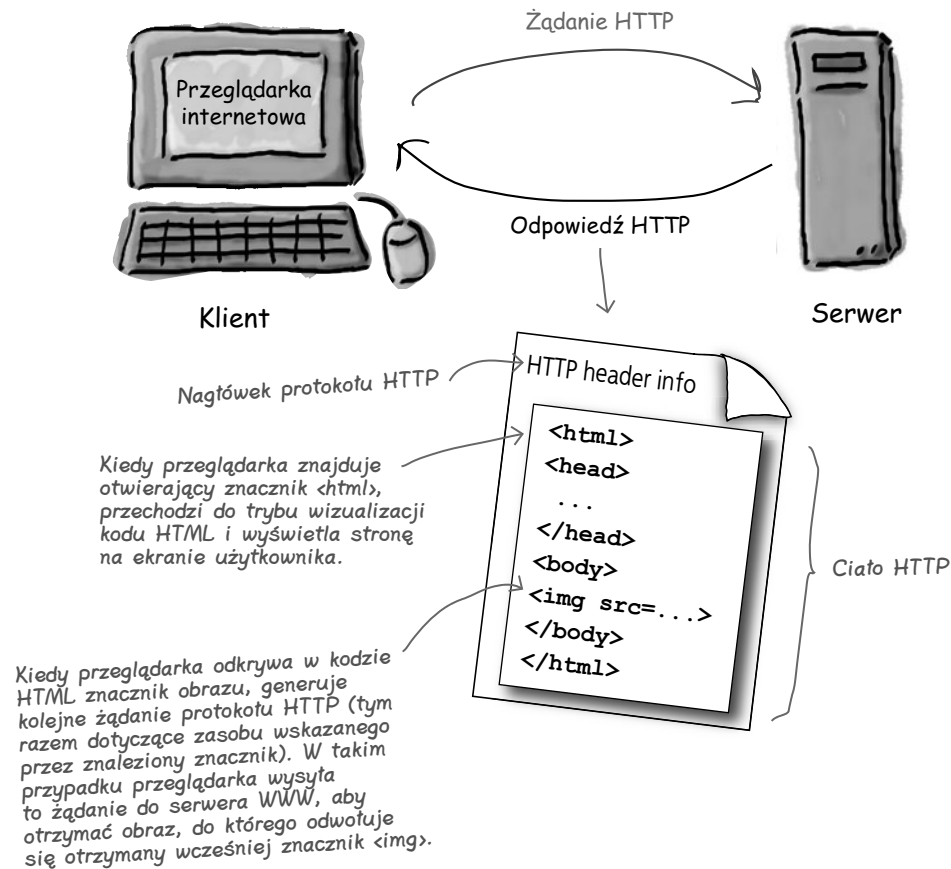
Relax

Specyfikacji protokołu HTTP nie musisz się uczyć na pamięć

Protokół HTTP jest standardem grupy roboczej IETF (od ang. Internet Engineering Task Force) oznaczonym numerem RFC 2616 (jeśli nie chcesz, oczywiście nie musisz tego pamiętać — na szczęście na egzaminie nikt nie będzie od Ciebie oczekiwał tego typu informacji). Przykładem serwera WWW przetwarzającego żądania protokołu HTTP jest Apache. Mozilla jest natomiast przykładem przeglądarki internetowej, która udostępnia użytkownikowi środki do tworzenia żądań HTTP i przeglądania dokumentów zwracanych przez serwer.

HTML jest częścią odpowiedzi protokołu HTTP

Odpowiedź protokołu HTTP może zawierać kod języka HTML. Protokół HTTP poprzedza właściwą zawartość odpowiedzi (czyli *to*, co serwer odsyła klientowi) informacjami nagłówka. Przeglądarka stron HTML wykorzystuje informacje zawarte w nagłówku jako element pomocniczy podczas przetwarzania otrzymanej strony internetowej. Właściwy kod HTML można więc traktować jak dane wklejone do wnętrza odpowiedzi protokołu HTTP.



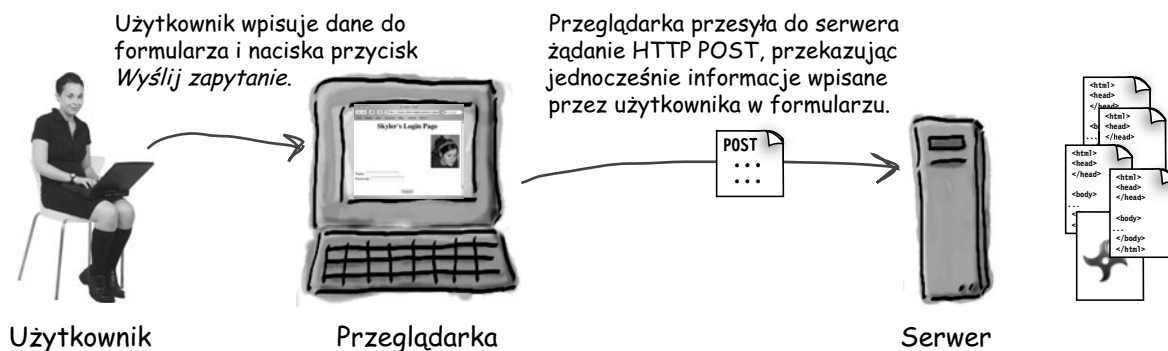
Skoro tak wygląda odpowiedź HTTP, co znajduje się w ządaniu tego protokołu?

Pierwszą rzeczą, z którą mamy do czynienia podczas analizy żądań przeglądarki, jest nazwa *metody* HTTP. Nie są to co prawda metody *Javy*, ale koncepcja ich stosowania jest podobna. Nazwa metody określa na potrzeby serwera rodzaj wygenerowanego żądania oraz sposób formatowania pozostałej części komunikatu. Protokół HTTP obsługuje wiele różnych metod, ale tymi, z których będziemy korzystać zdecydowanie najczęściej, są *GET* i *POST*.

GET



POST



GET jest prostym żądaniem, POST może zawierać dane użytkownika

GET jest najprostszą metodą protokołu HTTP — jej głównym zadaniem jest żądanie od serwera *dostarczenia* określonych zasobów i odesłania ich przeglądarce internetowej. Żądanym zasobem może być strona HTML, obraz w formacie JPEG, dokument PDF itp. To bez znaczenia. Celem metody GET jest zawsze *otrzymanie* czegoś z serwera WWW.

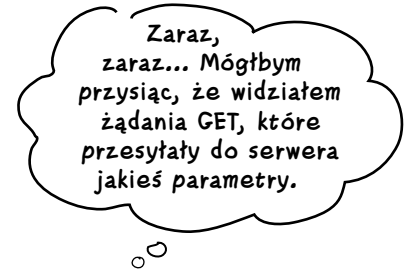
Metoda POST daje nieco większe możliwości. Żądania POST pod wieloma względami przypominają żądania GET, ale oferują coś więcej. Za pomocą żądań POST możesz *żądać* od serwera odesłania jakiegoś zasobu i jednocześnie *przesyłać* na serwer dane z formularza (w dalszej części tego rozdziału przekonasz się, co taki serwer WWW może zrobić z tymi danymi).

Nie ma
niemądrych pytań

P: Co z pozostałymi metodami protokołu HTTP (poza omówionymi przed chwilą żadaniami GET i POST)?

U: GET i POST to dwie najczęściej spotykane metody żądań, z których na co dzień korzysta każdy użytkownik internetu. Istnieje jednak kilka innych, rzadziej stosowanych metod (które mogą być obsługiwane przez serwlety), należą do nich: HEAD, TRACE, PUT, DELETE, OPTIONS oraz CONNECT.

Aby pomyślnie przejść egzamin, naprawdę nie musisz zbyt dużo wiedzieć na temat tych metod, choć nie można oczywiście wykluczyć ich wystąpienia w którymś z pytań. Wszystkie niezbędne szczegóły pozostałych metod protokołu HTTP omówimy w rozdziale poświęconym życiu i śmierci serwletu.



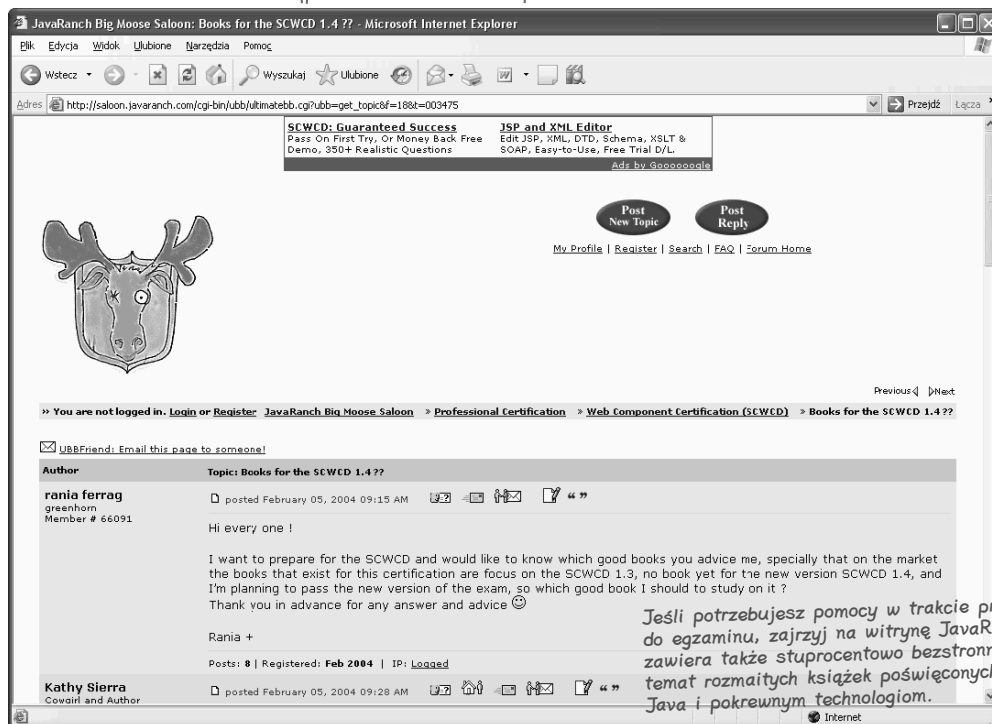
To prawda, można wysyłać niewielkie ilości danych także w ramach żądań GET protokołu HTTP

Z drugiej strony, możesz przecież nie chcieć, by podobne sytuacje miały miejsce. Oto kilka powodów, dla których lepiej używać żądań POST zamiast żądań GET:

- 1 Łączna liczba znaków w żądaniu GET w rzeczywistości jest ograniczona (w zależności od serwera WWW). Jeśli na przykład użytkownik wpisze długie wyrażenie w polu tekstowym *Szukaj*, metoda GET może nie zdać egzaminu.
- 2 Dane wysyłane razem z żądaniem GET są dołączane do adresu URL wyświetlanego później w odpowiednim pasku okna przeglądarki, zatem wszystkie te informacje są udostępniane użytkownikowi. Lepiej więc nie umieszczać w tego typu żądaniach hasła ani innych poufnych danych!
- 3 Z powodów wyjaśnionych w dwóch wcześniejszych punktach, jeśli zastosujesz metodę POST zamiast metody GET, użytkownik nie będzie mógł łatwo konstruować własnych żądań z (być może fałszywymi) danymi formularza. W zależności od charakteru budowanej aplikacji internetowej, możemy zdecydować, czy takie działania użytkownika są dopuszczalne, czy nie powinny mieć miejsca.

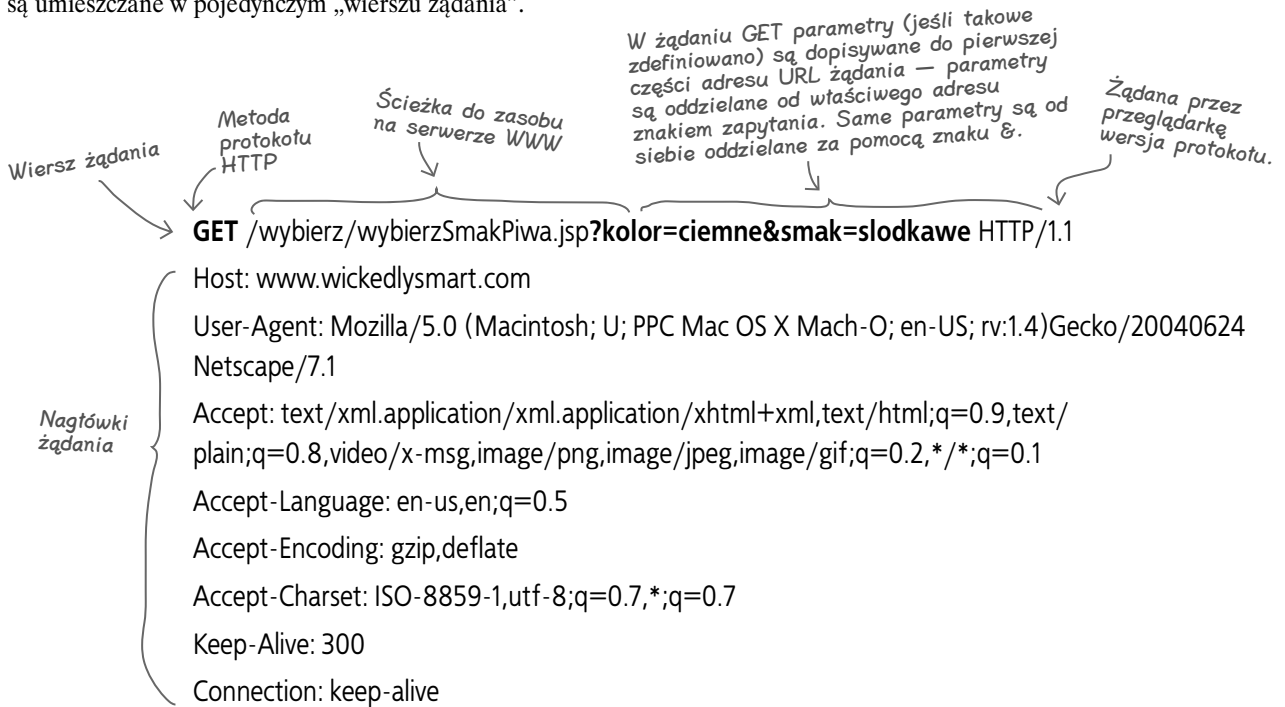
Znak zapytania (?) oddziela ścieżkę od parametrów żądania (dodatkowych danych). Ilość danych przesyłanych w ramach żądania GET jest ograniczona, a dane te zawsze są wyświetlane w odpowiednim pasku okna przeglądarki internetowej, zatem mogą być łatwo odczytywane przez każdego (wraz z całym ciągiem znaków przesyłanym w ramach żądania adresie URL).

Oryginalny adres URL przed dodatkowymi parametrami.



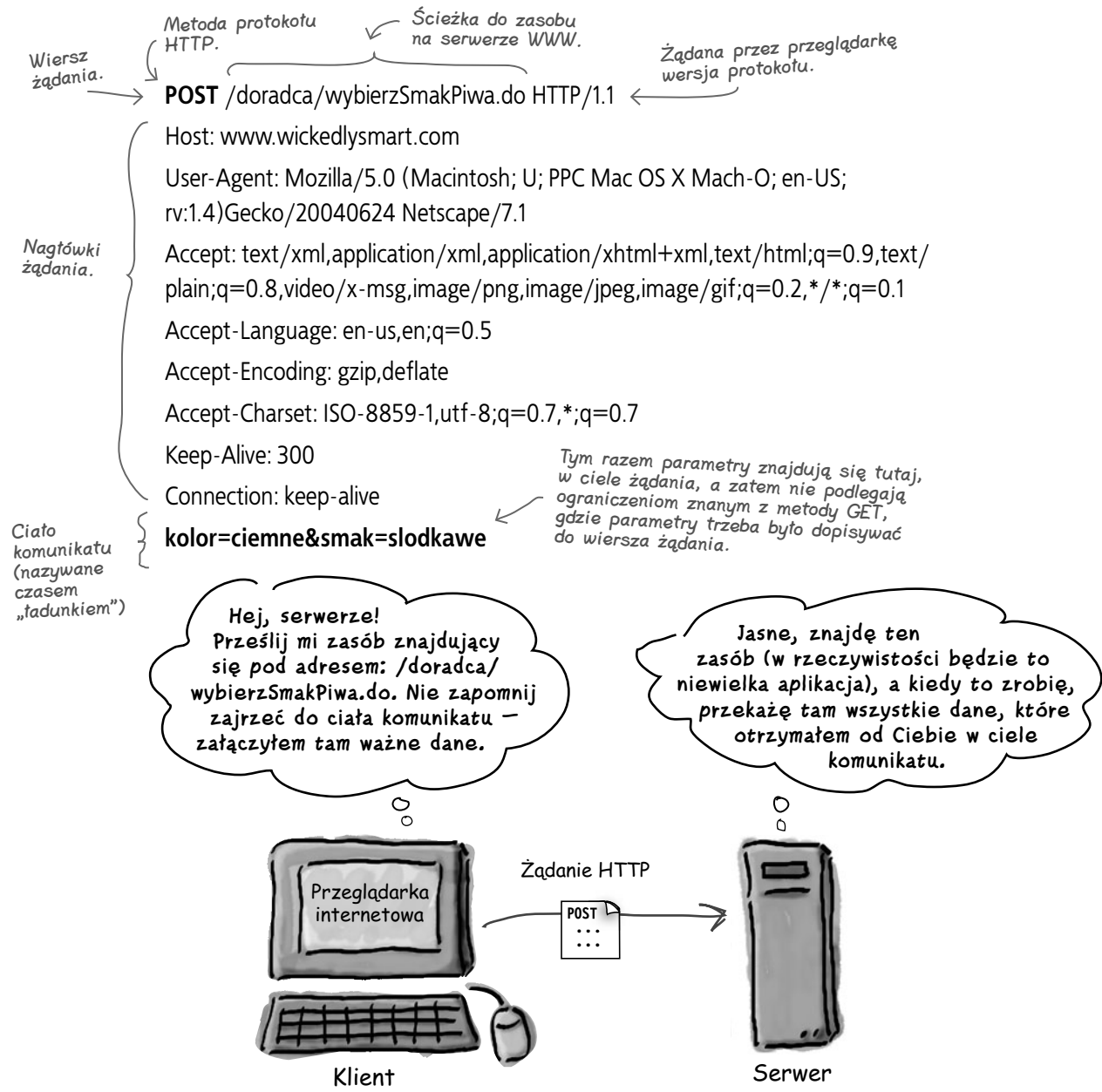
Anatomia żądania GET protokołu HTTP

Zarówno ścieżka do zasobu, jak i wszelkie parametry dodane do adresu URL są umieszczane w pojedynczym „wierszu żądania”.



Anatomia żądania POST protokołu HTTP

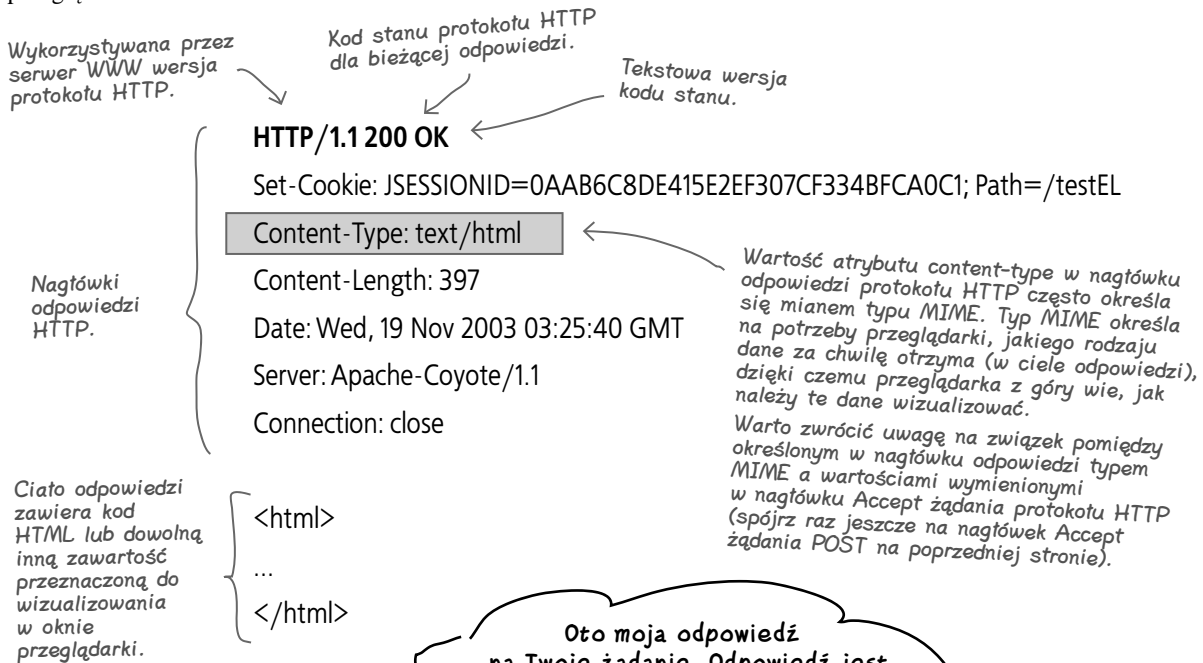
Żądania POST protokołu HTTP zostały zaprojektowane z myślą o przesyłaniu pomiędzy przeglądarkami internetowymi a serwerami WWW skomplikowanych poleceń. Jeśli na przykład użytkownik właśnie wypełnił obszerny formularz, zadaniem aplikacji może być umieszczenie wszystkich informacji z tego formularza w bazie danych. Dane (być może liczne) odsyłane w ten sposób do serwera określa się mianem *ciała komunikatu* lub *ładunku*.



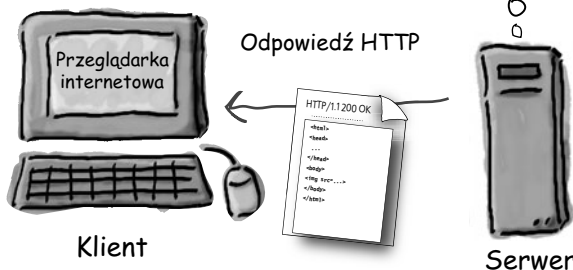
Anatomia odpowiedzi protokołu HTTP.

Czym u diabła jest ten cały „typ MIME”?

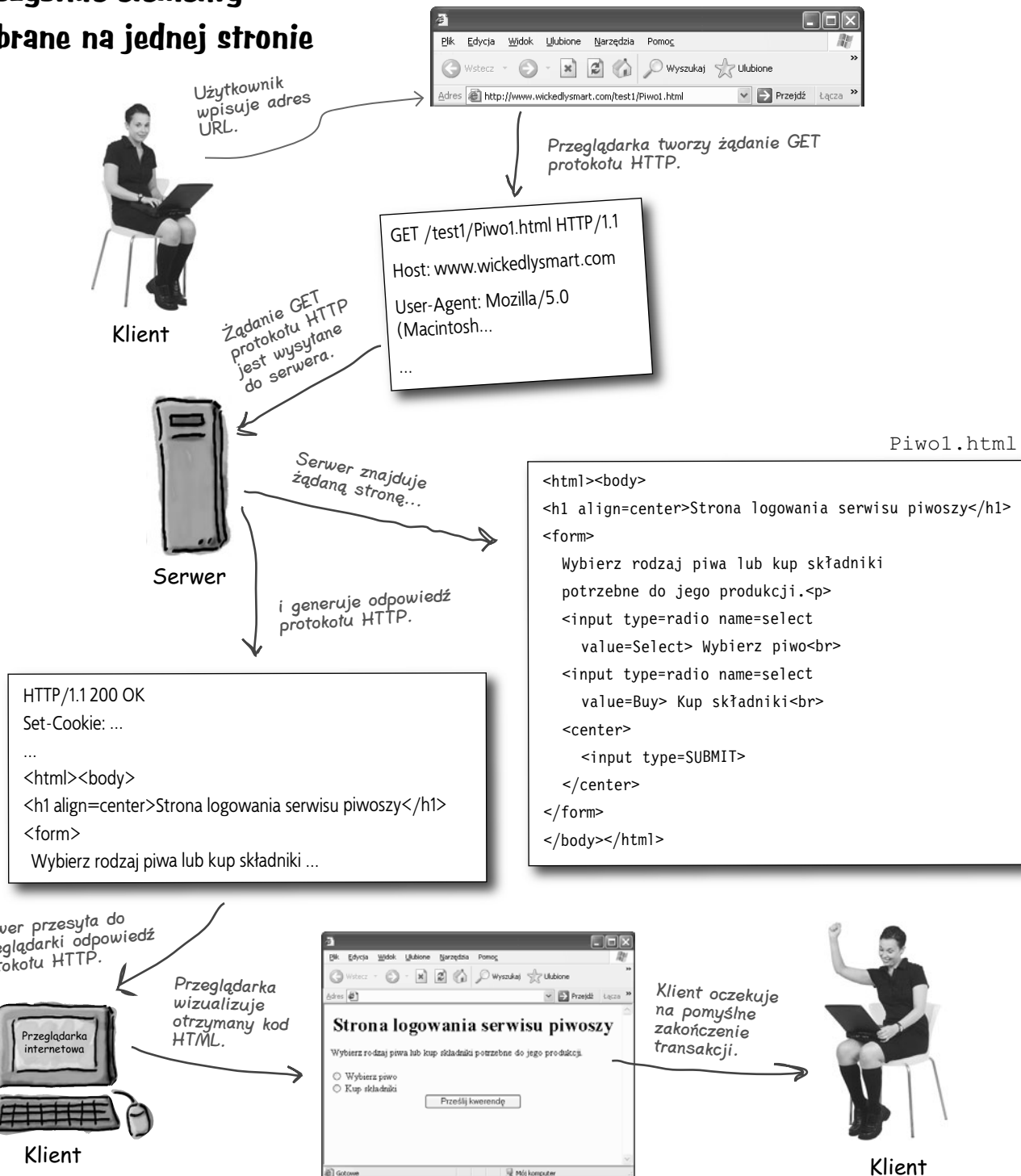
Skoro wiemy już, jak wyglądają zapytania przesyłane przez przeglądarkę do serwera WWW, warto teraz przeanalizować zawartość odsyłanego przez serwer komunikatu odpowiedzi. Odpowiedź protokołu HTTP zawiera zarówno nagłówki, jak i ciało. Informacje w nagłówku określają na potrzeby docelowej przeglądarki, którego protokołu użyto, czy zapytanie zostało pomyślnie przetworzone oraz jaki rodzaj zawartości umieszczono w ciele komunikatu odpowiedzi. Ciało odpowiedzi zawiera właściwą treść (np. kod HTML), która ma zostać zaprezentowana w oknie przeglądarki.



Oto moja odpowiedź na Twoje zapytanie. Odpowiedź jest typu text/html, ale jeśli mnie pamięć nie myli, twierdziłeś, że sobie z tym typem poradzisz. Jeśli więc wtedy mówiłeś prawdę...



Wszystkie elementy zebrane na jednej stronie





Zaostrz ołówek

GET czy POST?

Dla każdego opisu po prawej stronie oznacz kółkiem albo odpowiedź POST, albo odpowiedź GET w zależności od tego, którą z metod protokołu HTTP należy wybrać do zaimplementowania poszczególnych rozwiązań. Jeśli Twoim zdaniem obie metody pasują do danego rozwiązania, obrysuj te odpowiedzi razem. Przygotuj się jednak na uzasadnienie swoich odpowiedzi...

POST	GET	<i>Użytkownik zwraca identyfikator i hasło logowania.</i>
POST	GET	<i>Użytkownik wysyła żądanie nowej strony za pośrednictwem hiperłącza.</i>
POST	GET	<i>Użytkownik pokoju rozmów wysyła napisaną odpowiedź.</i>
POST	GET	<i>Użytkownik klika przycisk „Dalej”, aby przejść do następnej strony.</i>
POST	GET	<i>Użytkownik klika przycisk „Wyloguj” na stronie bezpiecznej witryny banku.</i>
POST	GET	<i>Użytkownik klika przycisk „Wstecz” na pasku standardowych przycisków przeglądarki.</i>
POST	GET	<i>Użytkownik wysyła do serwera formularz ze swoim nazwiskiem i adresem.</i>
POST	GET	<i>Użytkownik wybiera jeden z kilku przycisków opcji.</i>

URL. Cokolwiek robisz, nie wymawiaj „erl”

Przeglądając hasła na literę *U* w słowniku akronimów, szybko zorientujemy się, że jest tam zdecydowanie ciasno... URI, URL, URN i końca nie widać. Na razie skupmy się jednak na znanym i lubianym skrócie URL, który oznacza **Uniform Resource Locator**. Każdy zasób udostępniany w internecie ma swój własny, unikatowy adres zapisywany właśnie w formacie URL.

Protokół: Mówi serwerowi, który protokół komunikacyjny będzie stosowany (w tym przypadku będzie to oczywiście HTTP).

Port: Ta część adresu URL jest opcjonalna. Pojedynczy serwer może obsługiwać wiele portów. Każda aplikacja serwera jest identyfikowana właśnie za pomocą numeru portu. Jeśli nie określisz portu w adresie URL, zostanie użyty domyślny port 80. — jeśli dopisze Ci szczęście, będzie to jednocześnie domyślny port serwera WWW.

Zasób: Nazwa żądanej zawartości. Może to być strona HTML, serwet, obraz, dokument PDF, plik muzyczny, plik wideo lub dowolny inny zasób udostępniany przez serwer. Jeśli pominiemy tę opcjonalną część adresu URL, większość serwerów WWW domyślnie będzie szukała dokumentu `index.html`.

`http://www.wickedlysmart.com:80/poradaPiwna/wybierz/piwo1.html`

Serwer: Unikatowa nazwa szukanego przez nas fizycznego serwera. Podana nazwa jest odwzorowywana w unikatowy adres IP. Numeryczne adresy IP mają następującą postać: `xxx.yyy.zzz.aaa`. Możesz w tym miejscu użyć adresu IP zamiast nazwy serwera, jednak to nazwa serwera jest znacznie łatwiejsza do zapamiętania.

Ścieżka: Ścieżka określająca lokalizację żadanego zasobu na serwerze. Ponieważ większość wczesnych serwerów WWW pracowała w systemach operacyjnych Unix, do opisywania hierarchii katalogów serwerów WWW nadal stosuje się składnię tych systemów.

Element pominięty:

Opcjonalny tańcuch zapytania: Pamiętaj, że gdyby to było żądanie GET, do adresu URL zostałyby dopisane dodatkowe informacje (parametry), poczynwszy od znaku zapytania i ze znakiem & oddzielającym poszczególne parametry (pary nazwa-wartość).



Port protokołu TCP jest po prostu liczbą

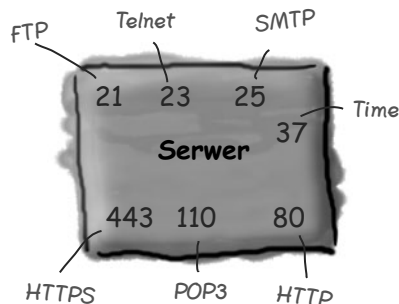
Port jest 16-bitową liczbą identyfikującą konkretne oprogramowanie działające na komputerze serwera.

Oprogramowanie naszego internetowego serwera WWW (HTTP) pracuje na porcie 80. To jeden ze standardów. Jeśli dodatkowo dysponujemy serwerem usługi Telnet, zapewne działa na porcie 23. FTP? 21. Serwer poczty elektronicznej POP3? 110. SMTP? 25. Serwer usługi Time pracuje na porcie 37. Wymienione porty należy traktować jak unikatowe identyfikatory usług (serwerów). Każdy port reprezentuje logiczne połączenie z określoną częścią oprogramowania działającego na komputerze serwera. No właśnie — jeśli odwrócisz swój komputer na biurku, nie znajdziesz w nim żadnego fizycznego portu TCP. Z jednej strony każdy serwer ma aż 65 536 takich portów (numerowanych od 0 do 65 535). Z drugiej strony porty protokołu TCP *nie* reprezentują fizycznych interfejsów, do których moglibyśmy podłączać urządzenia zewnętrzne. Są to po prostu liczby reprezentujące aplikacje serwera.

Bez numerów portów serwer nie mógłby określić, z którą aplikacją chce się połączyć dany klient. Ponieważ każda aplikacja może korzystać z własnego, unikatowego protokołu komunikacji klient-serwer, nietrudno sobie wyobrazić chaos, jaki zapanowałby w razie braku tego rodzaju identyfikatorów. Co by się stało, gdyby nasza przeglądarka połączyła się np. z serwerem poczty elektronicznej POP3 zamiast z serwerem HTTP? Serwer poczty nie wie przecież, jak należy przetwarzać żądania HTTP! Co więcej, nawet gdyby to wiedział, i tak nie potrafiłby odsyłać oczekiwanych przez klientów stron HTML.

Jeśli sam tworzysz usługi (programy serwera), które będą działały w lokalnej sieci komputerowej Twojej firmy, powinieneś sprawdzić z administratorami systemu, które porty są wolne, a które są zajęte przez usługi już istniejące. Administratorzy systemu mogą Ci powiedzieć, że np. możesz wykorzystać dowolny numer portu poniżej, powiedzmy, portu 3000.

Najbardziej popularne numery portów protokołu TCP dla powszechnie stosowanych aplikacji



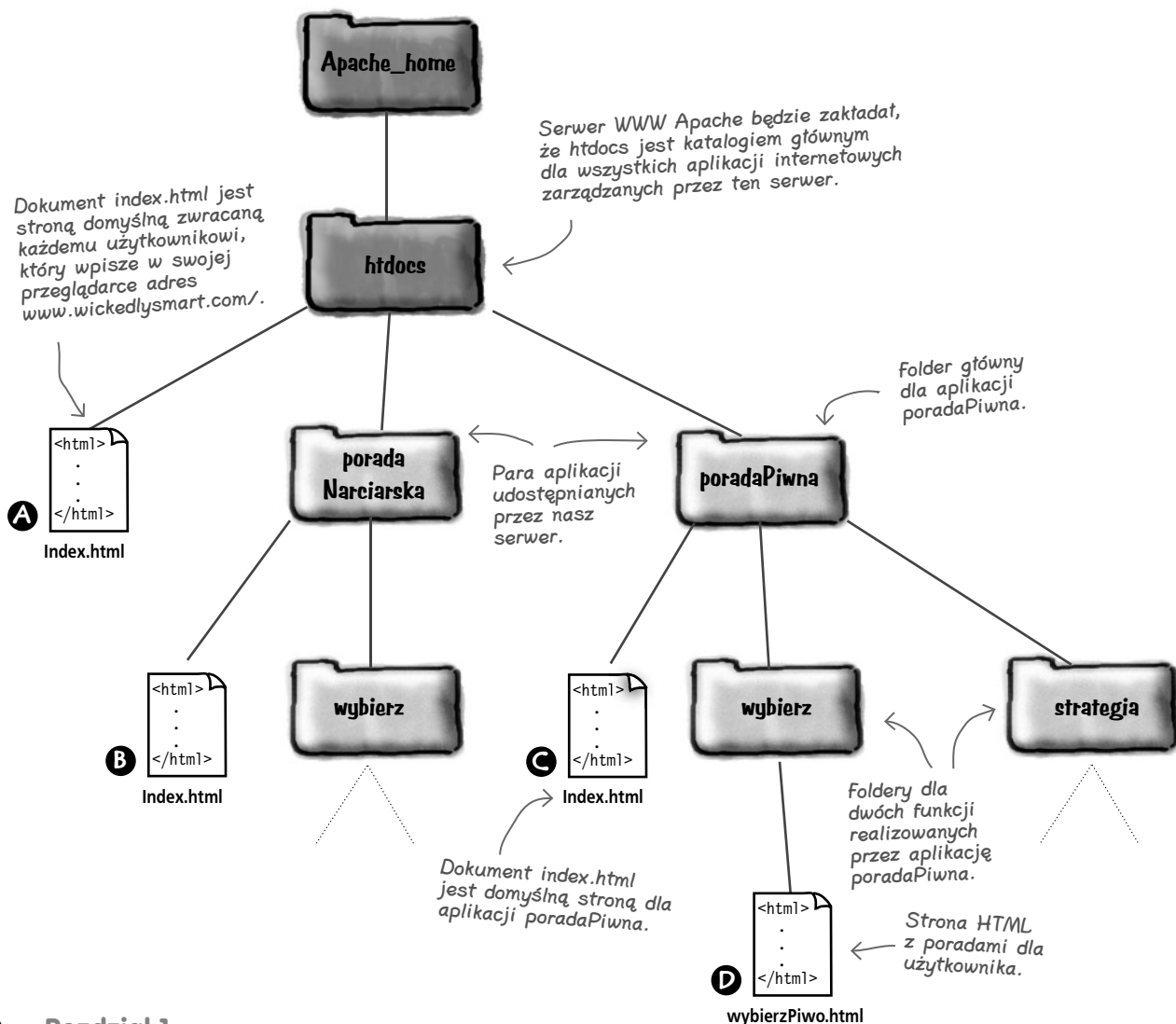
Gdybyśmy użyli po jednej aplikacji serwera dla każdego z dostępnych portów, pojedynczy serwer mógłby udostępniać aż 65 536 różnych aplikacji serwera.

Numery portów TCP z przedziału od 0 do 1023 są zarezerwowane dla najpopularniejszych usług (włącznie z usługą, która interesuje nas najbardziej — na porcie 80.). Nie należy używać tych portów dla własnych programów serwera!

Struktura katalogów dla prostej witryny internetowej zarządzanej przez serwer Apache

Serwery Apache i Tomcat omówimy bardziej szczegółowo w dalszej części tej książki; na razie przyjmijmy, że nasza prosta witryna internetowa wykorzystuje serwer Apache (którego najprawdopodobniej już używasz, ponieważ jest to wyjątkowo popularny serwer z otwartym dostępem do kodu źródłowego). Jak powinna wyglądać struktura katalogów witryny internetowej nazwanej *www.wickedlysmart.com* i obejmującej dwie aplikacje (jedną z poradami narciarskimi i drugą z poradami związanymi z piwem)? Przyjmijmy, że aplikacja Apache działa na porcie 80.

Strony *.html* oznaczono literami (A, B, C, D) z myślą o ćwiczeniu na kolejnej stronie.

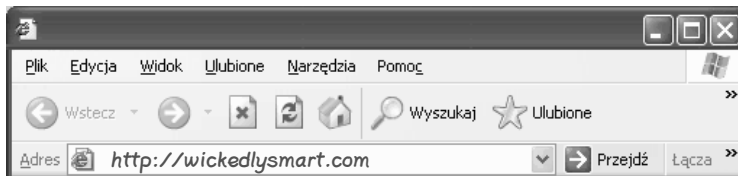




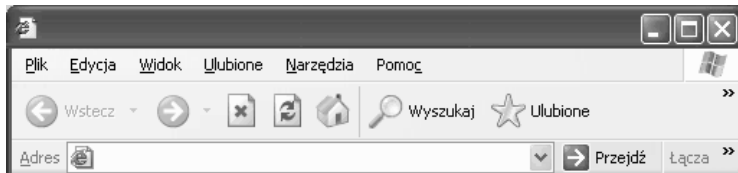
Zaostrz ołówek

Odwzorowywanie adresów URL w konkretną treść

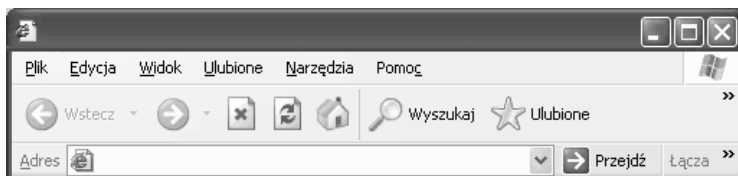
Przyjrzyj się raz jeszcze strukturze katalogów z poprzedniej strony i spróbuj zapisać adres URL, który doprowadzi Cię do każdej z czterech stron *.html* oznaczonych literami *A*, *B*, *C* i *D*. Pierwszy adres (*A*) zapisałeś za Ciebie — tacy już jesteśmy. Na potrzeby tego ćwiczenia przyjmij, że serwer Apache działa na porcie 80. protokołu TCP (prawidłowe odpowiedzi znajdują się na dole kolejnej strony).



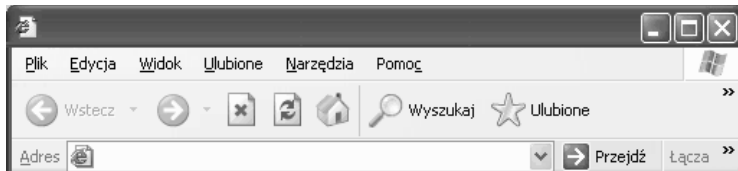
spowoduje zwrócenie przez serwer strony *index.html* składowanej w katalogu

A

spowoduje zwrócenie przez serwer strony *index.html* składowanej w katalogu

B

spowoduje zwrócenie przez serwer strony *index.html* składowanej w katalogu

C

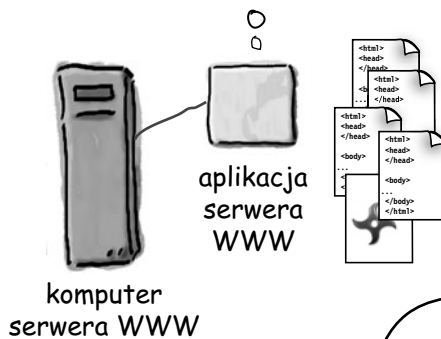
spowoduje zwrócenie przez serwer strony *wybierzPiwo.html* składowanej w katalogu

D

Serwery WWW uwielbiają udostępniać statyczne strony internetowe

Oto jak działam. Poproś mnie o konkretną stronę, a ja ją dla Ciebie znajdę, dotączę trochę nagłówków i niezwłocznie odeślę. To wszystko – nie prosz mnie, abym cokolwiek robił z odnalezioną stroną!

Strona statyczna jest po prostu przechowywana w odpowiednim katalogu aplikacji serwera. Serwer odnajduje ją i odsyła do klienta dokładnie w takiej postaci, w jakiej jest składowana na dysku. Każdy klient otrzymuje dokładnie taki sam dokument.



Te strony trafiają prosto do klienta w dokładnie takiej postaci, w jakiej zostały umieszczone na serwerze WWW.

Co powinienem zrobić, gdybym chciał umieścić na mojej stronie np. aktualną godzinę? Jak można umieścić na stronie jakieś elementy dynamiczne? Czy mogę używać w kodzie HTML takich konstrukcji jak zmienne?

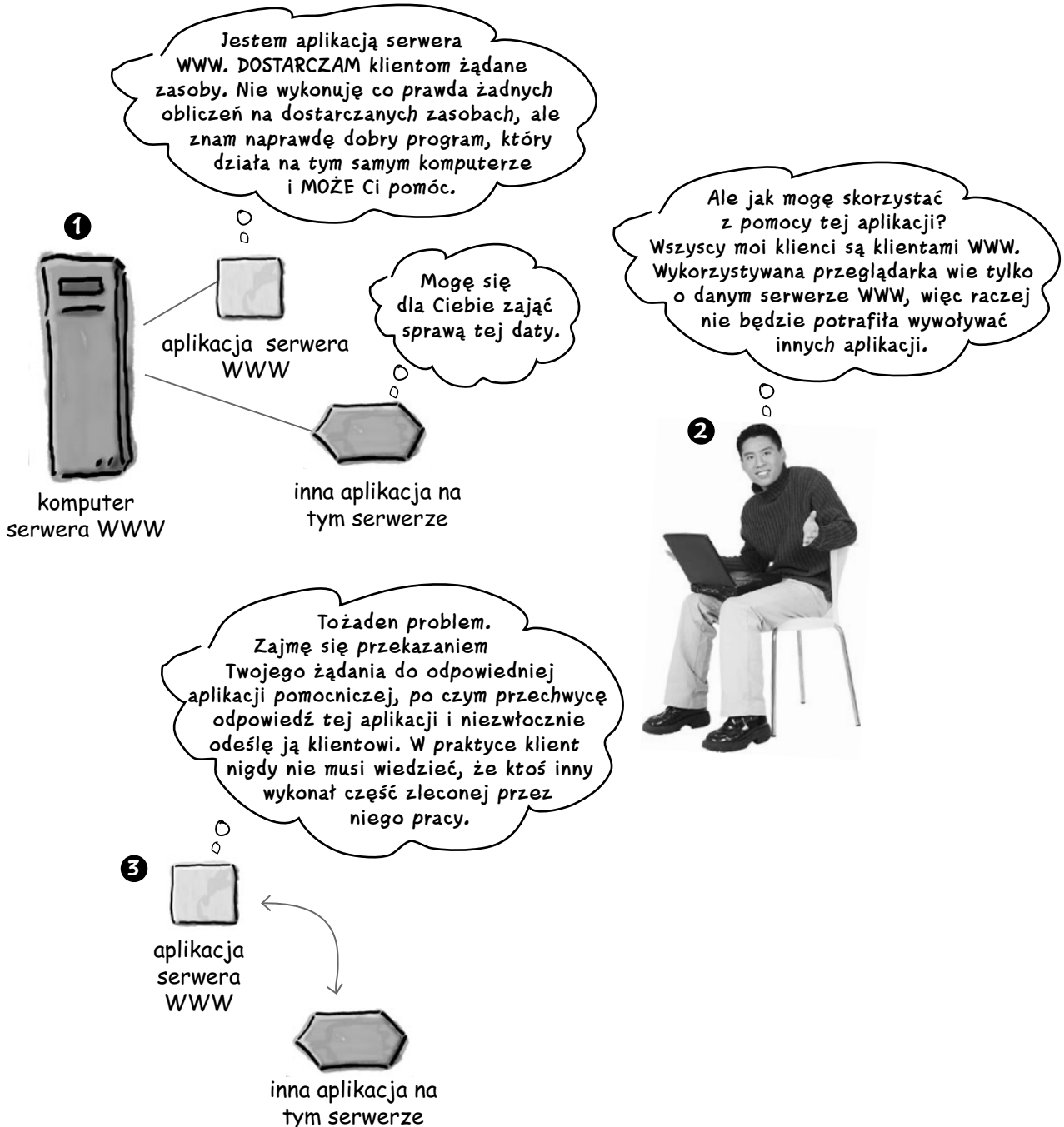
Co zrobić, aby na naszej stronie HTML znalazły się jakieś elementy zmienne?

```
<html>
<body>
  Jest godzina [wstawCzasSerwera].
</body>
</html>
```

B: www.wickedlysmart.com/poradaNarcarska
C: www.wickedlysmart.com/poradaPiwna
D: www.wickedlysmart.com/wybiez/wybiezPiwo.html

Odpowiedzi z poprzedniej strony:

Czasem jednak oczekujemy czegoś więcej niż tylko serwera WWW



Dwóch rzeczy serwer WWW sam na pewno nie zrobi

Jeśli potrzebujemy stron *just-in-time* (czyli takich stron tworzonych dynamicznie, które nie istnieją na serwerze przed otrzymaniem żądania) i musimy mieć możliwość wprowadzania i zapisywania danych na serwerze (a więc możliwość zapisu w pliku lub bazie danych), nie możemy polegać wyłącznie na serwerze WWW.

1. Zawartość dynamiczna

Co prawda samodzielna aplikacja serwera WWW może udostępniać jedynie statyczne strony internetowe, jednak już zewnętrzna aplikacja „pomocnicza” (z którą taki serwer może się komunikować) może budować niestacyjne strony *just-in-time*. Strona dynamiczna może być czymkolwiek — od katalogu produktów po blog internetowy, lub po prostu stroną wyświetlającą losowo wybierany obraz.

Jeśli zamiast tego:

```
<html>
<body>
Na tym serwerze
zawsze jest godzina
16:20.
</body>
</html>
```

Chcemy otrzymać coś takiego:

```
<html>
<body>
Na tym serwerze
jest godzina
[wstawCzasSerwera].
</body>
</html>
```

Strony *just-in-time* (w skrócie JIT) nie istnieją, zanim na serwer nie trafi odpowiednie żądanie. Obsługa takich żądań polega więc na tworzeniu stron HTML w locie.

Kiedy do serwera WWW dociera żądanie, to aplikacja pomocnicza „zapisuje” odpowiedni kod HTML, a serwer WWW ogranicza się do odesłania tak wygenerowanej strony klientowi.

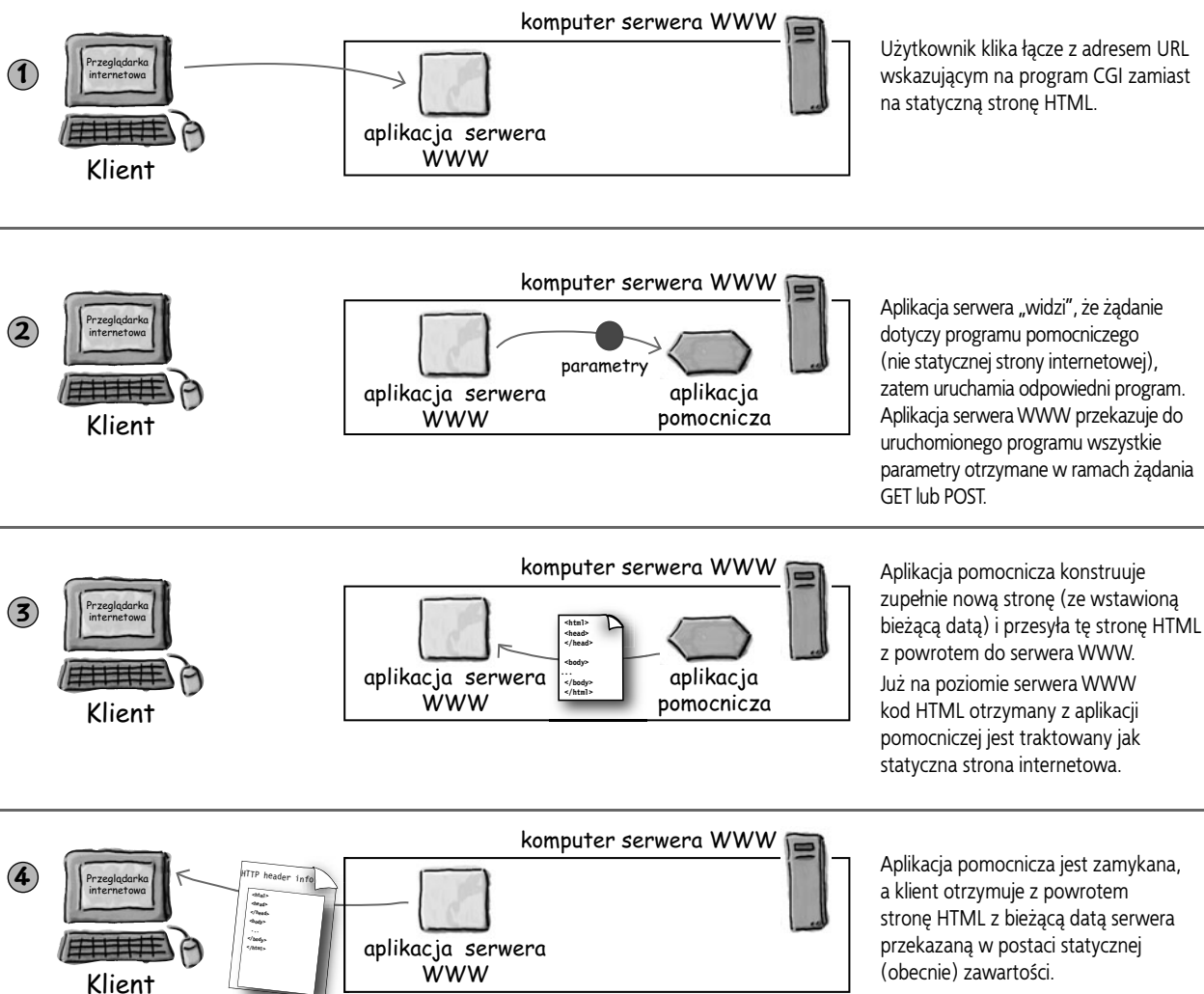
2. Zapisywanie danych na serwerze

Kiedy użytkownik wysła dane wpisane w formularzu, serwer WWW przegląda otrzymane informacje i myśli: „Cóż ja mogę z tym zrobić?”. Do przetworzenia danych z formularza — niezależnie od tego, czy ma ono polegać na ich zapisie do pliku lub bazy danych, czy tylko na wygenerowaniu odpowiedniej strony odpowiedzi — niezbędne jest użycie innej aplikacji. Kiedy serwer WWW widzi żądanie wymagające odwołania do aplikacji pomocniczej, zakłada, że także otrzymane parametry są przeznaczone dla tej aplikacji. Serwer WWW przekazuje więc te parametry dalej, zrzucając odpowiedzialność za wygenerowanie odpowiedzi dla klienta właśnie na aplikację pomocniczą.

Rozwiązaniem spoza świata Javy problemu aplikacji pomocniczych jest program „CGI”

Większość programów CGI ma postać skryptów języka Perl, co nie oznacza, że nie można do tego celu używać wielu innych języków programowania, jak C, Python czy PHP (skrót CGI pochodzi od ang. *Common Gateway Interface* — niespecjalnie nas interesuje, dlaczego użyto właśnie takiej nazwy).

Poniżej przedstawiamy możliwy sposób wykorzystania technologii CGI do tworzenia dynamicznych stron internetowych z aktualną datą serwera.



Zarówno serwlety, jak i programy CGI odgrywają rolę aplikacji pomocniczych serwera WWW

Posłuchaj dyskusji na temat zalet i wad programów CGI oraz serwletów prowadzonej przez naszych dwóch właścicieli czarnych pasów.

CGI



Serwlety



Programy CGI są lepsze od serwletów. Skrypty CGI można łatwo tworzyć w Perlu, a jak wiadomo, wszyscy znają Perla.

Rozumiem, że używanie Javy nie sprawia Ci kłopotów, ponieważ znasz ten język programowania. Z pewnością jednak nie warto przechodzić na Javę z innych języków, ponieważ nie oferuje ona nic nowego.

Chcesz mnie sprowokować? Masz jakieś konkretne argumenty?

Tu akurat nie ma różnicy pomiędzy Perlem a Javą... jak wy to nazywacie? Wirtualna maszyna Javy (JVM)? Czyż każdy egzemplarz wirtualnej maszyny Javy nie jest odrębnym, ciężkim procesem?

Widzę, że zapomniałeś o kilku istotnych kwestiach. Serwery WWW umożliwiają obecnie utrzymywanie pojedynczego programu napisanego w Perlu, który działa także pomiędzy kolejnymi żądaniami klientów. Argument o dodatkowych kosztach jest więc nietrafiony.

O czym Ty mówisz? Każdy komponent zgodny ze standardem CORBA może być klientem J2EE.

Muszę kończyć — spóźnię się na zajęcia Pilates. Ale pamiętaj, że to jeszcze nie koniec; dokończymy naszą dyskusję nieco później.

Wątpię, żeby wszyscy znali Perla. Lubię Perla, ale na co dzień wszyscy pracujemy w Javie, zatem wolalbym wybrać właśnie ten język.

Z całym szacunkiem, mistrzu, Java ma wiele zalet, których z pewnością nie ma Perl — różnice są widoczne szczególnie wtedy, gdy próbujemy tworzyć analogiczne rozwiązania w oparciu o serwlety Javy oraz skrypty Perla obsługiwane przez CGI.

Choćby wydajność. Skrypty Perla i technologia CGI wymaga od serwera tworzenia osobnego procesu dla każdego otrzymanego żądania zasobu!

Tak, to prawda, ale serwlety pozostają załadowane w pamięci i każde kolejne żądanie klienta dotyczące załadowanego wcześniej serwletu jest obsługiwane w formie osobnego *wątku* tego samego serwletu. Oznacza to, że przetwarzanie tego typu żądań nie wymaga każdorazowego uruchamiania wirtualnej maszyny Javy, wczytywania klas i innych podobnych działań...

O niczym nie zapomniałem, mistrzu. Wiesz przecież, że nie wszystkie serwery WWW potrafią to robić. Wspominasz o specjalnym przypadku, który nie ma zastosowania dla wszystkich programów CGI napisanych w Perlu. W tym względzie serwlety Javy zawsze będą bardziej efektywne. Nie zapominaj także, że serwlet może być klientem J2EE, co nie jest możliwe w przypadku napisanego w Perlu programu CGI.

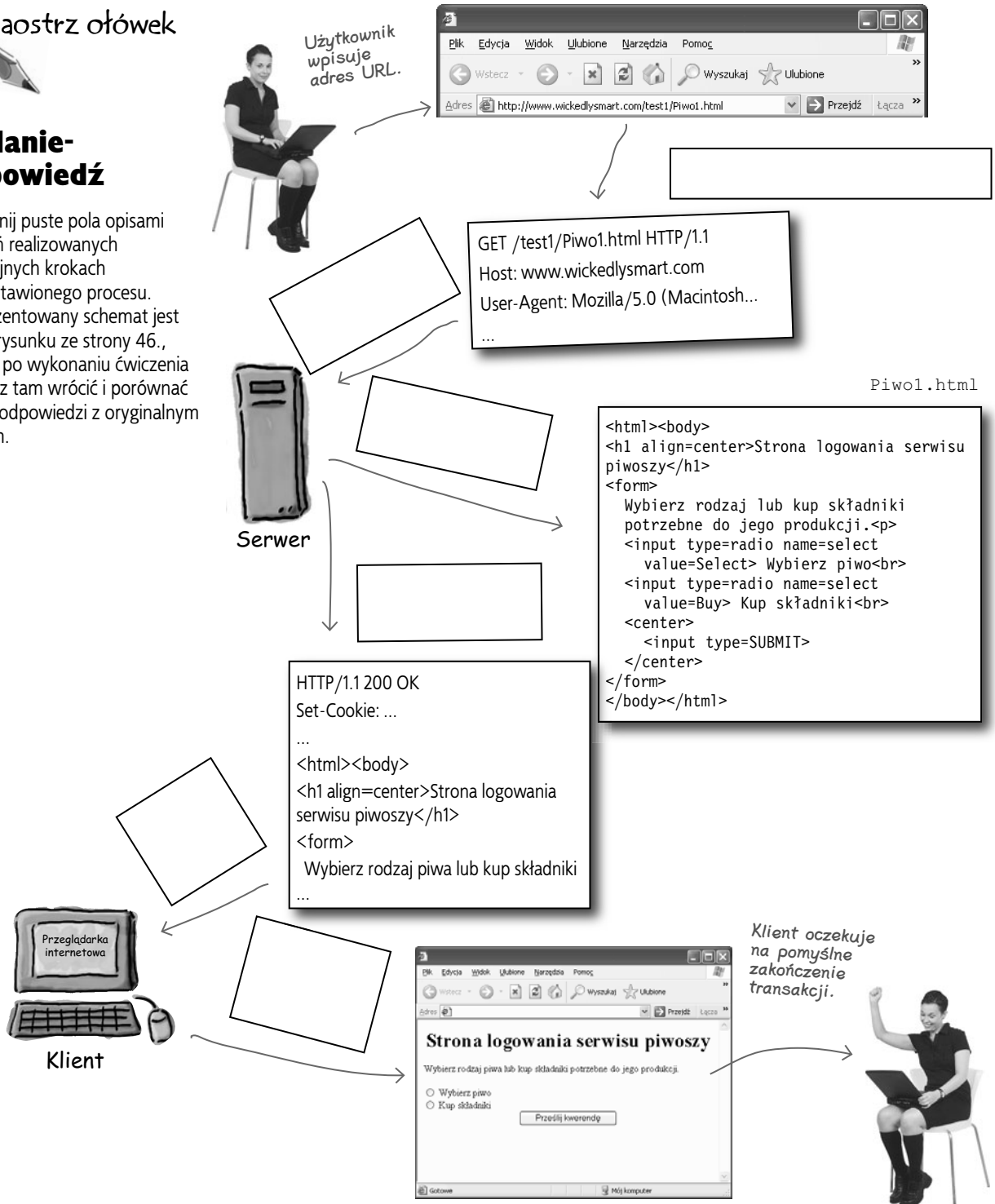
Nie chodzi mi o klienta *względem* programu J2EE, ale o klienta *będącego* programem J2EE. Serwlet działający w odpowiednim kontenerze J2EE może uczestniczyć w operacjach związanych z bezpieczeństwem i w transakcjach razem z komponentami J2EE, poza tym istnieją...

Ciąg dalszy nastąpi...



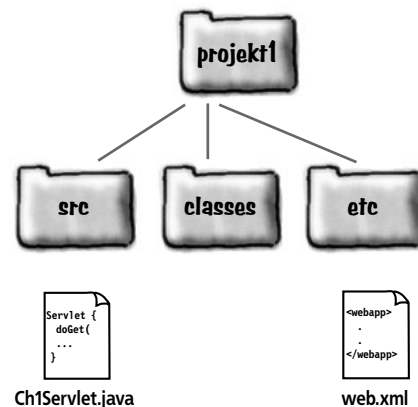
Żądanie- odpowieź

Wypełnij puste pola opisami działań realizowanych w kolejnych krokach przedstawionego procesu. Zaprezentowany schemat jest kopią rysunku ze strony 46., zatem po wykonaniu ćwiczenia możesz tam wrócić i porównać swoje odpowiedzi z oryginalnym opisem.



Serwlety bez tajemnic (pisanie, wdrażanie, uruchamianie)

Aby zaspokoić ciekawość zniecierpliwionych Czytelników, którzy nie mieli wcześniej do czynienia z serwletami, przedstawimy teraz błyskawiczną demonstrację pisania, wdrażania i uruchamiania serwletu. Taka prezentacja może oczywiście w większym stopniu być źródłem dodatkowych pytań niż rozwiewać już istniejące wątpliwości — **nie panikuj**, nie musisz jeszcze przystępować do samodzielnego *pisania* serwletów. Niniejsza demonstracja jest przeznaczona dla tych Czytelników, którzy nie mogli się tego doczekać. Kolejny rozdział tej książki zawiera bardziej szczegółowy materiał na ten temat.



- 1 Zbuduj następujące drzewo katalogów (gdzieś *poza* strukturą katalogów serwera Tomcat).
- 2 Napisz serwlet `Ch1Servlet` i umieść go w pliku `Ch1Servlet.java` w katalogu `src` (dla zachowania prostoty tego przykładu nie umieszczamy naszego serwletu w pakiecie — we wszystkich kolejnych przykładach serwlety będą umieszczane w odpowiednich pakietach).

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
```

```
public class Ch1Servlet extends HttpServlet {

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
                      throws IOException {
```

Standardowe deklaracje
serwletu (ich opis zajmuje
około 400 stron).

```
        PrintWriter out = response.getWriter();
        java.util.Date dzisiaj = new java.util.Date();
        out.println("<html> " +
                    "<body> " +
                    "<h1 align=center>Nasz pierwszy serwlet: Ch1Servlet</h1>" +
                    "<br>" + dzisiaj + "</body>" + "</html>");
```

Kod HTML osadzony
w programie napisanym
w języku Java. Wygląda
uroczo, prawda?

- 3 Stwórz deskryptor wdrożenia (ang. *Deployment Descriptor* — *DD*) nazwany `web.xml` i umieść go w podkatalogu `etc` w katalogu `projekt1`.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
  version="2.4">
  <servlet>
    <servlet-name>Chapter1 Servlet</servlet-name>
    <servlet-class>Ch1Servlet</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>Chapter1 Servlet</servlet-name>
    <url-pattern>/Serv1</url-pattern>
  </servlet-mapping>
</web-app>
```

Najważniejsze wnioski:

- Po jednym deskrytorze wdrożenia dla każdej aplikacji internetowej.
- Pojedynczy deskryptor może deklarować wiele serwletów.
- Element `<servlet-name>` wiąże element `<servlet>` z elementem `<servlet-mapping>`.
- Element `<servlet-class>` wskazuje klasę Javy.
- Element `<url-pattern>` określa nazwę wykorzystywaną w żądaniach klientów.

4 W istniejącym katalogu *tomcat* zbuduj następujące drzewo katalogów...

5 W katalogu *projekt1* skompiluj nasz serwet...

```
%javac -classpath /Twoja ścieżka/tomcat/common/lib/servlet-api.jar -d
classes src/Ch1Servlet.java
```

(cały ten tekst to jedno polecenie)

(w katalogu *projekt1/classes* zostanie utworzony plik *Ch1Servlet.class*)

6 Skopiuj plik *Ch1Servlet.class* do katalogu *WEB-INF/classes*, natomiast plik *web.xml* do katalogu *WEB-INF*.

7 W katalogu *tomcat* uruchom serwer WWW Tomcat...

%bin/startup.sh

8 Uruchom swoją przeglądarkę internetową i w polu adresu wpisz następujący tekst:

<http://localhost:8080/ch1/Serv1>

Aplikacja internetowa została nazwana *ch1*, natomiast sam serwet nazwaliśmy *Serv1*.

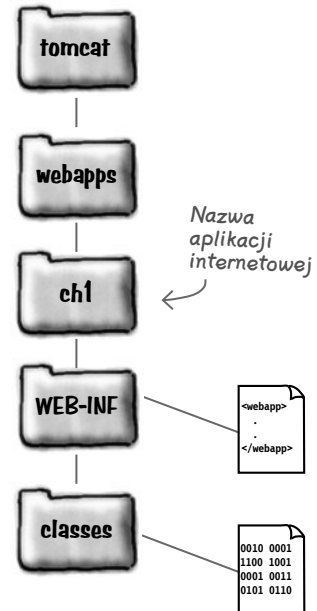
W oknie przeglądarki powinna zostać wyświetlona strona:



Data wyświetlona w Twojej przeglądarce oczywiście może być inna...

9 Od tej pory każda aktualizacja klasy serwetu lub deskryptora wdrożenia będzie wymagała zatrzymania serwera Tomcat:

%bin/shutdown.sh



jesteś tutaj ▶ 59

Bez obrazu, ale
w przedstawionym przykładzie
serwletu popełniono chyba POWAŻNY
błąd... jak można było umieścić kod HTML
w wywołaniu metody `println()`??
To chyba jakieś nieporozumienie...

W taki właśnie sposób tworzymy
dynamiczną stronę internetową za pomocą
serwletu Javy. Musimy zapisać kompletny
kod strony w strumieniu wyjściowym
(w rzeczywistości nasz kod jest częścią
strumienia odpowiedzi HTTP).

```
out.println("<html> " +  
    "<body> " +  
    "<h1>Strona logowania Skyler</h1>" +  
    "<br>" + dzisiaj +  
    "</body>" +  
    "</html>");
```

Formatowanie stron HTML za pomocą wywołań metody `out.println()` stosowanych w kodzie serwletów rzeczywiście jest kiepskim rozwiązaniem

Jest to jedna z najgorszych części (nie, ta część jest zdecydowanie *najgorsza*) całej technologii serwletów Javy. Osadzanie prawidłowo sformatowanych znaczników języka HTML w wywołaniach metody `println()` tylko po to, by mieć możliwość wstawiania tam zmiennych i wywołań innych metod, jest rozwiązaniem mało eleganckim, wręcz brutalnym. Nawet *nie myśl* o tworzeniu w ten sposób jakichkolwiek (choćby odrobinę) rozbudowanych rozwiązań.

Nie ma niemądrych pytań

P: Na pewno nie jest aż tak źle... czemu nie mogę po prostu skopiować całej strony HTML z mojego edytora stron internetowych (np. Dreamweavera) i wkleić w wywołaniu metody `println()`? Nie chodzi przecież o to, aby wklejony kod HTML był w tym miejscu szczególnie czytelny.

U: Oczywiście nie próbowaliśmy jeszcze tego typu rozwiązań. Brzmi rzeczywiście zachęcająco. Tak, to prawda, że można opracowywać strony internetowe w odpowiednich edytorach (lub nawet w prostym pliku tekstowym, który częstokroć jest łatwiejszy w edycji niż kod języka programowania Java), po czym szybko skopiować i wkleić gotowy kod do wywołania metody `println()`. To wszystko! Jest tylko jeden problem — otrzymamy około 1378 błędów kompilatora.

Pamiętaj, że w stałej łańcuchowej (typu `String`) nie można stosować (prawdziwego) znaku powrotu karetki. A skoro mowa o łańcuchach... co z interpretacją wszystkich znaków cudzysłowu użytych w kodzie HTML?

Och, gdyby
tylko istniał sposób
umieszczania kodu Javy
w ramach stron HTML, zamiast
stosowania kodu HTML
w klasach Javy.



Ona jeszcze nie wie o JSP

```
<html>
<body>
<h1>Strona logowania Skyler</h1>
<br>
<%= new java.util.Date() %>
</body>
</html>
```

Wspaniale! Wygląda na to,
że umieściliśmy odrobinę
kodu Javy w środku strony
HTML!?

skylerlogin.jsp

Strona JSP z pozoru nie różni się od zwykłej strony HTML
— w ramach strony JSP można jednak stosować elementy języka
programowania Java i konstrukcje pokrewne. Mamy więc wreszcie
możliwość uzupełniania kodu HTML o elementy dynamiczne.

Technologia JSP powstała przez wprowadzenie języka Java do kodu HTML

Umieszczanie kodu Javy w ramach stron HTML rozwiązuje dwa problemy:

1. Nie wszyscy projektanci stron HTML znają Javę

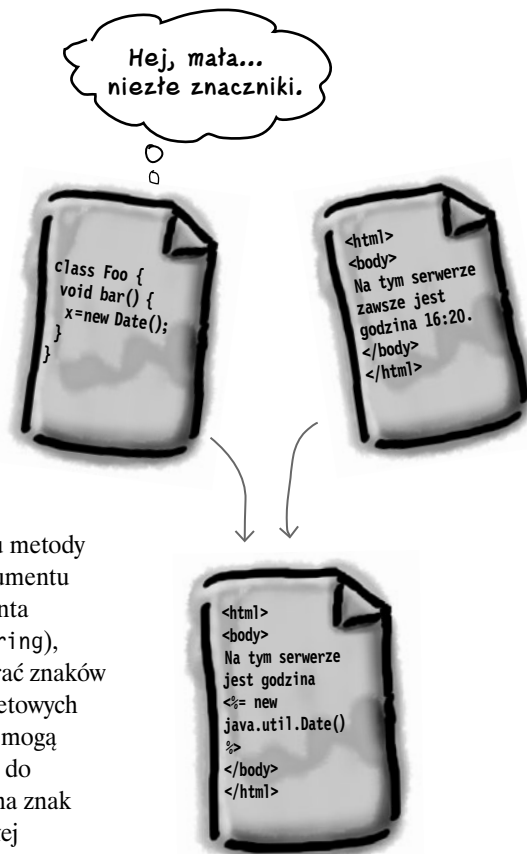
Programiści aplikacji znają Javę, natomiast programiści stron internetowych znają HTML-a. Dzięki technologii JSP programiści Javy mogą tworzyć kod w swoim ulubionym języku, a znawcy języka HTML mogą nadal konstruować strony internetowe.

2. Formatowanie kodu HTML w ramach serwletu Javy jest **NAPRAWDĘ** kłopotliwe

Umieszczając nawet bardzo nieskomplikowany kod HTML w ramach argumentu metody `println()`, aż prosimy się o błędy kompilatora. Prawidłowe sformatowanie dokumentu HTML do postaci, która gwarantuje możliwość wyświetlenia w przeglądarce klienta i jednocześnie zgodność z regułami obowiązującymi łańcuchy Javy (stałe typu `String`), może wymagać nadludzkiego wysiłku. Łańcuch Javy nie może na przykład zawierać znaków powrotu karetki, a większość kodu HTML kopiowanego z edytorów stron internetowych będzie zawierała właśnie ten znak na końcu każdego wiersza. Innym problemem mogą być znaki cudzysłowu — znaczna część znaczników HTML wykorzystuje te znaki do definiowania np. wartości atrybutów. Wiemy przecież, jak kompilator zareaguje na znak cudzysłowu w przetwarzanym kodzie... pomyśli sobie: „to musi być koniec tej stałej łańcuchowej”. Możemy oczywiście wrócić do programu i zastąpić każdy znak cudzysłowu odpowiednią sekwencją ucieczki, jednak takie rozwiązanie zawsze jest szalenie ryzykowne.

P: Zaczekaj... coś nadal jest nie tak! Pierwsza wymieniona korzyść mówi, że „nie wszyscy projektanci stron HTML znają Javę”, ale przecież projektanci stron HTML nadal muszą pisać kod Javy w ramach stron JSP!! Technologia JSP zwalnia co prawda programistów Javy z konieczności pisania kodu HTML, ale w praktyce nie oferuje niczego nowego projektantom stron HTML. Pisanie kodu HTML w ramach stron JSP jest być może *łatwiejsze* niż w przypadku wywołań metody `println()`, ale nie zwalnia twórców stron HTML z konieczności opanowania Javy.

U: Na pierwszy rzut oka faktycznie tak to wygląda. Jednak zgodnie z nową specyfikacją technologii JSP oraz stosując się do powszechnie znanych zaleceń, projektant stron internetowych powinien umieszczać bardzo niewiele (lub nie umieszczać *wcale*) rzeczywistego kodu Javy w stronach JSP. Projektanci stron internetowych muszą się oczywiście czegoś nauczyć, jednak ich zadanie w większości przypadków sprowadza się do umieszczania instrukcji *wywołujących* gotowe metody Javy, a więc nie polega na osadzeniu rzeczywistego kodu Javy w samych stronach JSP. Zamiast zapoznawać się z językiem programowania Java, programiści HTML-a muszą się jedynie nauczyć składni JSP.



KLUCZOWE ZAGADNIENIA



- HTTP jest akronimem słów HyperText Transfer Protocol i odnosi się do protokołu sieciowego wykorzystywanego do przesyłania stron WWW. Protokół HTTP działa ponad protokołem TCP/IP.
- Protokół HTTP wykorzystuje model żądanie-odpowiedź — klient wysyła żądanie HTTP, a serwer WWW zwraca właściwą odpowiedź HTTP, która z kolei jest obsługiwana przez przeglądarkę internetową (w zależności od rodzaju zwróconych treści).
- Jeśli odpowiedź serwera ma postać strony HTML, odpowiedni kod HTML jest dodawany do odpowiedzi protokołu HTTP.
- Żądanie protokołu HTTP obejmuje adres URL (wskazujący zasób, do którego klient chce uzyskać dostęp), metodę HTTP (GET, POST etc.) oraz opcjonalne dane parametrów formularza (nazywane także *łańcuchem zapytania*).
- Odpowiedź protokołu HTTP obejmuje kod stanu (kod błędu), typ zawartości (znany także jako typ MIME) oraz właściwą treść odpowiedzi (kod HTML, obraz etc.).
- W żądaniach GET dane z formularza są dopisywane do adresu URL.
- W żądaniach POST dane z formularza umieszcza się w ciele żądania.
- Typ MIME określa na potrzeby przeglądarki, jakiego rodzaju dane zostaną przesłane w ciele odpowiedzi, dzięki czemu przeglądarka może wybrać właściwy tryb obsługi tych danych (wizualizację kodu HTML, wyświetlenie grafiki, odtworzenie muzyki etc.).
- URL jest akronimem słów *Uniform Resource Locator*. Każdy zasób udostępniany w internecie ma przypisany swój unikatowy adres właśnie w tym formacie. Adres URL rozpoczyna się od określenia protokołu, bezpośrednio potem występuje nazwa serwera, opcjonalny numer portu oraz zazwyczaj konkretna ścieżka i nazwa żadanego zasobu. Jeśli adres URL jest częścią żądania GET, może dodatkowo zawierać opcjonalny łańcuch zapytania.
- Serwery WWW doskonale sprawdzają się w roli oprogramowania udostępniającego statyczne strony HTML; jeśli jednak musimy na stronie umieścić dynamicznie generowane dane (np. aktualną godzinę), powinniśmy użyć jakiejś aplikacji pomocniczej, która będzie współpracowała z naszym serwerem. Jednym z najbardziej popularnych rozwiązań w kwestii aplikacji pomocniczych, choć niezwiązanym z Javą (pisanym najczęściej w Perlu), jest technologia CGI (od ang. *Common Gateway Interface*).
- Umieszczanie kodu HTML w wywołaniach metody `println()` jest niewygodne i może być źródłem wielu błędów — rozwiązaniem tego problemu jest technologia JSP, która umożliwia umieszczanie kodu języka Java w ramach stron HTML zamiast osadzania HTML-a w kodzie języka programowania Java.

2. Bardziej szczegółowy przegląd zagadnień

Architektura aplikacji internetowej



Serwlety potrzebują pomocy. Kiedy żądanie HTTP dociera do serwera WWW, ktoś musi jeszcze stworzyć egzemplarz serwletu lub przynajmniej utworzyć nowy wątek obsługujący to żądanie. Ktoś musi przecież wywołać metodę `doPost()` lub `doGet()` serwletu. Warto też pamiętać o otrzymywaniu przez wspomniane metody dwóch kluczowych argumentów — obiektów reprezentujących żądanie i odpowiedź protokołu HTTP. Ktoś te obiekty musi oczywiście przekazać do serwletu. Ktoś musi zarządzać życiem, śmiercią i zasobami serwletu. Tym kimś jest kontener (ang. *container*). W tym rozdziale przyjrzymy się zasadom funkcjonowania aplikacji internetowych w ramach kontenerów i rzucimy okiem na strukturę aplikacji internetowych budowanych z wykorzystaniem wzorca projektowego MVC (ang. *Model View Controller*).



Wysokopoziomowa architektura aplikacji internetowych

- 1.1.** Dla każdej z metod przesyłania żądań protokołu HTTP (takich jak GET, POST, HEAD itp.) opisz jej przeznaczenie i charakterystyki techniczne, wymień czynniki, które mogą decydować o wyborze danej metody przez klienta (zazwyczaj przeglądarkę internetową), oraz zidentyfikuj metodę `HttpServlet`, która odpowiada danej metodzie protokołu HTTP.
- 1.4.** Opisz znaczenie i sekwencję zdarzeń składających się na cykl życia serwletu: (1) załadowanie klasy serwletu, (2) konkretyzacja serwletu, (3) wywołanie metody `init()`, (4) wywołanie metody `service()` oraz (5) wywołanie metody `destroy()`.
- 2.1.** Skonstruuj strukturę plików i katalogów dla aplikacji internetowej zawierającej (a) statyczną treść, (b) strony JSP, (c) klasy serwletów, (d) deskryptor wdrożenia, (e) biblioteki znaczników, (f) pliki JAR oraz (g) pliki klas Javy. Opisz techniki ochrony plików zasobów przed dostępem za pośrednictwem protokołu HTTP.
- 2.2.** Opisz przeznaczenie i semantykę każdego z wymienionych dalej elementów deskryptora wdrożenia: egzemplarz serwletu, nazwa serwletu, klasa serwletu, parametry inicjalizacji serwletu, adres URL nazwanego odwzorowania serwletu.

Uwagi wyjaśniające:

Wszystkie cele wymienione w tym podrozdziale zostaną dogłębnie przeanalizowane w pozostałych rozdziałach, zatem niniejszy rozdział należy traktować jak pierwsze spojrzenie na podstawy tego, czym szczegółowo zajmiemy się w kolejnych częściach. Innymi słowy, nie powinienes się przejmować, jeśli po przeczytaniu tego rozdziału nie będziesz potrafił odpowiedzieć na postawione tutaj pytania (lub jeśli nie będziesz tych pytań nawet pamiętał).

Na końcu tego rozdziału nie będziemy analizowali żadnych przykładowych pytań egzaminacyjnych, ponieważ ich analiza będzie możliwa dopiero po zapoznaniu się z bardziej szczegółowym materiałem zawartym w kolejnych rozdziałach.

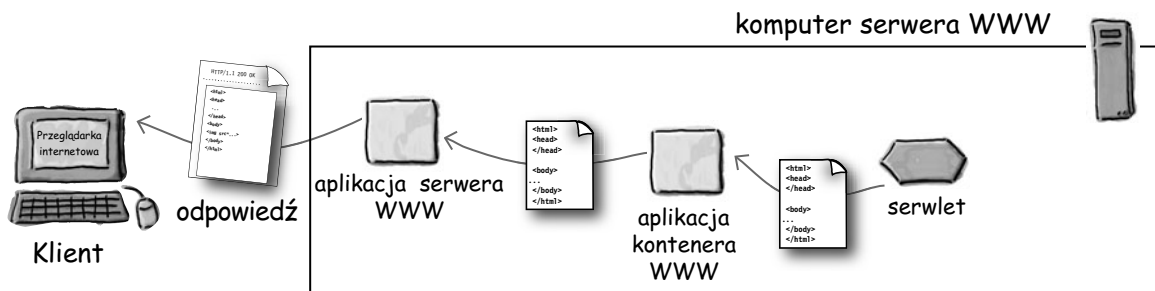
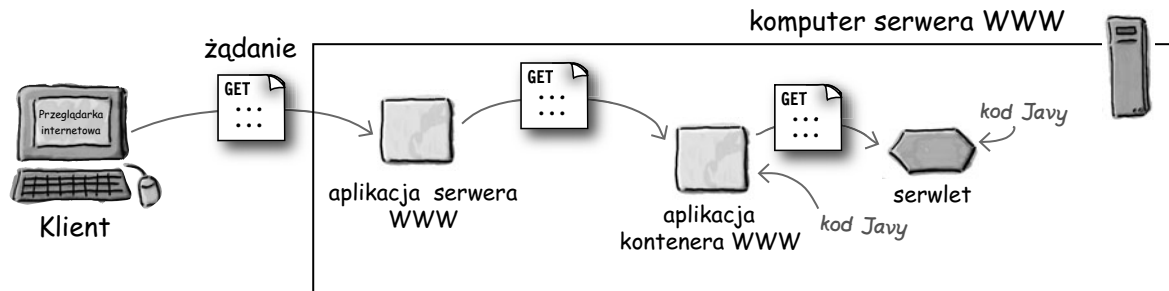
Ciesz się z prostego, wprowadzającego materiału, póki możesz!

PAMIĘTAJ JEDNAK, że prezentowane w tym rozdziale zagadnienia będą nam potrzebne w kolejnych rozdziałach. Jeśli masz już pewne doświadczenie w kwestii serwletów, prawdopodobnie możesz ten rozdział jedynie przekartkować (najlepiej analizując tylko najważniejsze rysunki i wykonując ćwiczenia) i przejść do rozdziału 3.

Czym jest kontener?

Serwlety nie zawierają metody `main()`. Serwlety są kontrolowane przez inną aplikację Javy nazywaną kontenerem.

Przykładem kontenera (ang. *container*) jest Tomcat. Kiedy nasza aplikacja serwera WWW (np. Apache) otrzymuje żądanie dotyczące *serwletu* (w przeciwieństwie np. do przestarzałej, płaskiej, statycznej strony HTML), serwer nie przekazuje tego żądania bezpośrednio do wskazywanego serwletu, tylko do kontenera, w którym ten serwlet wcześniej *umieszczono*. Od tej pory to kontener odpowiada za przekazanie do serwletu obiektów żądania i odpowiedzi protokołu HTTP oraz za wywołanie właściwych metod serwletu (takich jak `doPost()` lub `doGet()`).



Co zrobić, jeśli mam odpowiedni kod Javy, ale nie mam ani serwletów, ani kontenerów?

Jakie rozwiązanie należy zastosować w sytuacji, gdy musimy napisać program Javy, który ma obsługiwać dynamiczne żądania przychodzące do aplikacji serwera WWW (np. Apache'a), ale bez dodatkowego kontenera (takiego jak wspomniany już Tomcat)? Innymi słowy, wyobraźmy sobie, że nie istnieje pojęcie serwletu i że dysponujemy jedynie standardowymi bibliotekami J2SE. W takim przypadku należy oczywiście przyjąć, że mamy możliwość takiego skonfigurowania aplikacji serwera WWW, by wywoływał naszą aplikację napisaną w Javie. Stosowanie takiego rozwiązania byłoby uzasadnione, gdyby nie była nam znana koncepcja kontenera. Wystarczy sobie wyobrazić sytuację, w której musimy stworzyć aplikację internetową, dysponując tylko klasyczną Javą.

Prawdziwy wojownik nigdy nie ucieka się do stosowania kontenerów. Taki wojownik napisałby wszystko gołymi rękami, korzystając wyłącznie z klas J2SE.



Wymień kilka funkcji, które musielibyśmy zaimplementować w aplikacji J2SE, gdybyśmy nie mogli korzystać z aplikacji kontenera WWW:

** Stwórz gniazdo połączenia z serwerem i odpowiedni obiekt nastuchujący (odbiornik) dla nowego gniazda.*

Możliwe odpowiedzi: stwórz obiekt zarządzający wątkami, zaimplementuj techniki zabezpieczeń, coś do filtrowania takich elementów jak zapisy dziennika, obsługa JSP (o Boże!), zarządzanie pamięcią...

Co daje nam kontener?

Wiemy już, że to właśnie kontener uruchamia serwlety i nimi zarządza, ale właściwie *dlaczego* tak się dzieje? Czy warto wprowadzać dodatkową warstwę i godzić się na związane z tym opóźnienia?

Obsługa komunikacji Kontener zapewnia prosty mechanizm komunikacji pomiędzy naszymi serwletami a serwerem WWW. Dzięki kontenerom nie musimy budować gniazd serwera, nasłuchiwać komunikacji na porcie, tworzyć strumieni itp. Kontener zna odpowiedni protokół porozumiewania się z serwerem WWW, zatem nasze serwlety nie muszą się martwić o zapewnienie interfejsu API pomiędzy np. serwerem Apache a kodem naszej aplikacji internetowej. W tej sytuacji programista aplikacji internetowej musi jedynie zadbać o logikę biznesową umieszczoną w serwlecie (odpowiedzialną na przykład za przyjmowanie i przetwarzanie zamówień składanych w sklepie internetowym).

Zarządzanie cyklem życia Kontener jest panem życia i śmierci naszych serwletów. Właśnie kontener odpowiada za wczytywanie niezbędnych klas, tworzenie egzemplarzy i inicjalizację serwletów, wywoływanie ich metod oraz przystosowywanie serwletów do współpracy z mechanizmami odśmiecania pamięci. Dzięki kontenerom *nie* musimy tracić naszego cennego czasu na zarządzanie zasobami.

Obsługa wielowątkowości Kontener automatycznie tworzy nowy wątek Javy dla każdego otrzymanego żądania dotyczącego serwletu. Kiedy serwlet kończy wykonywanie metody obsługującej przysłane przez klienta żądanie HTTP, odpowiedni wątek jest zamykany (zabijany). Nie oznacza to jednak, że programista jest zwolniony z obowiązku zapewniania bezpieczeństwa wątków — nadal musi pamiętać o ich właściwej synchronizacji. Z drugiej strony, samo przejęcie przez serwer odpowiedzialności za tworzenie i zarządzanie wątkami obsługującymi równocześnie wiele żądań pozwala mu zaoszczędzić mnóstwo pracy.

Bezpieczeństwo deklaratywne Korzystanie z kontenera wiąże się ze stosowaniem deskryptora wdrożenia (w formacie XML), który konfiguruje (i modyfikuje) mechanizmy zabezpieczeń bez konieczności trwałego umieszczania odpowiednich instrukcji w kodzie klasy serwletu (lub kodzie dowolnej innej klasy Javy). Aż trudno w to uwierzyć! Możesz zarządzać i wprowadzać zmiany w zabezpieczeniach bez konieczności modyfikowania i ponownego kompilowania swojego kodu źródłowego Javy.

Obsługa JSP Wiesz już, jak wygodne są strony JSP. Kto Twoim zdaniem zajmuje się tłumaczeniem kodu JSP do postaci właściwego kodu języka programowania Java? To oczywiście — *kontener*.

Dzięki kontenerowi **MOŻESZ** się w większym stopniu skoncentrować na własnej logice biznesowej, zamiast martwić się pisanem kodu zarządzającego wątkami, zapewniającego bezpieczeństwo oraz obsługującego komunikację sieciową.

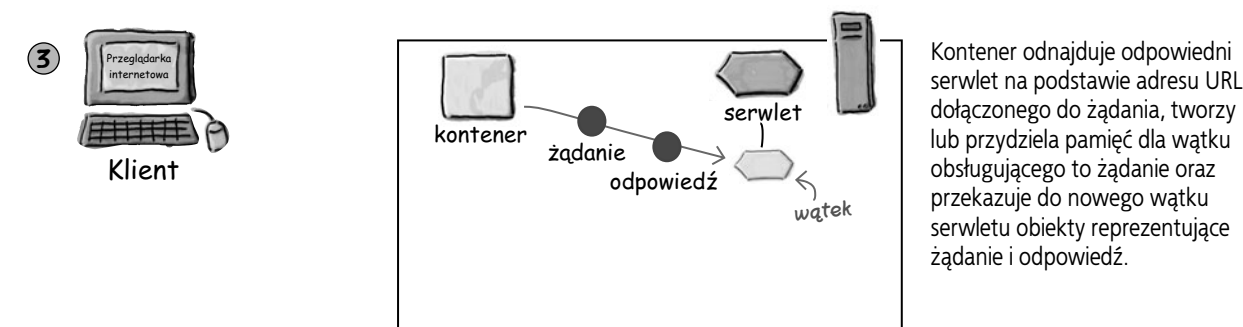
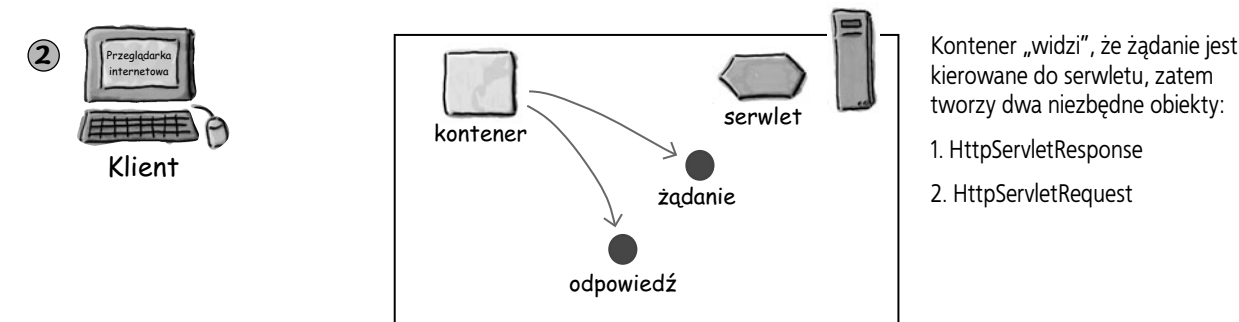
Możesz skupić całą swoją energię na opracowaniu bajecznego sklepu internetowego z folią z bąbelkami i pozostawić usługi pracujące w tle (w tym zapewnianie bezpieczeństwa oraz przetwarzanie stron JSP) kontenerowi.

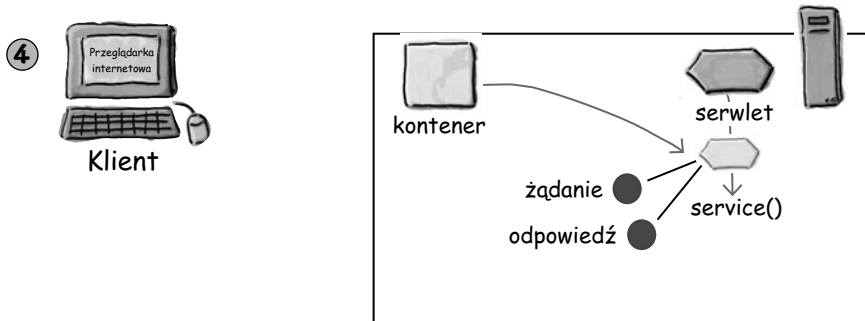
Teraz, zamiast pisania całego kodu działań realizowanych przez kontener, muszę się martwić wyłącznie o to, jak sprzedać moją folię z bąbelkami jej miłośnikom.



Jak kontener obsługuje żądanie HTTP?

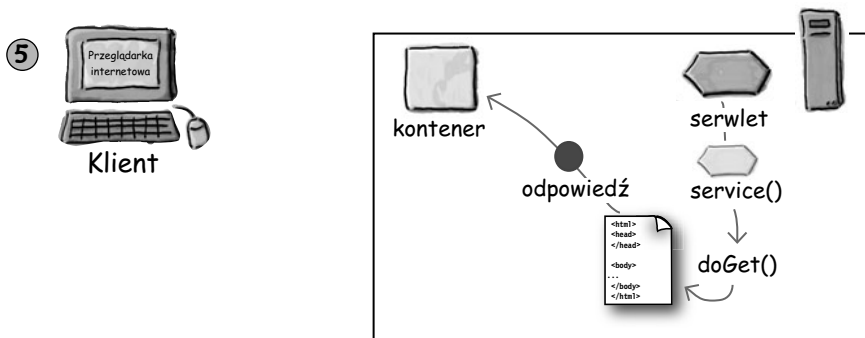
Co prawda najsmakowitsze kąski pozostawimy sobie na kolejne rozdziały tej książki, jednak warto już teraz dokonać krótkiego przeglądu sposobu funkcjonowania kontenerów:



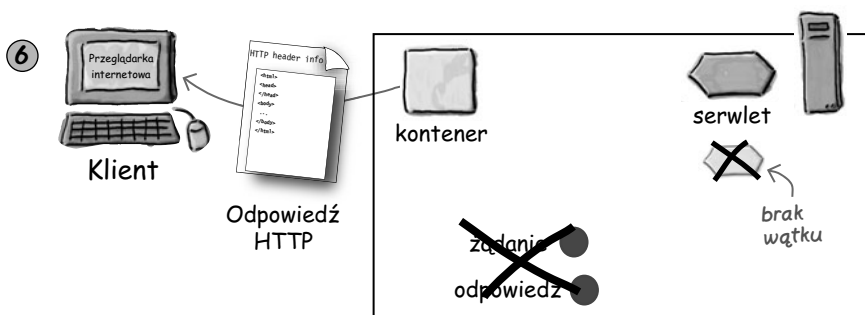


Kontener wywołuje metodę `service()` serwletu. W zależności od typu żądania metoda `service()` wywołuje albo metodę `doGet()`, albo metodę `doPost()`.

W tym przypadku zakładamy, że do kontenera dotarło żądanie protokołu HTTP typu GET.



Metoda `doGet()` generuje dynamiczną stronę HTML i umieszcza ją w obiekcie odpowiedzi. Pamiętaj, że kontener cały czas utrzymuje referencję do obiektu odpowiedzi!



Działanie wątku się kończy, kontener przekształca obiekt odpowiedzi w odpowiedź protokołu HTTP, odsyła tę odpowiedź do klienta, po czym usuwa obiekty żądania i odpowiedzi.

Jak to wszystko wygląda w kodzie (co sprawia, że serwlet jest serwletem)?

W świecie rzeczywistym 99,9% wszystkich serwletów nadpisuje metodę `doGet()` bądź metodę `doPost()`.

Zwróć uwagę na brak metody `main()`. Metody związane z cyklem życia serwletu (w tym `doGet()`) są wywoływane przez kontener.

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
```

99,9999% serwletów dziedziczy właśnie po klasie `HttpServlet`.

```
public class Ch2Servlet extends HttpServlet {
```

```
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
                      throws IOException {
```

W tym miejscu nasz serwlet otrzymuje referencje do utworzonych przez kontener obiektów reprezentujących żądanie i odpowiedź.

```
        PrintWriter out = response.getWriter();
        java.util.Date dzisiaj = new java.util.Date();
        out.println("<html> " +
                   "<body> " +
                   "<h1 style=\"text-align:center>" +
                   "Nasz drugi serwlet: Ch2Servlet</h1>" +
                   "<br>" + dzisiaj +
                   "</body>" +
                   "</html>");
```

Obiekt klasy `PrintWriter` możemy uzyskać za pośrednictwem obiektu odpowiedzi przekazanego naszemu serwletowi przez kontener. Obiekt `PrintWriter` służy do zapisywania w obiekcie odpowiedzi tekstu HTML (można też użyć innych obiektów umożliwiających zapisywanie zamiast tekstu HTML, np. obrazów).

```
    }
}
```

Nie ma niemądrych pytań

P: Pamiętam o metodach `doGet()` oraz `doPost()`, ale na poprzedniej stronie była przecież mowa o metodzie `service()`. Skąd się wzięła ta metoda?

O: Twój serwlet odziedziczył ją po klasie `HttpServlet`, która z kolei odziedziczyła tę metodę po klasie `GenericServlet`, która odziedziczyła ją po... zresztą, hierarchię klas opisemy wyczerpująco w rozdziale „Być serwletem”, więc musisz się wykazać odrobiną cierpliwości.

P: Do tej pory unikałeś wyjaśnienia, skąd kontener wie, gdzie szukać właściwego serwletu i jak adres URL jest odwzorowywany na odpowiedni serwlet. Czy użytkownik musi podawać kompletną ścieżkę i nazwę pliku klasy serwletu?

O: Dobre pytanie, ale odpowiedź brzmi „nie”. Poruszyłeś jednak bardzo istotny problem (odwzorowywania serwletu oraz wzorców URL), którym krótko zajmiemy się na kolejnych kilku stronach i który dogłębnie omówimy w rozdziale poświęconym wdrażaniu aplikacji internetowych.

Zastanawiasz się pewnie, jak kontener znajduje odpowiedni serwlet...

Adres URL, który dociera do serwera WWW jako część żądania klienta, jest w jakiś sposób *odwzorowywany* na odwołanie do konkretnego serwera. Takie odwzorowywanie adresów URL na serwlety może się odbywać na wiele różnych sposobów i jest jednym z kluczowych problemów, przed którymi stajemy w procesie wytwarzania aplikacji internetowych. Żądanie użytkownika musi zostać przekształcone w prawidłowe odwołanie do konkretnego serwletu — zrozumienie i (często) *konfiguracja* tego typu odwzorowań należy do programisty aplikacji internetowej. Co o tym myślisz?



WYTĘŻ UMYŚL

Jak kontener powinien odwzorowywać serwlety na adresy URL?

Kiedy użytkownik robi *coś* w swojej przeglądarce (klika łącze, naciska przycisk *Wyślij zapytanie*, wpisuje adres URL itp.), należy podjąć *jakiś* krok, który spowodują wysłanie żądania do *konkretnego* serwletu (lub innego zasobu aplikacji internetowej, np. strony JSP). Jak to działa w praktyce?

Wymień zalety i wady każdego z poniższych rozwiązań.

- ① *Zapisanie odwzorowania na stałe w kodzie strony HTML. Innymi słowy, klient wykorzystuje dokładną ścieżkę i nazwę pliku (klasy) serwletu:*

ZALETY:

WADY:

- ② *Wykorzystanie do odwzorowania narzędzia dostarczonego przez producenta kontenera:*

ZALETY:

WADY:

- ③ *Użycie swoistej tabeli właściwości reprezentującej odwzorowania:*

ZALETY:

WADY:

Serwlet może mieć aż TRZY nazwy

Serwlet musi oczywiście mieć *nazwę ścieżki* do pliku klasy (np. `classes/registration/SignUpServlet.class`), czyli ścieżkę do faktycznego pliku klasy. Twórca oryginalnej klasy serwletu wybiera jej nazwę (i nazwę pakietu, która definiuje odpowiednią część struktury katalogów), natomiast pełna nazwa ścieżki jest uzależniona od położenia pliku klasy na serwerze. Każda osoba wdrażająca serwlet na serwerze WWW może mu dodatkowo nadać specjalną *nazwę wdrożenia*. Nazwa wdrożenia jest po prostu *poufną nazwą wewnętrzną*, która nie musi być taka sama jak nazwa klasy lub nazwa pliku. Nazwa wdrożenia może co prawda odpowiadać nazwie klasy serwletu (`registration.SignUpServlet`) lub względnej ścieżce do pliku klasy (`classes/registration/SignUpServlet.class`), ale też może być zupełnie inna (np. `EnrollServlet`).

I wreszcie, serwlet ma *publiczną nazwę URL* — nazwę znaną klientowi. Publiczną nazwę URL koduje się w języku HTML, dzięki czemu w chwili kliknięcia przez użytkownika łącza wskazującego nasz serwlet można tę nazwę wysłać na serwer wraz z żądaniem HTTP.



Znana klientowi nazwa URL

Klient widzi adres URL serwletu (zakodowany w języku HTML), ale nie wie, jak ta nazwa serwletu zostanie w praktyce odwzorowana na rzeczywiste katalogi i pliki składowane na serwerze. Publiczna nazwa URL jest swoistą podróbką stworzoną specjalnie dla klientów.

Znana wdrożeniowcowi poufna nazwa wewnętrzna

Wdrożeniowiec może stworzyć nazwę, która będzie znana tylko jemu i innym użytkownikom pracującym w rzeczywistym środowisku operacyjnym. Także ta nazwa jest fałszywką stworzoną wyłącznie na potrzeby procesu wdrażania serwletu. Poufna nazwa wewnętrzna nie musi odpowiadać ani wykorzystywanej przez klienta publicznej nazwie URL, ani rzeczywistej nazwie pliku czy ścieżki do klasy serwletu.

Faktyczna nazwa pliku

Opracowana przez programistę *klasa* serwletu ma w pełni kwalifikowaną nazwę, która obejmuje zarówno nazwę klasy, jak i nazwę pakietu. *Plik* klasy serwletu ma oczywiście rzeczywistą ścieżkę i swoją nazwę (zależną od miejsca składowania struktury katalogów pakietu na serwerze WWW).

Czy to nie dziwne, że każdy może swobodnie wyrażać swoją kreatywność i definiować własną nazwę dla tego samego składnika aplikacji internetowej? W czym tkwi problem? Dlaczego nie możemy po prostu korzystać z jednej, prawdziwej i jednoznacznej nazwy pliku?



Odwzorowywanie nazw serwletów poprawia elastyczność i bezpieczeństwo naszych aplikacji internetowych

Przemyśl to.

Przyjmijmy, że trwale zakodowałeś rzeczywistą ścieżkę i nazwę pliku w swoich stronach JSP i pozostałych stronach HTTP korzystających z danego serwletu. Świetnie. Co w takim razie należałoby zrobić w razie konieczności reorganizacji aplikacji i — być może — przeniesienia niektórych jej składników do innej struktury katalogów? *Czy naprawdę chcesz wymusić na wszystkich użytkownikach swojego serwletu znajomość (i konieczność nieustannej weryfikacji) tej samej struktury katalogów?*

Jeśli zamiast zapisywania w kodzie źródłowym rzeczywistej nazwy pliku i ścieżki zastosujemy mechanizm odwzorowań, będziemy mogli swobodnie przenosić składniki aplikacji bez konieczności kosztownego czasochłonnego wyszukiwania i aktualizowania kodu klienta, który sztywno odwołuje się do starej lokalizacji plików serwletu.

A co z bezpieczeństwem? Czy naprawdę zależy nam na tym, aby klient dokładnie znał strukturę plików i katalogów w naszym serwerze? Czy chcemy, by na przykład próbował bezpośrednio przeglądać pliki serwletów bez kontroli zapewnianej przez właściwe strony lub formularze? Nie ma wątpliwości, że użytkownik końcowy dysponujący wiedzą o rzeczywistej ścieżce do pliku klasy serwletu będzie mógł tę ścieżkę wpisać w swojej przeglądarce, aby przynajmniej spróbować uzyskać bezpośredni dostęp do tego serwletu.

Odwzorowania adresów URL na serwlety za pośrednictwem deskryptora wdrożenia

Podczas wdrażania naszego serwletu w kontenerze WWW musimy opracować stosunkowo prosty dokument XML nazywany deskryptorem wdrożenia (ang. *Deployment Descriptor* — *DD*), który na potrzeby kontenera określa sposób, w jaki należy wykonywać nasze serwlety i (lub) strony JSP. Deskryptory wdrożenia nie służą co prawda wyłącznie odwzorowywaniu nazw, ale zawierają dwa elementy języka XML odpowiedzialne właśnie za definiowanie odwzorowań adresów URL na serwlety — jeden odwzorowuje znaną klientowi *publiczną nazwę URL* na naszą *nazwę wewnętrzną*, drugi odwzorowuje naszą *nazwę wewnętrzną* do postaci w pełni kwalifikowanej nazwy klasy.

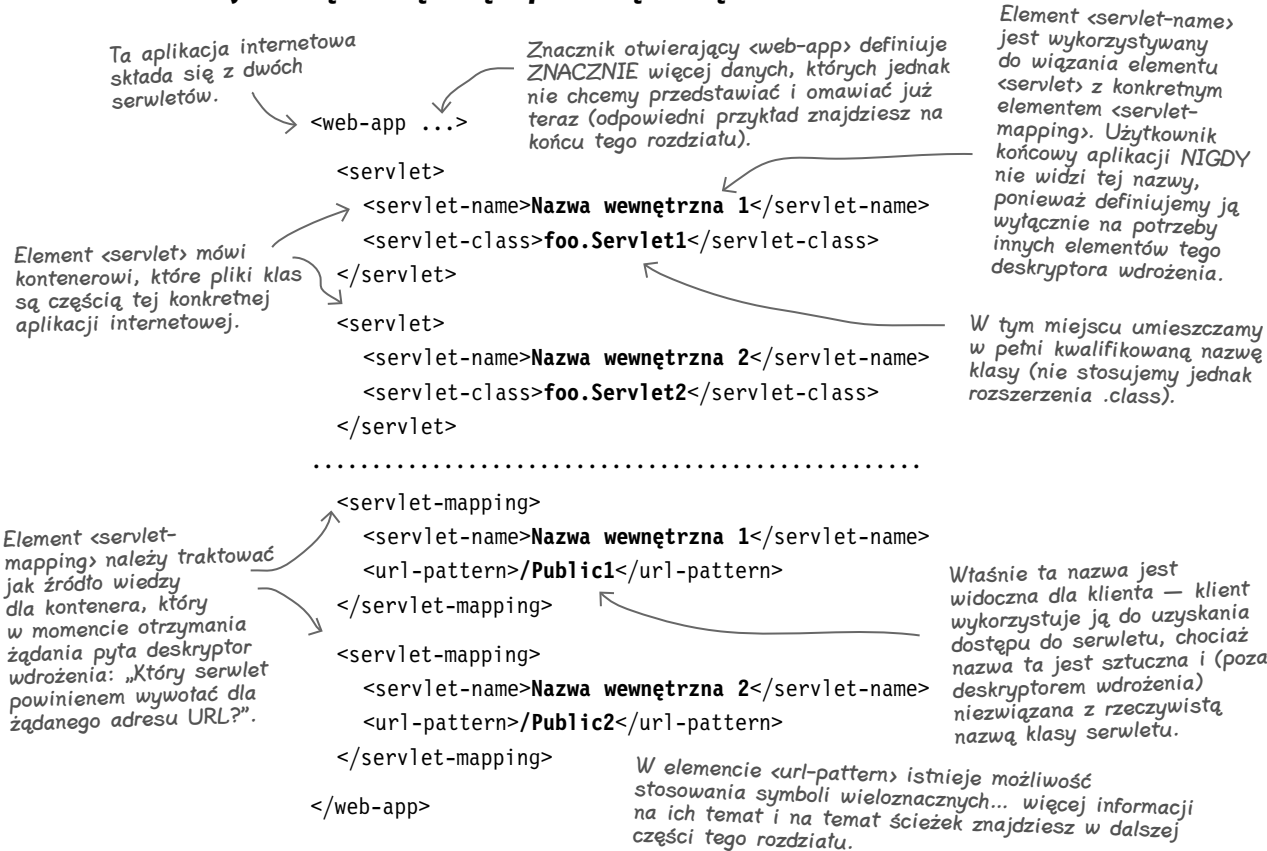
Dwa elementy deskryptora wdrożenia wykorzystywane do odwzorowywania adresów URL:

1 <servlet>

odwzorowuje nazwę wewnętrzną do postaci w pełni kwalifikowanej nazwy klasy

2 <servlet-mapping>

odwzorowuje nazwę wewnętrzną w publiczną nazwę URL



Zaczekaj! W deskrytorze wdrożenia możemy zrobić znacznie więcej

Poza definiowaniem odwzorowań adresów URL na rzeczywiste nazwy serwletów, deskryptory wdrożenia można wykorzystywać do dostosowywania do naszych potrzeb także innych aspektów aplikacji internetowej, jak role bezpieczeństwa, strony o błędach, biblioteki znaczników, informacje o konfiguracji początkowej czy wręcz (jeśli korzystamy z pełnowartościowego serwera J2EE) deklaracje dostępu do konkretnych komponentów Enterprise JavaBeans (EJB).

Nie przejmuj się na razie szczegółami. Póki co zasadnicze znaczenie ma dla nas możliwość deklaratywnego modyfikowania naszej aplikacji na poziomie deskryptora wdrożenia zamiast wprowadzania zmian w kodzie źródłowym (i ponownego kompilowania tego kodu)!

Pomyśl tylko... oznacza to, że nawet osoby niebędące programistami Javy mogą dostosowywać do swoich potrzeb napisane w tym języku aplikacje internetowe bez konieczności zwracania nam głowy i zakłócania wakacji w tropikach.

Nie ma
niemądrych pytań

P: Mam kłopot. W przedstawionym deskrytorze wdrożenia nadal nie widzę niczego, co mogłoby wskazywać na rzeczywistą nazwę ścieżki do serwletu! Deskryptor wspomina tylko o nazwie klasy. Nadal nie ma więc odpowiedzi na pytanie, w jaki sposób kontener wykorzystuje tę nazwę do odnajdywania konkretnych plików klas serwletów. Może istnieje gdzieś INNE odwzorowanie, które określa, że taka i taka nazwa klasy jest odwzorowywana na taki i taki plik w takiej i takiej lokalizacji?

U: Wiedziałem, że zwrócisz na to uwagę. Masz rację — w elemencie `<servlet-class>` w deskrytorze wdrożenia umieszczamy jedynie nazwę klasy (w pełni kwalifikowaną, a więc z nazwą pakietu). Wynika to z faktu, iż kontener dysponuje określonym miejscem, w którym odnajduje wszystkie te serwlety, dla których w deskrytorze wdrożenia zdefiniowano odpowiednie odwzorowania.

W praktyce kontener stosuje wyszukany zbiór reguł do odnajdywania pasujących do siebie adresów URL (pochodzących z żądań klienta) i rzeczywistych klas Javy (przechowywanych gdzieś w serwerze). Omówimy to zagadnienie bardziej szczegółowo w dalszej części tej książki (w rozdziale poświęconym wdrażaniu aplikacji internetowych). Na razie w zupełności wystarczy, jeśli zapamiętasz, że definiowanie tego typu odwzorowań nie wymaga żadnych dodatkowych czynności.

Deskryptor wdrożenia (DD) oferuje nam mechanizm „deklaratywnego” dostosowywania aplikacji internetowych do potrzeb użytkownika bez konieczności modyfikowania ich kodu źródłowego!

Zalety deskryptorów wdrożenia



- Minimalizują liczbę operacji na kodzie źródłowym, który przeszedł już niezbędne testy.
- Umożliwiają nam dostosowywanie możliwości naszej aplikacji, nawet jeśli *nie mamy* dostępu do kodu źródłowego.
- Dają możliwość przystosowywania naszej aplikacji do korzystania z różnych zasobów (np. baz danych) bez konieczności ponownego kompilowania i testowania kodu źródłowego.
- Ułatwiają zarządzanie dynamicznymi regułami bezpieczeństwa, w tym listami kontrolnymi i rolami bezpieczeństwa.
- Osobom niebędącym programistami umożliwiają modyfikowanie i wdrażanie naszych aplikacji internetowych w czasie, gdy *my* możemy się koncentrować na ciekawszych zajęciach (np. na doborze najlepszych ciuchów przed wyjazdem nad morze).

Opowiadanie: Bob buduje witrynę swatającą

Umawianie się na randki jest w dzisiejszych czasach dosyć trudne. Kto ma na to czas, skoro zawsze jest jakiś dysk do zdefragmentowania? Bob, który chce wejść w interes *dot.com* (założmy na chwilę, że rynek tego typu witryn nie jest jeszcze nasycony), wierzy, że stworzenie specjalnej witryny randkowej dla pasjonatów informatyki będzie jego przepustką do wielkiej kariery i ostatecznego porzucenia dotychczasowej roli komiksowego Dylberta.

Problem w tym, że Bob był menedżerem projektów informatycznych tak długo, że nie jest już zbyt biegły we współczesnych technikach inżynierii oprogramowania. Zna jednak kilka trudnych pojęć oraz niektóre konstrukcje Javy; poświęcił też trochę czasu na pobieżną lekturę materiałów o serwetach, zatem błyskawicznie opracował niezbędny projekt i przystąpił do kodowania...

Chcę stworzyć elastyczną witrynę randkową, na której informatycy będą mogli się spotykać i łączyć w pary. Ponieważ nie każdemu udaje się zainstalować Linuksa...



Bob przystąpił do tworzenia garści serwletów... po jednym dla każdej strony

Bob rozważał zastosowanie tylko jednego serwletu pełnego niezbędnych wyrażań warunkowych, ale ostatecznie zdecydował, że podział funkcjonalności pomiędzy wiele osobnych serwletów będzie rozwiązaniem właściwszym — każdy serwlet powinien odpowiadać za generowanie i obsługę pojedynczej strony (strony zapytania, strony zapisywania do serwisu, strony wyników wyszukiwania itp.).

Każdy serwlet obejmuje kompletną logikę biznesową niezbędną do zmodyfikowania lub odczytania zawartości bazy danych oraz zapisania w strumieniu odpowiedzi (a więc odesłania do klienta) odpowiedniego kodu HTML.

// polecenia importowania

```
public class DatingServlet extends HttpServlet {

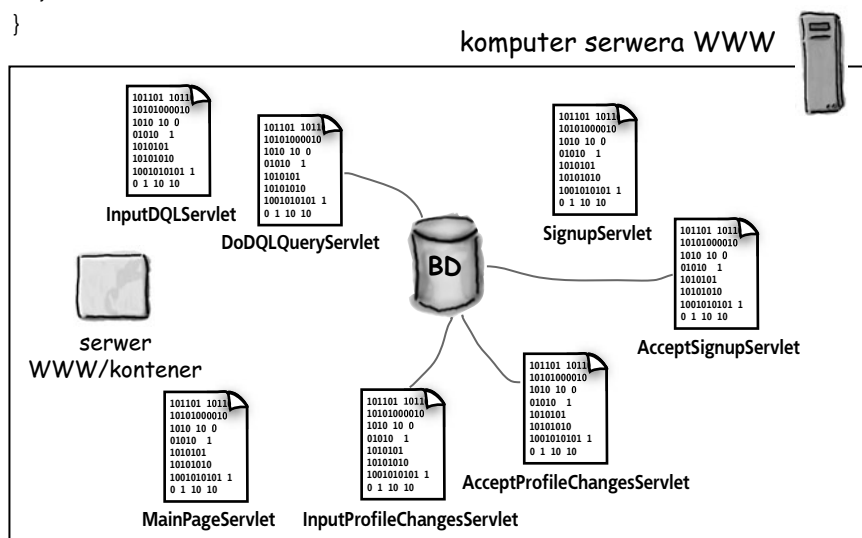
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
                      throws IOException {

        // w tym miejscu jest zakodowana logika biznesowa
        // bieżącego serwletu; wykonywany kod zależy od zakresu
        // działań danego serwletu (zapis w bazie danych,
        // wykonanie zapytania itp.)

        PrintWriter out = response.getWriter();

        // wygeneruj dynamiczną stronę HTML
        out.println("w tym miejscu znajduje się coś naprawdę okropnego");
    }
}
```

Nareszcie mam
prawdziwie obiektowy
projekt aplikacji. Każdy
z moich serwletów będzie
realizował dokładnie jedno
zadanie.



Serwlet podejmuje wszystkie działania niezbędne do przetworzenia otrzymanego żądania (jak wstawienie rekordu do bazy danych lub jej przeszukanie), po czym zwraca stronę HTML za pośrednictwem obiektu reprezentującego odpowiedź protokołu HTTP.

Cała logika biznesowa ORAZ odpowiedź ze stroną HTML dla klienta znajduje się wewnątrz kodu serwletu.

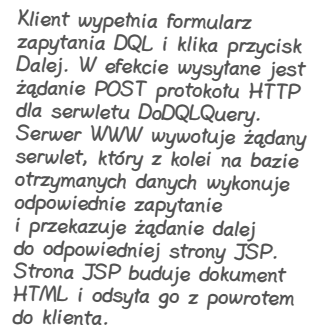
Ten projekt ze stronami JSP jest zdecydowanie lepszy. Kod serwletu jest teraz bardziej przejrzysty... każdy serwlet realizuje wyłącznie pewien wyinek logiki biznesowej, po czym wywołuje określoną stronę JSP, która obsługuje zadania związane z generowaniem dla klienta kodu HTML wysyłanego w odpowiedzi HTTP. Tym samym skutecznie oddzieliłem logikę biznesową od prezentacji.

```
public class DatingServlet extends HttpServlet {

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
                      throws IOException {

        // w tym miejscu jest zakodowana logika biznesowa
        // bieżącego serwletu; wykonywany kod zależy od zakresu
        // działań danego serwletu (zapis w bazie danych,
        // wykonanie zapytania itp.)

        // przekaż żądanie do konkretnej strony JSP
        // zamiast próbować samodzielnie zapisywać kod
        // HTML w strumieniu wyjściowym
    }
}
```



Ale dopiero wtedy kolega zapytał go: „STOSUJESZ wzorzec projektowy MVC, prawda?”

Kim chciał wiedzieć, czy usługa umawiania na randki będzie dostępna z poziomu aplikacji z graficznym interfejsem użytkownika (GUI) opartym na komponentach Swing.

Bob odrzekł: „No cóż, nie myślałem o tym”. Wówczas Kim odpowiedział:

„Nie przejmuj się, to żaden problem — pewnie zastosowałeś wzorzec MVC nieświadomie, więc będziemy mogli bez trudu stworzyć klienta Swing GUI, który uzyska dostęp do Twoich klas logiki biznesowej”.

Bob przełknął ślinę.

Kim odpowiedział: „Tylko nie mów, że... *nie* stosowałeś MVC?”.

Bob powiedział: „Cóż, oddzieliłem prezentację od logiki biznesowej...”.

Kim odrzekł: „To dopiero początek... ale, niech zgadnę... pewnie umieściłeś całą logikę biznesową w *serwletach*!?”.

Bob nagle uświadomił sobie, dlaczego na pewnym etapie swojej kariery zrezygnował z programowania na rzecz zarządzania.

Jest jednak na tyle zdeterminowany by doprowadzić swoją aplikację do właściwego końca, że poprosił Kima o błyskawiczny przegląd tajemniczego wzorca MVC.

Stosowanie wzorca projektowego MVC oznacza, że logika biznesowa jest nie tylko oddzielona od prezentacji... logika biznesowa nie powinna nawet wiedzieć, że ISTNIEJE jakaś prezentacja.

Istotą wzorca *MVC* (ang. *Model View Controller*) jest nie tylko oddzielenie logiki biznesowej od prezentacji, ale także umieszczenie czegoś *między* nimi, dzięki czemu możliwe będzie utrzymywanie logiki biznesowej w samodzielnej klasie Javy wielokrotnego użytku, która nie musi dysponować żadną wiedzą o działaniach widoku.

Bob był całkiem blisko, ponieważ sam wpadł na pomysł rozdzielenia logiki biznesowej i prezentacji, ale w jego aplikacji logika biznesowa pozostawała w *ścisłym związku* z widokiem. Innymi słowy, Bob *włączył logikę biznesową do serwletu* i — tym samym — wykluczył możliwość ponownego wykorzystania tego kodu z innym widokiem (np. aplikacją Swing GUI czy nawet aplikacją mobilną). Jego logika biznesowa utknęła w serwletach, a powinna się znaleźć w osobnych klasach Javy, których Bob mógłby używać w przyszłości.

Co zrobisz, kiedy zaistnieje potrzeba zastosowania dla Twojej usługi randkowej aplikacji z graficznym interfejsem użytkownika opartym na komponentach Swing; aplikacji, która będzie korzystała z tej samej logiki biznesowej?

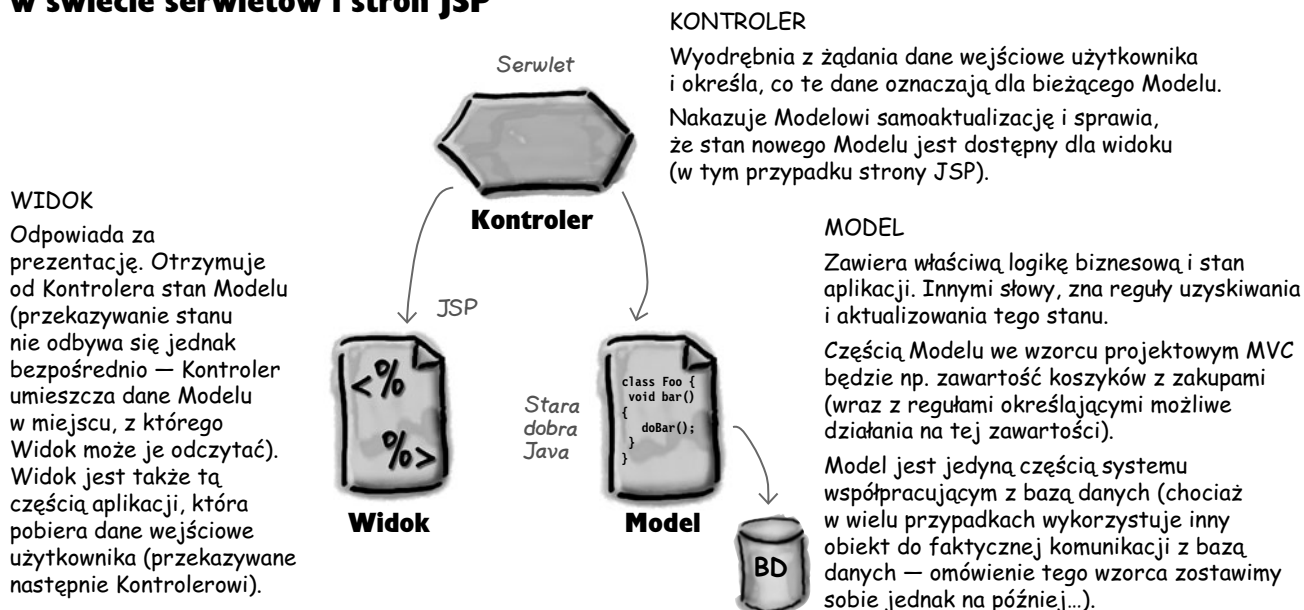


Rozwiązaniem zaistniałego problemu jest zastosowanie wzorca projektowego model-widok-kontroler (MVC)

Gdyby Bob znał i rozumiał wzorzec projektowy MVC, wiedziałby, że logika biznesowa nie powinna być osadzana w ramach serwletu bądź serwletów. Zdawałby sobie sprawę, że umieszczanie logiki biznesowej w kodzie serwletu powoduje ogromne komplikacje w razie konieczności uzyskania dostępu do serwisu randkowego z poziomu innej aplikacji (np. aplikacji z interfejsem GUI opartym na komponentach Swing). Wzorcowi projektowemu MVC (i innym wzorcom) poświęcimy co prawda znacznie więcej uwagi w dalszej części tej książki, jednak teraz powinieneś przejść krótki kurs na ten temat, ponieważ na końcu tego rozdziału zaprezentujemy przykładową aplikację zbudowaną właśnie w oparciu o wzorzec MVC.

Jeśli znasz już wzorzec projektowy MVC, zapewne wiesz, że wzorzec ten nie dotyczy wyłącznie serwletów i stron JSP — jednoznaczne oddzielenie logiki biznesowej od prezentacji jest równie istotne we wszelkiego rodzaju aplikacjach. Okazuje się jednak, że właśnie w przypadku aplikacji internetowych taki podział jest *szczególnie* istotny, ponieważ nie można zakładać, że nasza logika biznesowa będzie udostępniana *wyłącznie* za pośrednictwem stron WWW! Jesteśmy przekonani, że Twoje doświadczenie w pracy programisty w zupełności wystarczy, aby rozumieć, że jedynym pewnym aspektem procesu wytwarzania oprogramowania jest **stale modyfikowana specyfikacja**.

Wzorzec projektowy MVC w świecie serwletów i stron JSP

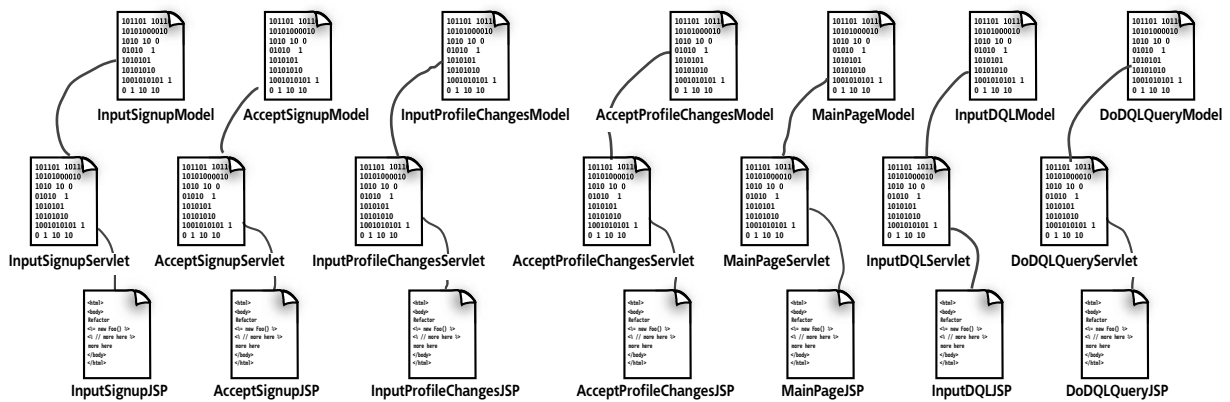


Stosowanie wzorca projektowego MVC dla aplikacji internetowej Boba

Bob wie już, co musi zrobić. Powinien oddzielić logikę biznesową od serwletów i dla każdego z nich stworzyć zwykłą klasę Javy — klasy te będą tworzyły Model.

Po wprowadzeniu tych zmian oryginalny serwlet będzie pełnił funkcję Kontrolera, nowa klasa logiki biznesowej będzie występowała w roli Modelu, a użyte strony JSP będą tworzyły Widok aplikacji.

Dla każdej strony swojej aplikacji Bob dysponuje teraz serwletem (Kontroler), klasą Javy (Model) oraz stroną JSP (Widok).



Jak myślisz: jestem
dobry, czy jeszcze lepszy?
Na tym właśnie polega
perfekcyjny projekt MVC.



Ale wtedy na całą aplikację rzucił okiem jego przyjaciel Kim

Kim odwiedził Boba, spojrzął na jego aplikację i stwierdził, że JEST to co prawda projekt MVC, ale jego architektura jest zupełnie nieprzemyślana. To fakt, logika biznesowa została wyodrębniona i umieszczona w Modelu, a serwlety pełnią funkcję Kontrolerów działających pomiędzy Modelami a Widokami, zatem Modele teoretycznie mogą nie mieć pojęcia o istnieniu Widoków. Taki podział jest oczywiście pożądany, warto jednak zwrócić uwagę na wszystkie te małe serwlety.

Co tak naprawdę te serwlety *robią*? Skoro logika biznesowa znajduje się już w bezpiecznej odległości od serwletów (Kontrolerów) — w osobnych klasach tworzących Model — same serwlety nie mają zbyt wiele do roboty poza ogólną obsługą aplikacji i (tak, pamiętam o tym) aktualizacją Modelu i wywoływaniem stron Widoku.

Zasadniczą wadą Twojego rozwiązania jest to, że ogólna logika aplikacji jest powielana w każdym z tych cholernych serwletów! Jeśli będziesz musiał zmienić choć jeden drobiazg, odpowiednie modyfikacje będziesz musiał wprowadzać we wszystkich serwletach. Konserwacja tego rodzaju aplikacji jest koszmarem.

„No tak, czułem, że z tym powielanym kodem coś jest nie tak” — odpowiedział Bob — „ale cóż innego mogłem w tej sytuacji zrobić? Nie chcesz chyba, żebym jeszcze raz umieszczał wszystko w jednym serwlecie? Czy *takie* rozwiązanie miałoby w ogóle sens?”.

W życiu nie widziałem tak beznadziejnego projektu! Spójrz na ten powielany kod we wszystkich serwletach. Musisz dodać jeden wspólny kod aplikacji (np. dotyczący bezpieczeństwa), który będzie wykorzystywany we wszystkich serwletach.

No nie... NAPRAWDĘ chcesz, żebym znowu umieścił wszystko w jednym serwlecie? A co z ideą programowania obiektowego?



Czy istnieje rozwiązanie?

Czy Bob powinien wrócić do jednego serwletu Kontrolera, aby uniknąć powtarzania kodu? Czy takie rozwiązanie będzie zgodne z koncepcją programowania obiektowego — przecież każdy ze stosowanych do tej pory serwletów w praktyce robi coś innego? Czy Keanu Reeves rzeczywiście zna Kung Fu?



WYTEŻ UMYŚŁ

Zostaw ten problem na później — my też tak zrobimy.

Jak sądzisz? Czy znasz odpowiedź? Czy w ogóle ISTNIEJE jedna odpowiedź? Czy zgodnie z propozycją Boba pozostawiłbyś istniejące serwlety, czy może umieściłbyś ten kod w jednym serwlecie Kontrolera? A gdybyś faktycznie użył jednego Kontrolera dla wszystkich działań, skąd taki Kontroler wiedziałby, który Model i Widok wywoływać?

To pytanie pozostanie bez odpowiedzi niemal do samego końca tej książki, zatem nie trać na nie zbyt dużo czasu i przekazaj je wątkowi pracującemu w tle Twojego umysłu...





- 1 Jeśli zastosujemy wzorec projektowy MVC w świecie serwetów i stron JSP, każdy z trzech wykorzystywanych komponentów (strona JSP, klasa Javy oraz serwet) będzie odgrywał jedną z trzech ról MVC. Oznacz kółkami litery *M*, *V* i *C* w zależności od tego, jaką funkcję w ramach wzorca MVC odgrywa dany komponent. Dla każdego komponentu powinienś oznaczyć w ten sposób tylko jedną literę.



JSP

M
V
C



klasa Javy
niebędąca
serwetem

M
V
C



serwet

M
V
C

- 2 Które elementy wzorca projektowego MVC reprezentują poszczególne litery składające się na akronim MVC?

M oznacza _____

V oznacza _____

C oznacza _____

KLUCZOWE ZAGADNIENIA



- Kontener zapewnia naszej aplikacji internetowej obsługę komunikacji, zarządzanie cyklem życia, obsługę przetwarzania wielowątkowego, bezpieczeństwo deklaratywne oraz pełną obsługę stron JSP — dzięki temu możemy się skoncentrować na tworzeniu logiki biznesowej tej aplikacji.
- Kontener tworzy obiekty żądania i odpowiedzi, które mogą być wykorzystywane przez serwlety (i inne składniki aplikacji internetowej) do uzyskiwania niezbędnych informacji o żądaniu i odsyłania odpowiednich danych do klienta.
- Typowy serwet jest klasą dziedziczącą po klasie `HttpServlet` (rozszerzającą klasę `HttpServlet`) i nadpisującą przynajmniej jedną z metod odpowiadających metodom protokołów HTTP wywoływanym przez przeglądarkę internetową (`doGet()`, `doPost()` itp.).
- Wdrożeniowiec może odwzorować klasę serwetu na odpowiedni adres URL, który będzie dostępny dla klientów zainteresowanych wysyłaniem żądań do danego serwetu. Nazwa użyta w adresie URL może nie mieć nic wspólnego z faktyczną nazwą *pliku* klasy.



Kto za co odpowiada?

Wypełnij poniższą tabelę. Określ, czy serwer WWW, kontener, a może serwlet odpowiada w największym stopniu za realizację wymienionych zadań. W kilku przypadkach prawidłowa może być więcej niż jedna odpowiedź. Jako zadanie dodatkowe umieść w tabeli krótkie komentarze opisujące poszczególne procesy.

Zadanie	Serwer WWW	Kontener	Serwlet
Tworzenie obiektów żądania i odpowiedzi			
Wywoływanie metody service()			
Uruchamianie nowych wątków obsługujących przychodzące żądania			
Konwersja obiektu odpowiedzi do postaci odpowiedzi protokołu HTTP			
Znajomość protokołu HTTP			
Dodawanie kodu HTML do obiektu odpowiedzi			
Utrzymywanie referencji do obiektów odpowiedzi			
Odnajdywanie adresów URL w deskryptorze wdrożenia			
Usuwanie obiektów żądania i odpowiedzi			
Koordynowanie tworzenia dynamicznej zawartości stron			
Zarządzanie cyklami życia			
Posiadanie właściwej dla elementu <servlet-class> nazwy z deskryptora wdrożenia			

Ćwiczenie z serwletów i deskryptorów wdrożenia



Ćwiczenie



Magnesiki metod

Fragmenty kodu działającego serwletu i odpowiadającego mu deskryptora wdrożenia wymieszano na drzwiach lodówki. Czy potrafisz prawidłowo połączyć te fragmenty, aby z powrotem otrzymać właściwe listingi serwletu i deskryptora wdrożenia, których adres URL kończy się słowem **/Dice**? Pamiętaj, że na lodówce mogą się znajdować dodatkowe magnesy, których w ogóle nie będziesz potrzebował.

Serwlet

```
public class
```

```
extends HttpServlet {
```

```
public void doGet(
```

```
throws IOException {
```

```
String d1 = Integer.toString((int)((Math.random()*6)+1));
String d2 = Integer.toString((int)((Math.random()*6)+1));

out.println("<html> <body> " +
    "<h1 align=center>Serwlet rzucający kostkami do gry: Ch2Dice</h1>" +
    "<p>Wyrzucono liczby " + d1 + " i " + d2 +
    "</body> </html>");
}
```

Deskryptor wdrożenia

```
<web-app ... >
```

(Pamiętaj, że nie jest to kompletny znacznik otwierający `<web-app>`; prawidłowy przykład tego znacznika przedstawimy na końcu tego rozdziału. Nie ma to oczywiście wpływu na rozwiązanie tego ćwiczenia.)

```
C2dice </servlet-name>
```

```
</web-app>
```

Magnesiki z kodem, kontynuacja...

</url-pattern>

public void service(

C2dice

Ch2Dice

<servlet-name>

ServletRequest request,

PrintWriter out = response.getWriter();

HttpServletResponse response)

<servlet-mapping>

C2dice

ServletResponse response,

<servlet-name>

/Dice

</servlet-class>

Ch2Dice

HttpServletRequest request,

<servlet>

PrintWriter out = request.getWriter();

/Dice

</servlet-name>

<url-pattern>

Ch2Dice

</servlet>

</servlet-mapping>

<servlet-class>

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
```



Zadanie	Serwer WWW	Kontener	Serwlet
Tworzenie obiektów żądania i odpowiedzi		Bezpośrednio przed uruchomieniem wątku.	
Wywoływanie metody service()		Metoda service() wywołuje następnie metodę doGet() lub doPost().	
Uruchamianie nowych wątków obsługujących przychodzące żądania		Uruchamia wątek serwletu.	
Konwersja obiektu odpowiedzi do postaci odpowiedzi protokołu HTTP		Generuje strumień odpowiedzi HTTP na podstawie danych zawartych w obiekcie odpowiedzi.	
Znajomość protokołu HTTP	Wykorzystuje protokół HTTP do komunikacji z przeglądarką klienta.		
Dodawanie kodu HTML do obiektu odpowiedzi			Przeznaczona dla klienta dynamiczna treść strony.
Utrzymywanie referencji do obiektów odpowiedzi		Przekazuje referencję serwletowi.	Wykorzystuje referencję do przekazania odpowiedzi.
Odnajdywanie adresów URL w deskrytorze wdrożenia		W ten sposób odnajduje serwlet właściwy dla otrzymanego żądania.	
Usuwanie obiektów żądania i odpowiedzi		Po zakończeniu pracy przez serwlet.	
Koordinowanie tworzenia dynamicznej zawartości stron	Wie, jak przekazywać odpowiednie dane do kontenera.	Wie, który serwlet należy wywołać.	
Zarządzanie cyklami życia		Wywołuje metodę service() (i, jak się przekonamy, nie tylko).	
Przechowywanie właściwej dla elementu <servlet-class> z deskryptora wdrożenia			public class Cokolwiek

Servlet

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
```

```
public class Ch2Dice extends HttpServlet {
```

```
    public void doGet( HttpServletRequest request,
```

```
                        HttpServletResponse response)
```

```
    throws IOException {
```

```
        PrintWriter out = response.getWriter();
```

```
        String d1 = Integer.toString((int)((Math.random()*6)+1));
        String d2 = Integer.toString((int)((Math.random()*6)+1));

        out.println("<html> <body> " +
            "<h1 align=center>Servlet rzucający kostkami do gry: Ch2Dice</h1> " +
            "<p>Wyrzucono liczby " + d1 + " i " + d2 +
            "</body> </html>");
    }
}
```

Deskryptor wdrożenia

```
<web-app ... >
```

```
    <servlet>
```

```
        <servlet-name> C2dice </servlet-name>
```

```
        Ch2Dice </servlet-class>
```

```
    <servlet-class>
```

```
</servlet>
```

```
    <servlet-mapping>
```

```
        C2dice </servlet-name>
```

```
        <servlet-name>
```

```
        <url-pattern> /Dice </url-pattern>
```

```
    </servlet-mapping>
```

```
</web-app>
```

„Działający” deskryptor wdrożenia (DD)

Na razie nie musisz się martwić o rzeczywiste znaczenia poszczególnych fragmentów deskryptora wdrożenia (w *dalszych* rozdziałach zdobędziesz odpowiednią wiedzę i zostaniesz z tej wiedzy rozliczony). W tym miejscu chcemy Ci tylko pokazać deskryptor wdrożenia *web.xml*, który faktycznie *działa*. W pozostałych przykładach prezentowanych w tym rozdziale pominiemy sporo fragmentów otwierającego znacznika `<web-app>`. (Teraz już wiesz, dlaczego nie rezygnowaliśmy z jego każdorazowego powielania).

Sposób, w jaki często przedstawiamy deskryptory wdrożenia w tej książce:

```
<web-app ...>
<servlet>
  <servlet-name>Ch3 Piwo</servlet-name>
  <servlet-class>com.example.web.WyborPiwa</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>Ch3 Piwo</servlet-name>
  <url-pattern>/WybierzPiwo.do</url-pattern>
</servlet-mapping>
</web-app>
```

Użyty tutaj otwierający znacznik `<web-app>` jest niekompletny.

RZECZYWISTA postać deskryptora wdrożenia:

```
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
  version="2.4">
  <servlet>
    <servlet-name>Ch3 Piwo</servlet-name>
    <servlet-class>com.example.web.WyborPiwa</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>Ch3 Piwo</servlet-name>
    <url-pattern>/WybierzPiwo.do</url-pattern>
  </servlet-mapping>
</web-app>
```

Żadnego z użytych w tym miejscu znaczników otwierających NIE musisz zapamiętywać. Jeśli używasz kontenera, który jest zgodny ze specyfikacją serwetów w wersji 2.4 (tak jest np. w przypadku serwera Tomcat 5), wystarczy te znaczniki kopiować i wklejać do deskryptorów.

Jaka w tym wszystkim jest rola platformy J2EE?

Java 2 Enterprise Edition (w skrócie J2EE) jest pewnego rodzaju superspecyfikacją, ponieważ obejmuje wiele innych specyfikacji, włącznie ze specyfikacją serwetów w wersji 2.4 oraz specyfikacją stron JSP w wersji 2.0 (obie obowiązują kontenery WWW). Warto jednak pamiętać, że platforma J2EE 1.4 obejmuje także specyfikację komponentów Enterprise JavaBeans (w skrócie EJB) w wersji 2.1, która z kolei obowiązuje twórców kontenerów EJB. Innymi słowy, kontener WWW ma na celu obsługę komponentów WWW (serwetów i stron JSP), natomiast zadaniem kontenera EJB jest obsługa komponentów *biznesowych*.

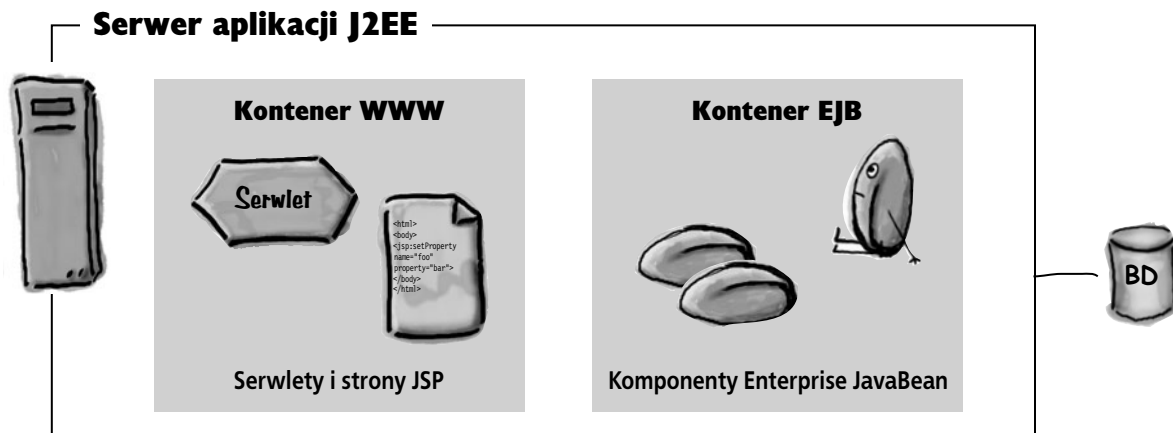
Serwer aplikacji w pełni zgodny ze specyfikacją J2EE musi zawierać zarówno kontener WWW, jak i kontener EJB (a także inne mechanizmy, jak implementacja standardów JNDI i JMS). Warto pamiętać, że np. Tomcat jest jedynie kontenerem WWW! Jego zgodność ze specyfikacją J2EE dotyczy wyłącznie elementów składających się na kontener WWW.

Tomcat jest kontenerem WWW, a nie kompletnym serwerem aplikacji J2EE, ponieważ nie oferuje funkcjonalności kontenera EJB.

Serwer aplikacji J2EE zawiera zarówno kontener WWW, JAK I kontener EJB.

Tomcat jest co prawda kontenerem WWW, ale NIE jest kompletnym serwerem aplikacji J2EE.

Specyfikacja J2EE 1.4 obejmuje specyfikację serwetów w wersji 2.4, specyfikację stron JSP w wersji 2.0 oraz specyfikację technologii EJB w wersji 2.1.



P: Czy fakt, iż Tomcat jest samodzielnym kontenerem WWW, oznacza, że istnieją samodzielne (autonomiczne) kontenery EJB?

U: W dawnych czasach (powiedzmy, że w roku 2000) istniały kompletne serwery aplikacji J2EE, samodzielne kontenery WWW oraz samodzielne kontenery EJB. Obecnie jednak niemal wszystkie kontenery EJB stanowią elementy składowe pełnowartościowych serwerów J2EE, choć istnieje jeszcze kilka autonomicznych kontenerów WWW (np. Tomcat i Resin).

Samodzielne kontenery WWW są zwykle konfigurowane w taki sposób, aby możliwa była ich współpraca z serwerami WWW wykorzystującymi protokół HTTP (np. z serwerem Apache), chociaż np. Tomcat *sam* oferuje możliwość funkcjonowania w roli prostego serwera HTTP. Warto jednak pamiętać, że funkcjonalność Tomcata w tym zakresie daleka jest od możliwości Apache'a, zatem większość aplikacji internetowych pozbawionych komponentów EJB wykorzystuje odpowiednio skonfigurowane serwery Apache i Tomcat — gdzie Apache pełni funkcję *serwera WWW*, natomiast Tomcat występuje w roli *kontenera*.

Do najbardziej popularnych serwerów J2EE należą: WebLogic firmy BEA, JBoss AS typu open source oraz WebSphere firmy IBM.

3. Omówienie MVC

Minipodręcznik MVC



Tworzenie i wdrażanie aplikacji internetowych MVC. Nadszedł czas, aby utrudzić nasze dłonie pisaniem formularzy HTML, kontrolerów serwetów, modelu (zwykłych, tradycyjnych klas Javy), deskryptora wdrożenia w formacie XML oraz widoku opartego na stronach JSP. Najwyższa pora zbudować, wdrożyć i przetestować taką aplikację. Najpierw jednak musimy przygotować odpowiednie środowisko *wytwarzania* aplikacji — stworzyć osobną (względem już wdrożonej i działającej aplikacji) strukturę katalogów projektu. Kolejnym krokiem jest skonfigurowanie środowiska *wdrażania* zgodnie ze specyfikacją serwetów i stron JSP oraz wymaganiami Tomcata. Dopiero po pomyślnym zrealizowaniu tych zadań będziemy gotowi do pisania, kompilowania, wdrażania i uruchamiania aplikacji. Budowana w tym rozdziale aplikacja internetowa jest co prawda bardzo mała, jednak w praktyce niemal NIE ma aplikacji, które byłyby za małe do stosowania wzorca projektowego MVC. Pamiętaj, że mała aplikacja dzisiaj, jutro może być dochodowym przedsięwzięciem *dot.com*.



Wdrażanie aplikacji internetowych

- 2.1.** Skonstruuj strukturę plików i katalogów dla aplikacji internetowej, która może zawierać: (a) statyczną treść, (b) strony JSP, (c) klasy serwetów, (d) deskryptor wdrożenia, (e) biblioteki znaczników, (f) pliki JAR oraz (g) pliki klas Javy. Opisz sposób, w jaki Twoim zdaniem można chronić pliki z zasobami przed dostępem za pośrednictwem protokołu HTTP.
- 2.2.** Opisz przeznaczenie i semantykę każdego z wymienionych elementów deskryptora wdrożenia: `error-page`, `init-param`, `mime-mapping`, `servlet`, `servlet-class`, `servlet-mapping`, `servlet-name` oraz `welcome-file`.
- 2.3.** Zbuduj prawidłową strukturę dla każdego z następujących elementów deskryptora wdrożenia: `error-page`, `init-param`, `mime-mapping`, `servlet`, `servlet-class`, `servlet-name` oraz `welcome-file`.

Wdrażanie aplikacji internetowych

Wszystkie wymienione obok cele zostaną dogłębnie omówione w rozdziale poświęconym wdrażaniu aplikacji internetowych — treść niniejszego rozdziału należy traktować jak pierwsze spojrzenie na tę tematykę. Ten rozdział jest kompletnym (a więc obejmującym wszystkie etapy prezentowanego procesu) podręcznikiem stosowania wzorca MVC, zatem jeśli go pominiesz, możesz mieć trudności podczas eksperymentów z przykładami prezentowanymi w późniejszych rozdziałach (gdzie do wielu szczegółów nie będziemy już wracać).

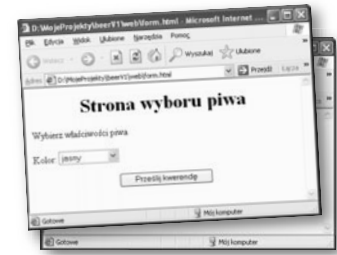
Podobnie jak w dwóch poprzednich rozdziałach, także tutaj nie musisz tracić czasu na wielokrotne powtarzanie i zapamiętywanie omawianego materiału — wystarczy, że przeczytasz ten rozdział z uwagą i zweryfikujesz proponowane przykłady w swoim środowisku.

Zbudujemy prawdziwą (małą) aplikację internetową

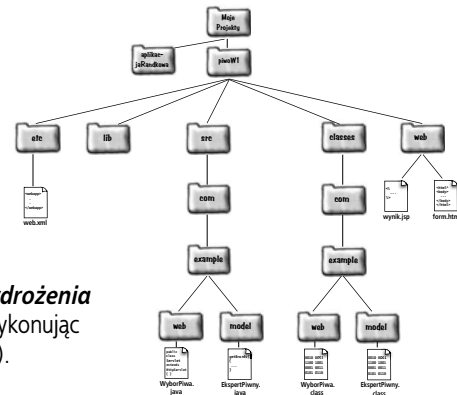
Przeanalizowaliśmy już rolę kontenera, wspominaliśmy o deskryptorach wdrożenia oraz rzuciliśmy okiem na architekturę MVC. Cały dotychczasowy materiał miał jednak charakter rozważań teoretycznych. Z drugiej strony, trudno sobie wyobrazić, by ktokolwiek chciał cały dzień tylko *czytać* o aplikacjach internetowych — najwyższy czas wreszcie coś *zrobić*.

Cztery kroki, które będziemy musieli wykonać:

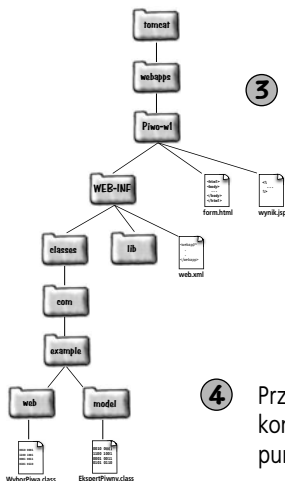
- 1 Dokonamy przeglądu **widoków (perspektyw) użytkownika** (czyli tego, co jest wyświetlane w oknie przeglądarki) oraz **architektury** wysokiego poziomu.



- 2 Stworzymy **środowisko wytwarzania** aplikacji, które będziemy wykorzystywali w pracach nad tym projektem (i którego będziesz mógł używać także w pozostałych przykładach omawianych w książce).

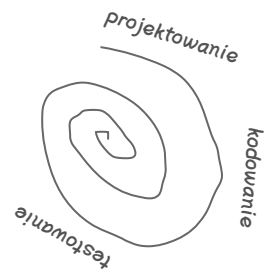


- 3 Stworzymy dla tego projektu **środowisko wdrożenia** (z którego także będziesz mógł korzystać, wykonując pozostałe przykłady omawiane w tej książce).



- 4 Przeprowadzimy **iteracyjny proces wytwarzania i testowania** rozmaitych komponentów składających się na naszą aplikację internetową (to fakt, ten punkt należy potraktować bardziej jak strategię niż tylko pojedynczy krok).

Uwaga: Zawsze zalecamy stosowanie strategii iteracyjnego wytwarzania i testowania aplikacji, choć nie wszędzie w tej książce będziemy analizowali wszystkie kroki tego procesu.

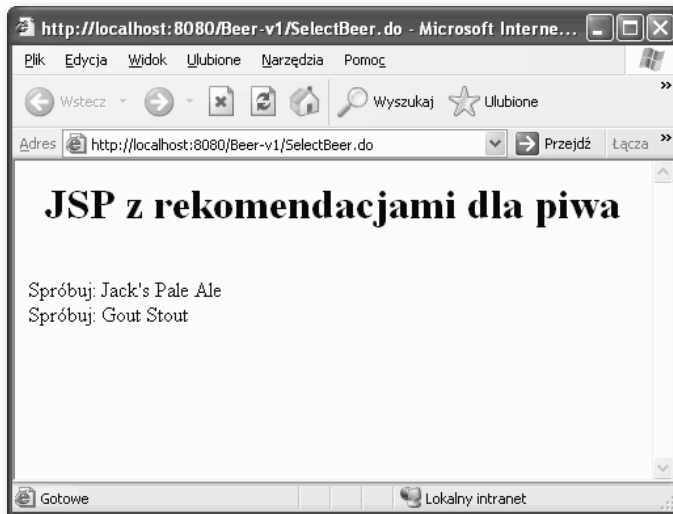


Widok użytkownika aplikacji internetowej — Doradca piwny

Naszą aplikację internetową nazwiemy *Doradca piwny*. Użytkownicy będą mogli przechodzić pomiędzy kolejnymi stronami naszej aplikacji, odpowiadając na pytania i uzyskiwać doskonałe porady na temat piwa.



Ta strona zostanie napisana w języku HTML i będzie generowała żądanie POST protokołu HTTP z określonym przez użytkownika kolorem piwa.



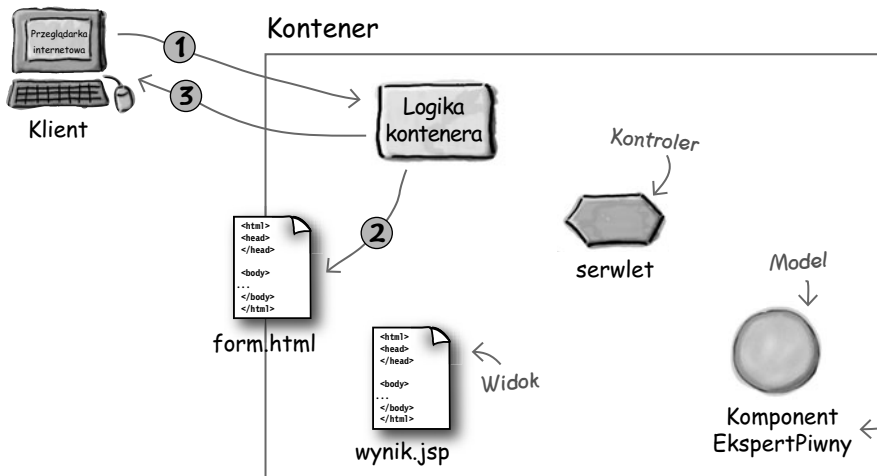
To będzie strona JSP z udzieloną użytkownikowi poradą odnośnie do marki piwa (według określonych przez niego preferencji w zakresie koloru).

P: Dlaczego piszemy aplikację internetową, która generuje dla użytkownika porady w sprawie piwa?

U: Po przeprowadzeniu wyczerpujących badań rynku, stwierdziliśmy, że 90% Czytelników naszych książek ceni sobie smak piwa. Pozostałym 10% Czytelników proponujemy zastąpienie słowa „piwo” słowem „kawa”.

Oto nasza architektura

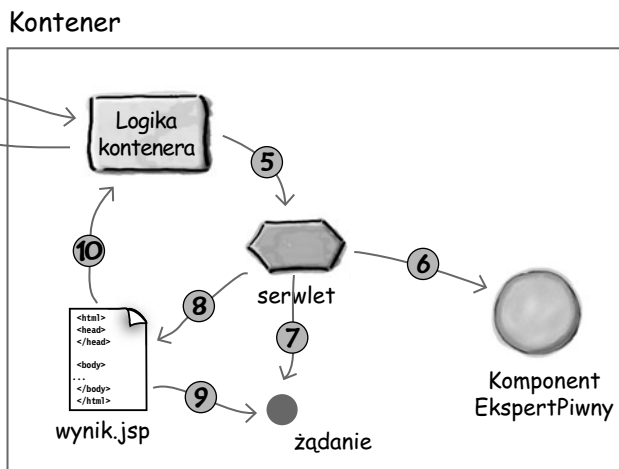
Chociaż prezentowana aplikacja jest bardzo skromna, zbudujemy ją zgodnie z prostą architekturą MVC. Dzięki temu, kiedy nasza aplikacja stanie się NAJPOPULARNIEJSZĄ witryną w internecie, będziemy mogli bez trudu rozszerzyć jej funkcjonalność.



1. Klient tworzy żądanie dotyczące strony *form.html*.
2. Kontener uzyskuje dostęp do strony *form.html*.
3. Kontener zwraca stronę do przeglądarki internetowej, gdzie użytkownik odpowiada na pytania zawarte w formularzu.

Zwyczajny, tradycyjny obiekt Javy

4. Przeglądarka przesyła do kontenera dane żądania.
5. Kontener odnajduje właściwy serwlet na podstawie określonego przez klienta adresu URL i przekazuje do tego serwletu otrzymane żądanie.

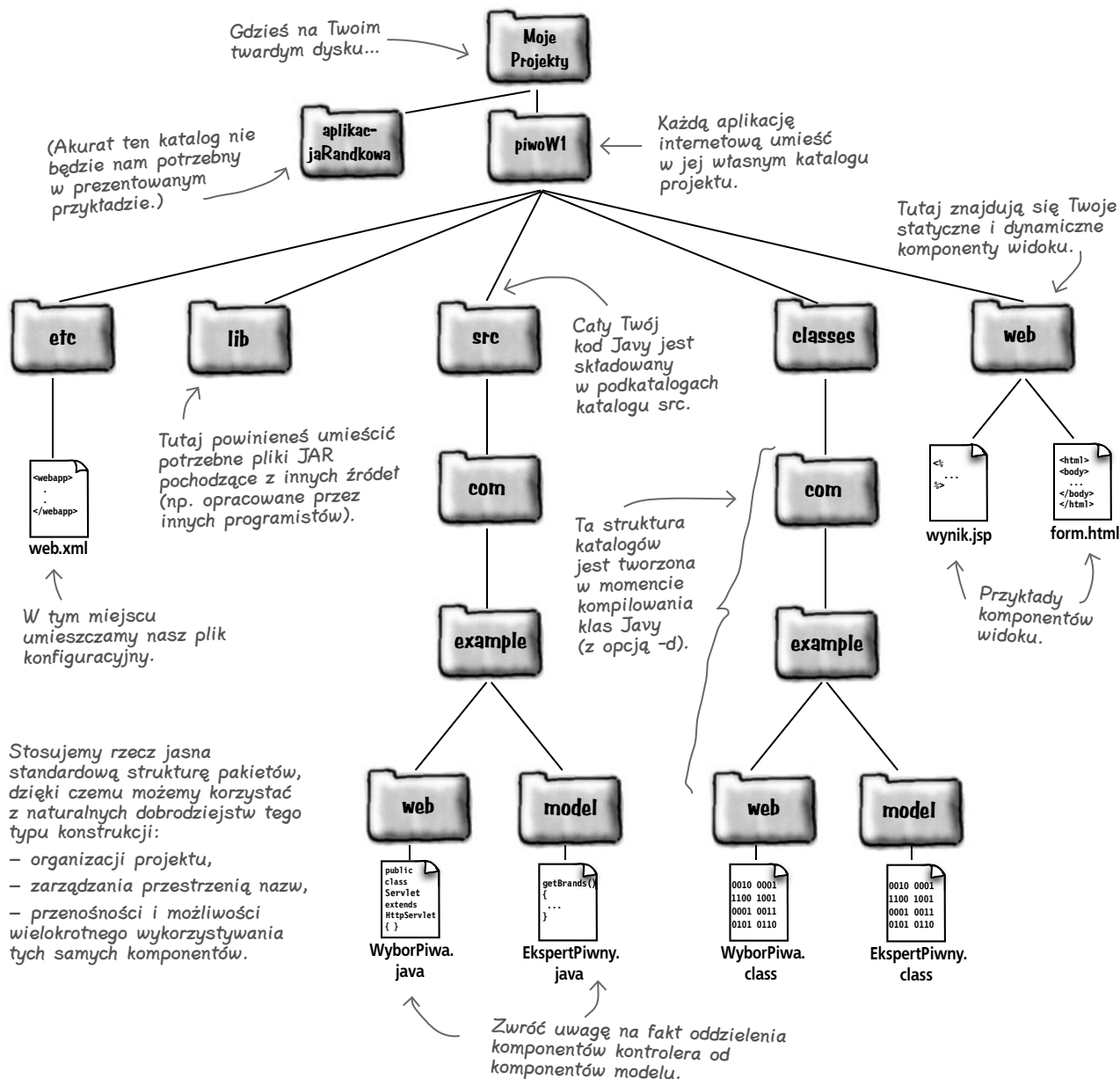


6. Serwlet wywołuje do pomocy komponent Ekspert+Piwny.
7. Klasa piwnego eksperta zwraca odpowiedź, którą serwlet dołącza do obiektu żądania.
8. Serwlet przekazuje żądanie dalej do strony JSP.
9. Strona JSP odczytuje odpowiedź z żądania obiektu.
10. Kod JSP generuje dla kontenera stronę wynikową.
11. Kontener zwraca tę stronę do szczęśliwego użytkownika.

Od tego miejsca, nawet jeśli na prezentowanym schemacie nie ma mowy o serwerze WWW, należy przyjąć, że serwer ten i tak musi gdzieś występować w architekturze aplikacji internetowej.

Tworzenie środowiska wytwarzania aplikacji

Istnieje wiele sposobów organizowania struktury katalogów stanowiącej środowisko wytwarzania aplikacji internetowych — poniżej przedstawiliśmy naszą propozycję tego typu struktury dla małych i średnich projektów. Kiedy przyjdzie czas wdrożenia naszej aplikacji internetowej, po prostu skopiujemy część tej struktury w miejsce wyznaczone do tego celu przez nasz kontener (w tym minipodręczniku wykorzystujemy kontener Tomcat w wersji 5.).



Tworzenie środowiska wdrażania aplikacji

Z wdrażaniem aplikacji internetowej wiąże się nie tylko szereg reguł właściwych dla wykorzystywanego kontenera, ale też wymagań zdefiniowanych w specyfikacjach serwetów i stron JSP (jeśli nie używasz Tomcata, będziesz musiał sprawdzić, w jaki sposób należy powiązać *Twój* kontener z Twoją aplikacją internetową). W tym przypadku cała struktura poniżej katalogu *piwoW1* pozostaje niezmieniona *niezależnie* od stosowanego kontenera!

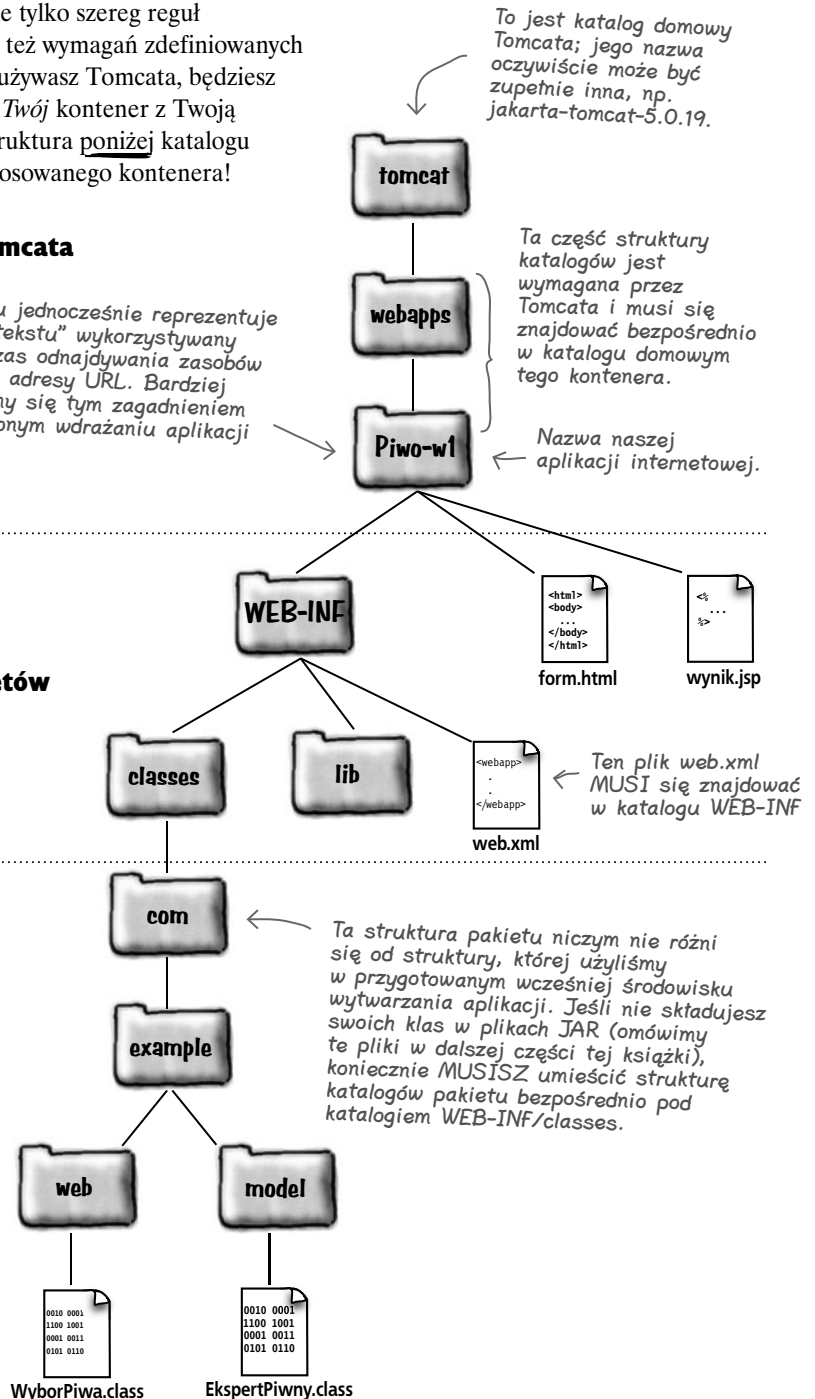
Katalogi właściwe dla Tomcata

Nazwa tego katalogu jednocześnie reprezentuje „katalog główny kontekstu” wykorzystywany przez Tomcata podczas odnajdywania zasobów wskazywanych przez adresy URL. Bardziej szczegółowo zajmiemy się tym zagadnieniem w rozdziale poświęconym wdrażaniu aplikacji internetowych.

Wszystko *POD* tą kropkowaną linią *STANOWI* już aplikację internetową, zatem będzie miało taką samą postać *niezależnie* od producenta wykorzystywanego kontenera WWW.

Część specyfikacji serwetów

Katalogi właściwe dla danej aplikacji



Nasz plan procesu budowy aplikacji

Na początku tego rozdziału przedstawiliśmy w skrócie czteroetapowy proces wytwarzania naszej aplikacji internetowej. Do tej pory zrealizowaliśmy następujące zadania:

1. Przeanalizowaliśmy *widoki* użytkownika dla naszej aplikacji internetowej.
2. Przyjrzeliliśmy się *architekturze* tej aplikacji.
3. Przygotowaliśmy środowiska *wytwarzania* i *wdrażania* naszej nowej aplikacji.

Nadszedł czas, by omówić krok czwarty, czyli właściwy proces *tworzenia* aplikacji internetowej.

Wykorzystamy w tym miejscu techniki zaczerpnięte z kilku popularnych metodyk wytwarzania oprogramowania (część pochodzi z tzw. programowania ekstremalnego, część z iteracyjnego wytwarzania aplikacji) i wymieszamy je do własnych celów...

Pięć kroków, które musimy wykonać w ramach kroku 4.:

- 4a **Budowa i testowanie formularza HTML**, za pośrednictwem którego użytkownik przygotowuje pierwsze żądanie.
- 4b **Budowa i testowanie pierwszej wersji testowej serwletu kontrolera** z formularzem HTML. Ta wersja będzie wywoływana za pośrednictwem formularza HTML, a jej jedynym zadaniem będzie wyświetlanie parametrów otrzymanych w żądaniu.
- 4c **Budowa klasy testowej** dla klasy eksperta (modelu) oraz skonstruowanie i przetestowanie właściwej klasy eksperta (modelu).
- 4d **Aktualizacja serwletu do wersji drugiej**. W tej wersji dodamy możliwość wywoływania klasy modelu i — tym samym — uzyskiwania porady dotyczącej piwa.
- 4e **Budowa strony JSP, aktualizacja serwletu do wersji trzeciej** (która doda możliwość przydzielania stron JSP) i przetestowanie całej aplikacji.

Kod HTML dla początkowej strony formularza

Nasz pierwszy kod HTML jest prosty — wyświetlamy tekst nagłówka, listę rozwijaną, za pomocą której użytkownik może wybrać odpowiedni kolor piwa, oraz przycisk akceptacji formularza.

```
<html><body>
<h1 align="center">Strona wyboru piwa</h1>
<form method="POST"
  action="WybierzPiwo.do">
  Wybierz właściwości piwa<p>
  Kolor:
  <select name="kolor" size="1">
    <option value="jasny"> jasny </option>
    <option value="bursztynowy"> bursztynowy </option>
    <option value="brązowy"> brązowy </option>
    <option value="ciemny"> ciemny </option>
  </select>
  <br><br>
  <center>
    <input type="SUBMIT">
  </center>
</form></body></html>
```

Dlaczego wybraliśmy metodę POST zamiast metody GET?

To jest nazwa, którą wykorzystujemy w kodzie HTML do wywołania odpowiedniego serwletu. W naszej strukturze katalogów nie ma nic, co miałoby nazwę WybierzPiwo.do! To tylko nazwa logiczna...

W ten sposób tworzymy listę rozwijaną (opcje w Twoim przykładzie mogą się oczywiście różnić od tych, które tutaj przedstawiliśmy) (Czy zwróciłeś uwagę na atrybut size="1"?)

P: Dlaczego nasz formularz odwołuje się do serwletu WybierzPiwo.do, skoro nasza aplikacja NIE zawiera serwletu z taką nazwą? W analizowanych przed chwilą strukturach katalogów nie widziałem pliku nazwanego WybierzPiwo.do. Nie wiem nawet, co oznacza samo rozszerzenie .do.

U: WybierzPiwo.do to tylko nazwa logiczna, która nie ma nic wspólnego z rzeczywistą nazwą pliku. Użyliśmy tej nazwy, ponieważ chcemy, aby była znana klientom naszej aplikacji! W praktyce klient NIGDY nie powinien mieć bezpośredniego dostępu do pliku klasy serwletu, zatem nie należy np. tworzyć stron HTML z łączami lub akcjami zawierającymi rzeczywiste ścieżki do plików klas serwletów.

W powyższym przykładzie zastosowaliśmy sztuczkę polegającą na użyciu deskryptora wdrożenia w formacie XML (wspominanego już pliku *web.xml*) do odwzorowania żądań klientów (*WybierzPiwo.do*) na faktyczną nazwę klasy serwletu (wykorzystywaną za każdym razem, gdy kontener otrzyma żądanie wskazujące na nieistniejący serwlet *WybierzPiwo.do*). Na razie możesz traktować rozszerzenie *.do* po prostu jak część logicznej nazwy serwletu (nie jak rzeczywisty typ pliku). W dalszej części tej książki dowiesz się nieco więcej na temat innych sposobów wykorzystywania rozszerzeń (zarówno tych rzeczywistych, jak i tych tworzonych na potrzeby klientów, czyli logicznych) w odwzorowaniach swoich serwletów.

Wdrażanie i testowanie strony otwierającej

Aby przetestować pierwszą stronę HTML naszej aplikacji internetowej, musimy ją umieścić w strukturze katalogów kontenera WWW (w naszym przypadku będzie to Tomcat), uruchomić ten kontener oraz wyświetlić tę stronę w oknie przeglądarki internetowej.

❶ Stwórz stronę HTML w swoim środowisku wytwarzania aplikacji

Stwórz nowy plik HTML, nazwij go *form.html* i zapisz w katalogu */piwoW1/web/* przygotowanego wcześniej środowiska wytwarzania.

❷ Skopiuj nowy plik do środowiska wdrażania aplikacji

Umieść kopię pliku *form.html* w katalogu *tomcat/webapps/Piwo-w1/* (pamiętaj, że katalog domowy Twojego kontenera Tomcat może mieć zupełnie inną nazwę).

❸ Opracuj deskryptor wdrożenia w swoim środowisku wytwarzania

Opracuj nowy dokument XML, nadaj mu nazwę *web.xml* i zapisz w katalogu */piwoW1/etc/* swojego środowiska wytwarzania.



```
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
  version="2.4">
```

Nie musisz wiedzieć, co znaczą te tajemnicze zapisy; po prostu wklep je w swoim pliku XML.

```
<servlet>
  <servlet-name>R3 Piwo</servlet-name>
  <servlet-class>com.example.web.WyborPiwa</servlet-class>
</servlet>
```

Jest to przykład sztucznej nazwy, która będzie wykorzystywana WYŁĄCZNIE w innych fragmentach tego samego deskryptora wdrożenia.

```
<servlet-mapping>
  <servlet-name>R3 Piwo</servlet-name>
  <url-pattern>/WybierzPiwo.do</url-pattern>
</servlet-mapping>
```

W pełni kwalifikowana nazwa pliku klasy serwletu.

```
</web-app>
```

Nie zapomnij o ukośniku na początku.

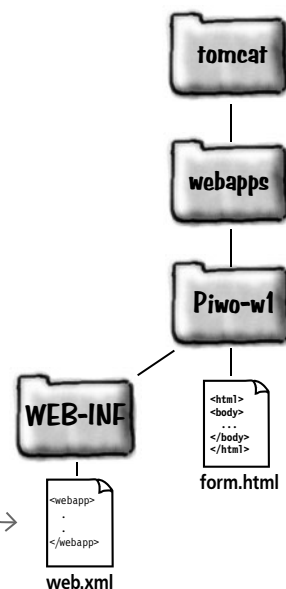
Chcemy, aby klient odwoływał się do naszego serwletu właśnie w ten sposób. Rozszerzenie .do zastosowano wyłącznie dla zachowania zgodności z konwencją.

Głównym zadaniem naszego deskryptora wdrożenia jest definiowanie odwzorowań pomiędzy nazwą logiczną wykorzystywaną przez klienta dla żądań (w tym przypadku *WybierzPiwo.do*) a faktyczną nazwą pliku klasy serwletu (w tym przypadku *com.example.web.WyborPiwa*).

4 Skopiuj plik deskryptora do środowiska wdrażania aplikacji

Umieść kopię pliku *web.xml* w katalogu *tomcat/webapps/Piwo-w1/WEB-INF/*.

Plik *web.xml* MUSI się znajdować w tym miejscu; w przeciwnym razie kontener tego pliku nie znajdzie i nasza aplikacja po prostu nie będzie działała, a Ty popadniesz w głęboką depresję.



5 Uruchom Tomcata

We wszystkich rozdziałach tej książki będziemy wykorzystywali Tomcata zarówno w roli *serwera* WWW, jak i w roli *kontenera* WWW. W praktyce najczęściej stosuje się bardziej wyszukany serwer WWW (np. Apache) skonfigurowany w sposób umożliwiający współpracę z jednym z istniejących kontenerów WWW (np. z Tomcatem). Tomcat oferuje jednak funkcjonalność przyzwoitego (choć skromnego) serwera WWW, który w zupełności wystarcza do analizy przykładów z tej książki.

Aby uruchomić Tomcata, przejdź do jego katalogu domowego i uruchom polecenie *bin/startup.sh*.

Plik Edycja Okno Pomoc OpenSource

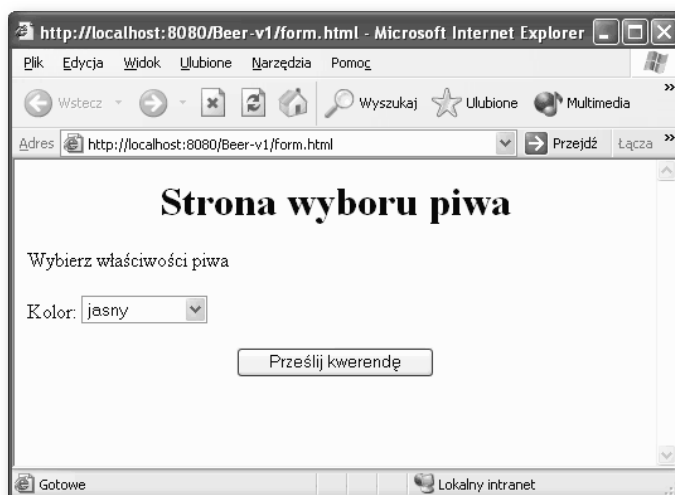
```
% cd tomcat
% bin/startup.sh
```

6 Przetestuj nową stronę

Otwórz okno przeglądarki internetowej i wpisz adres:

http://localhost:8080/Piwo-w1/form.html

Na ekranie powinna zostać wyświetlona strona podobna do tej na pokazanym obok zrzucie ekranu.



Odwzorowywanie nazw logicznych w pliki klas serwletów

- ❶ Dlane wypełnia formularz i klika przycisk jego akceptacji. Przeglądarka generuje następujący adres URL żądania:

→ **/Piwo-w1/WybierzPiwo.do**

Katalog główny serwera. Katalog główny kontekstu aplikacji internetowej. Logiczna nazwa zasobu.



Klient

```
POST /Piwo-w1/WybierzPiwo.do
HTTP/1.1
Host: www.wickedlysmart.com
User-Agent: Mozilla/5.0 (Macintosh;
U; PPC Mac OS X Mach-O; en-US;
rv:1.4)Gecko/20040624 Netscape/7.1
Accept: text/xml,application/xml,
application/xhtml+xml,text/
html;q=0.9;text/plain;q=0.8;video/
x-msg,image/png,image/jpeg,image/
gif;q=0.2,*/*;q=0.1
```



Kontener

W naszym kodzie HTML katalog **/Piwo-w1/** nie był częścią ścieżki do serwletu. Użyliśmy tam jedynie odwołania w postaci:

```
<form method="POST"
    action="WybierzPiwo.do">
```

Okazuje się jednak, że przeglądarka sama umieszcza nazwę katalogu **/Piwo-w1/** na początku żądania, ponieważ właśnie stamtąd pochodzi żądanie klienta. Innymi słowy, użyte w kodzie HTML odwołanie **WybierzPiwo.do** jest nazwą względną w stosunku do adresu URL bieżącej strony. W tym przypadku jest to ścieżka interpretowana względem katalogu głównego naszej aplikacji internetowej, czyli właśnie **/Piwo-w1/**.

- ❷ Kontener przeszukuje deskryptor wdrożenia i odnajduje element `<servlet-mapping>` z podelementem `<url-pattern>` pasującym do użytej w żądaniu nazwy **/WybierzPiwo.do** (gdzie ukośnik reprezentuje katalog główny kontekstu aplikacji internetowej, natomiast **WybierzPiwo.do** jest logiczną nazwą żądanego zasobu).



Kontener

- ❸ Kontener widzi, że nazwą serwletu (`<servlet-name>`) dla tego wzorca URL (`<url-pattern>`) jest **R3 Piwo**. Nie jest to jednak nazwa rzeczywistego pliku klasy serwletu. **R3 Piwo** jest nazwą serwletu, nie nazwą klasy!

Dla kontenera serwlet jest tylko czymś, co zostało nazwane w znaczniku `<servlet>` deskryptora wdrożenia. Nazwa serwletu jest po prostu ciągiem znaków używanym w ramach deskryptora wdrożenia w sposób umożliwiający jej odwzorowanie w pozostałych częściach tego deskryptora.



Kontener

```
<web-app>
  <servlet>
    <servlet-name>
      R3 Piwo
    </servlet-name>
    <servlet-class>
      com.example.web.WyborPiwa
    </servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>
      R3 Piwo
    </servlet-name>
    <url-pattern>
      /WybierzPiwo.do
    </url-pattern>
  </servlet-mapping>
</web-app>
```

- 4 Kontener zagląda do znacznika `<servlet>`, aby znaleźć zapis `R3 Piwo` w którymś z elementów `<servlet-name>`.



Kontener

```
<web-app>
  <servlet>
    <servlet-name>
      R3 Piwo
    </servlet-name>
    <servlet-class>
      com.example.web.WyborPiwa
    </servlet-class>
  </servlet>

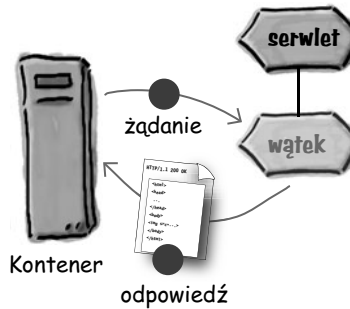
  <servlet-mapping>
    <servlet-name>
      R3 Piwo
    </servlet-name>
    <url-pattern>
      /WybierzPiwo.do
    </url-pattern>
  </servlet-mapping>
</web-app>
```

- 5 Kontener wykorzystuje podzacznik `<servlet-class>` znacznika `<servlet>` do określania, która klasa serwetu odpowiada za obsługę danego żądania. Jeśli odpowiedni serwet nie został jeszcze zainicjalizowany, klasa jest wczytywana, a sam serwet jest inicjalizowany.



Kontener

- 6 Kontener uruchamia dla otrzymanego żądania odrębny wątek, po czym przekazuje to żądanie nowemu wątkowi (a konkretnie metodzie `service()` serwetu).



- 7 Kontener odsyła do klienta odpowiedź (oczywiście za pośrednictwem serwera WWW).



Klient



Kontener



Pierwsza wersja serwletu kontrolera

Nasz plan przewiduje budowę serwletu w kilku etapach, które będą obejmowały testowanie na bieżąco rozmaitych łączy komunikacyjnych. Jak zapewne pamiętasz, serwlet w swojej ostatecznej wersji będzie przyjmował parametry z żądania, wywoływał metodę należącą do modelu, zapisywał informacje w miejscu, z którego będą odczytywane przez stronę JSP, oraz przekazywał żądanie do strony JSP. Naszym celem w pierwszej wersji będzie jednak wyłącznie upewnienie się co do możliwości prawidłowego wywołania serwletu przez stronę HTML oraz właściwego odebrania przekazanych przez tę stronę parametrów.

Kod serwletu

```
package com.example.web;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class WyborPiwa extends HttpServlet {

    public void doPost(HttpServletRequest request,
                        HttpServletResponse response)
                        throws IOException, ServletException {

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("Porada piwna<br>");
        String c = request.getParameter("kolor");
        out.println("<br>Wybrany kolor piwa: " + c);

    }
}
```

← Upewnij się, że zadeklarowany pakiet odpowiada stworzonym wcześniej strukturom wytwarzania i wdrażania aplikacji.

Klasa `HttpServlet` dziedziczy po klasie `GenericServlet`, która z kolei implementuje interfejs `Servlet`.

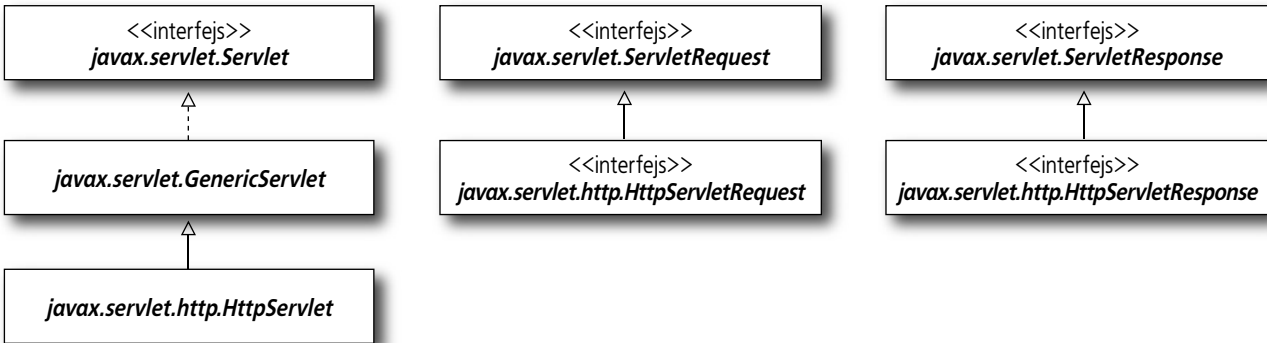
Użyjemy metody `doPost` do obsługi żądania protokołu HTTP, ponieważ w formularzu HTML zdefiniowano wcześniej atrybut: `method=POST`.

Ta metoda pochodzi z interfejsu `ServletResponse`.

Ta metoda pochodzi z interfejsu `ServletRequest`. Zwróć uwagę na fakt, iż argument tej metody odpowiada użytej w kodzie HTML wartości atrybutu `name` znacznika `<select>`.

W tym miejscu nie odsyłamy porady, wyświetlamy tylko informację testową.

Kluczowe interfejsy API



Kompilowanie, wdrażanie i testowanie serwletu kontrolera

No dobrze, zbudowaliśmy, wdrożyliśmy i przetestowaliśmy naszą stronę HTML; zbudowaliśmy i wdrożyliśmy także nasz deskryptor wdrożenia (co prawda umieściliśmy już plik *web.xml* w środowisku wdrażania, jednak z technicznego punktu widzenia nasz deskryptor nie będzie w pełni wdrożony do momentu ponownego uruchomienia Tomcata). Nadszedł czas skompilowania, wdrożenia i przetestowania (za pośrednictwem już istniejącego formularza HTML) pierwszej wersji naszego serwletu. Uruchomimy teraz kontener Tomcat, aby mieć pewność, że „widzi” zarówno deskryptor *web.xml*, jak i klasę serwletu.

Kompilowanie serwletu

Skompiluj serwlet z flagą `-d`, aby umieścić gotową klasę w środowisku *wdrażania*.

Zmodyfikuj ten fragment w taki sposób, aby odpowiadał strukturze katalogów istniejącej w Twoim systemie! Ścieżka za katalogiem *tomcat/* powinna pozostać niezmienną.

```

C:\> cd projekty\Piwo-w1

> javac -classpath d:\java\jakarta-tomcat-5.0.27\common\lib\servlet-api.jar:classes:.
-d src\com\example\web\WyborPiwa.java
  
```

Użyj opcji `-d`, aby wymusić na kompilatorze umieszczenie pliku klasy w katalogu *classes* w ramach prawidłowej struktury pakietów. Twój plik klasy (z rozszerzeniem *.class*) powinien się znaleźć w katalogu */piwoW1/classes/com/example/web/*.

W systemie operacyjnym Windows należałoby użyć średnika (*;*).

Wdrażanie serwletu

Aby wdrożyć serwlet, stwórz kopię wygenerowanego w poprzednim kroku pliku *.class* i przenieś ją do katalogu */Piwo-w1/WEB-INF/classes/com/example/web/* w przygotowanej wcześniej strukturze wdrożenia.

Testowanie serwletu

1. Uruchom ponownie Tomcata!
2. Uruchom swoją przeglądarkę i przejdź na stronę <http://localhost:8080/Piwo-w1/form.html>.
3. Wybierz kolor piwa i kliknij przycisk *Wyślij zapytanie*.
4. Jeśli Twój serwlet działa, powinieneś zobaczyć w oknie przeglądarki odpowiedź serwletu w następującej postaci:

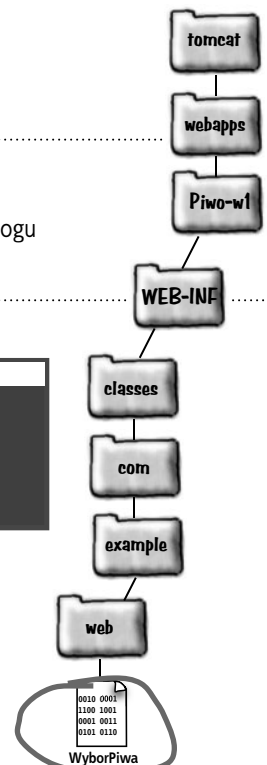
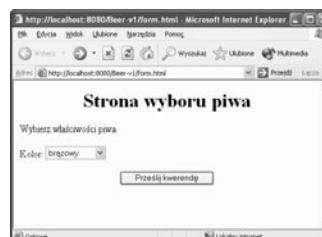
Porada piwna

Wybrany kolor piwa: brązowy

```

Plik Edycja Okno Pomoc UkośnikIKropki

% cd tomcat
% bin/shutdown.sh
% bin/startup.sh
  
```



Budowa i testowanie klasy modelu

We wzorcu projektowym MVC model jest traktowany jak „zaplecze” aplikacji. Funkcję modelu często pełni istniejący od dawna system informatyczny, którego funkcjonalność dopiero teraz jest udostępniana za pośrednictwem stron WWW. W większości przypadków jest to tradycyjny kod Javy opracowany bez świadomości zalet jego wywoływania z poziomu serwetów. Model nigdy nie powinien być ściśle wiązany z pojedynczą aplikacją internetową, zatem jego kod należy umieszczać w odrębnych pakietach.

Specyfikacja modelu

- kod modelu powinien należeć do pakietu `com.example.model`,
- struktura katalogów powinna być składowana w katalogu `/WEB-INF/classes/com/example/model`,
- model powinien udostępniać pojedynczą metodę `getMarki()`, która pobiera preferowany kolor piwa (w postaci łańcucha) i która zwraca obiekt klasy `ArrayList` z rekomendowanymi markami piwa (także w postaci łańcuchów).

Budowa i testowanie klasy dla danego modelu

Opracuj klasę testową dla danego modelu (tak, klasę testową należy przygotować *przed* zbudowaniem samego modelu). Teraz wszystko zależy tylko od Ciebie — nie musisz dalej realizować zaleceń tego skróconego podręcznika. Pamiętaj, że kiedy będziesz po raz pierwszy testował swój model, nadal będzie on pozostawał w środowisku wytwarzania — tak jak każdą inną klasę Javy, można ten model przetestować bez Tomcata.

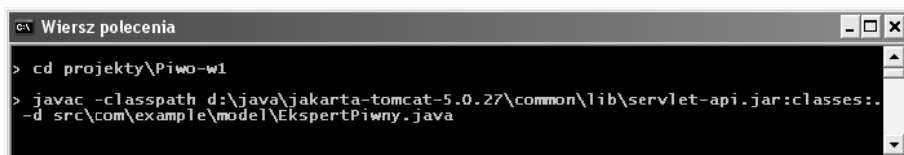
Budowa i testowanie samego modelu

Modele mogą być wyjątkowo skomplikowane. Często zawierają odziedziczone po istniejących systemach informatycznych połączenia z bazami danych i odwołania do skomplikowanej logiki biznesowej. Poniżej przedstawiono nasz wyszukany i oparty na zaawansowanych regułach system ekspercki w zakresie porad piwnych:

```
package com.example.model;
import java.util.*;

public class EkspertPiwny {
    public List getMarki(String kolor) {
        List marki = new ArrayList();
        if (kolor.equals("bursztynowy")) {
            marki.add("Jack Amber");
            marki.add("Red Moose");
        }
        else {
            marki.add("Jail Pale Ale");
            marki.add("Gout Stout");
        }
        return (marki);
    }
}
```

Zwróć uwagę na sposób, w jaki wyraziliśmy skomplikowaną wiedzę ekspercką na temat piwa za pomocą zaawansowanych wyrażeń warunkowych.



```
C:\ Wiersz polecenia
> cd projekty\Piwo-w1
> javac -classpath d:\java\jakarta-tomcat-5.0.27\common\lib\servlet-api.jar:classes:..
-d src\com\example\model\EkspertPiwny.java
```

Rozszerzanie kodu serwletu o wywołanie modelu — zapewnienie klientom RZECZYWISTYCH porad...

W nowej (*drugiej*) wersji serwletu uzupełnimy opracowaną wcześniej metodę `doPost()` o wywołanie modelu zapewniającego porady piwne (w *trzeciej* wersji porady będą przychodziły z odpowiedniej strony JSP). Zmiany wprowadzane w kodzie są zupełnie trywialne — najważniejsze jest w tym momencie dobre zrozumienie procesu ponownego wdrażania rozszerzonej aplikacji internetowej. Możesz teraz albo spróbować przygotować i ponownie skompilować kod aplikacji, po czym wdrożyć go w swoim środowisku, albo zajrzeć na kolejną stronę i wykorzystać zaproponowane tam rozwiązanie...



Zaostrz ołówek

Rozszerzenie serwletu — druga wersja

Zapomnij na chwilę o serwletach i pomyśl wyłącznie o klasach języka programowania Java. Jakie kroki powinniśmy podjąć, aby zrealizować następujące zadania?

1. Uzupełnienie metody `doPost()` o wywołanie modelu.
2. Skompilowanie zmienionego serwletu.
3. Wdrożenie i przetestowanie zaktualizowanej aplikacji internetowej.

```
public class WyborPiwa extends HttpServlet {
```

Druga wersja kodu serwletu

Pamiętaj, że model jest tylko zwykłą, tradycyjną klasą języka Javy, zatem wywołujemy jego funkcje dokładnie tak, jak wszelkie inne metody tego języka — tworzymy obiekt klasy modelu i wywołujemy jej metodę!

```
package com.example.web;

import com.example.model.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;

public class WyborPiwa extends HttpServlet {

    public void doPost(HttpServletRequest request,
                       HttpServletResponse response)
        throws IOException, ServletException {

        String c = request.getParameter("kolor");
        EkspertPiwny be = new EkspertPiwny();
        List wynik = be.getMarki(c);

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("Porada piwna<br>");

        Iterator it = wynik.iterator();
        while (it.hasNext()) {
            out.println("<br>Spróbuj: " + it.next());
        }
    }
}
```

Nie zapomnij zaimportować pakietu, do którego należy klasa EkspertPiwny.

Modyfikujemy oryginalny serwlet, ale wcale nie tworzymy nowej klasy.

Tworzymy obiekt klasy EkspertPiwny i wywołujemy metodę getMarki().

Wyświetlamy poradę (elementy zwróconego przez model obiektu klasy ArrayList, które reprezentują marki piwa). W ostatecznej (trzeciej) wersji porady będą wyświetlane przez stronę JSP, a nie (jak w tej wersji) przez serwlet.

Kluczowe kroki związane z finalizacją drugiej wersji serwletu

Pozostały nam do zrobienia jeszcze dwie rzeczy: *ponowne skompilowanie serwletu* oraz *wdrożenie klasy modelu*.

Kompilowanie serwletu

Zastosujemy to samo polecenie kompilatora, którego użyliśmy podczas kompilacji pierwszej wersji naszego serwletu.

Plik Edycja Okno Pomoc ZabawmySię

```
% cd piwoW1
% javac -classpath /Users/bert/Applications2/tomcat/common/lib/
servlet-api.jar:classes:. -d classes src/com/example/web/WyborPiwa.java
```

Wdrażanie i testowanie aplikacji internetowej

Poza samym serwletem musimy teraz wdrożyć także zbudowany model. Do kluczowych kroków tego procesu należą:

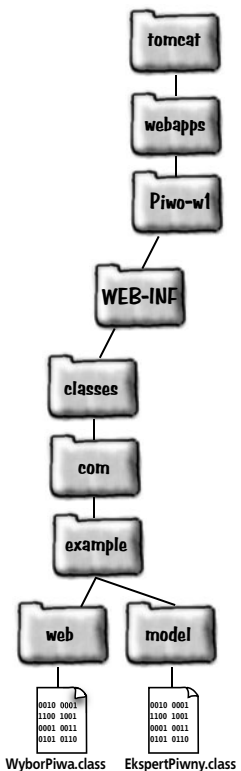
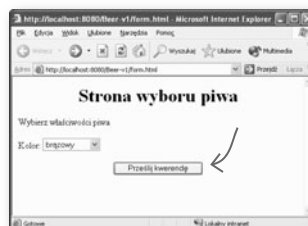
1. Przeniesienie kopii pliku `.class` serwletu do katalogu `../Piwo-w1/WEB-INF/classes/com/example/web/`. W ten sposób **zastąpimy** pierwszą wersję pliku klasy serwletu!
2. Przeniesienie kopii pliku `.class` modelu do katalogu `../Piwo-w1/WEB-INF/classes/com/example/model/`.
3. Zatrzymanie i **ponowne uruchomienie Tomcata**.
4. **Przetestowanie aplikacji** za pośrednictwem strony internetowej `form.html`, w oknie przeglądarki ostatecznie powinna zostać wyświetlona następująca porada:

Porada piwna

Spróbuj: Jack Amber

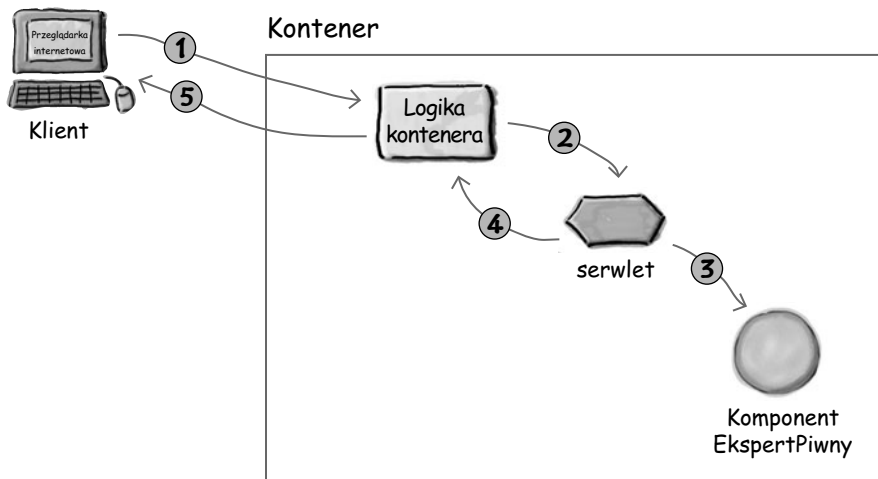
Spróbuj: Red Moose

```
File Edit Window Help ShellHigh
% cd tomcat
% bin/shutdown.sh
% bin/startup.sh
```



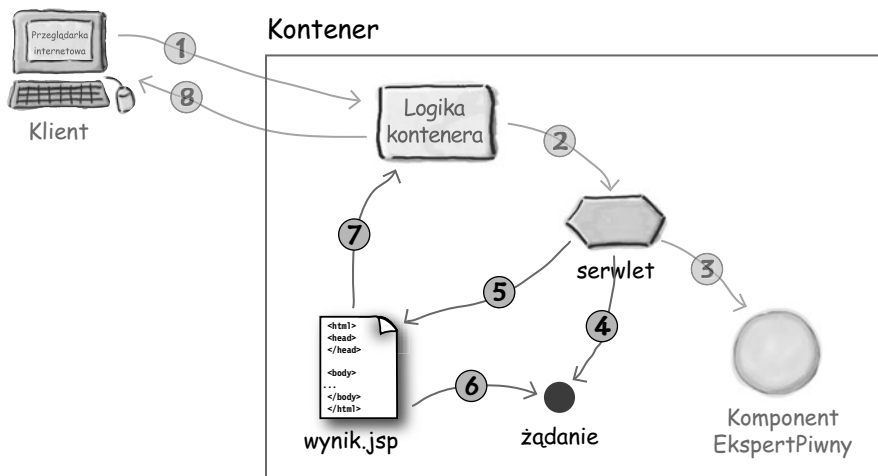
Przegląd prawie kompletnej aplikacji internetowej MVC zapewniającej porady piwne

Co już działa...



1. Przeglądarka wysyła do kontenera dane żądania.
2. Kontener odnajduje (w oparciu o użyty adres URL) właściwy serwet i przekazuje do tego serwetu otrzymane wcześniej żądanie.
3. Serwet wywołuje do pomocy komponent EkspertPiwny.
4. Serwet wyświetla odpowiedź (w formie porady piwnej).
5. Kontener zwraca szczęśliwemu użytkownikowi gotową stronę.

Co CHCEMY osiągnąć...



1. Przeglądarka wysyła do kontenera dane żądania.
2. Kontener odnajduje (w oparciu o użyty adres URL) właściwy serwet i przekazuje do tego serwetu otrzymane wcześniej żądanie.
3. Serwet wywołuje do pomocy komponent EkspertPiwny.
4. Klasa eksperta zwraca odpowiedź, którą serwet dołącza do obiektu żądania.
5. Serwet przekazuje to żądanie dalej do odpowiedniej strony JSP.
6. Strona JSP pobiera odpowiedź z obiektu żądania.
7. Strona JSP generuje stronę HTML dla kontenera.
8. Kontener zwraca szczęśliwemu użytkownikowi gotową stronę.

Opracowanie „widoku” JSP, który będzie przekazywał poradę

Nie trać nadziei. Będziesz się musiał wykazać cierpliwością jeszcze przez kilka rozdziałów, zanim rzeczywiście zaczniemy analizować technologię stron JSP. Przedstawiony poniżej kod JSP nie jest szczególnie dobrym przykładem (przede wszystkim z powodu kodu skryptletu, któremu poświęcimy trochę uwagi w dalszej części tej książki). Z drugiej strony, kod strony JSP w tej formie nikomu nie powinien sprawić kłopotu, zatem jeśli chcesz trochę poeksperymentować, nie powinieneś napotykać żadnych przeszkód. Chociaż *moglibyśmy* już teraz przetestować tę stronę JSP w przeglądarce, wstrzymamy się do czasu zmodyfikowania serwletu (do wersji trzeciej), aby przekonać się, jak działa cała nasza aplikacja.

Oto nasz kod JSP...

```
<%@ page import="java.util.*" %>
<html>
<body>
<h1 align="center">JSP z rekomendacjami dla piwa</h1>
<p>

<%
    List styles = (List)request.getAttribute("styles");
    Iterator it = styles.iterator();
    while (it.hasNext()) {
        out.print("<br>Spróbuj: " + it.next());
    }
%>

</body>
</html>
```

← To jest „dyrektywa strony” (naszym zdaniem znaczenie tego wiersza nie wymaga komentarza).

← Trochę standardowego kodu HTML (który w świecie JSP określa się mianem „tekst szablonu”).

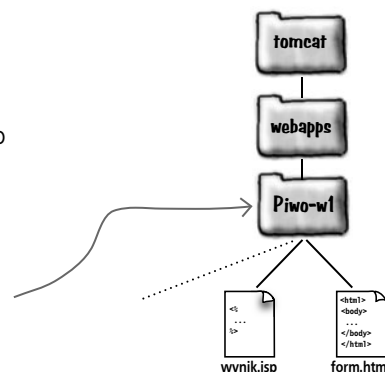
← W tym miejscu pobieramy atrybut z obiektu żądania. Nieco później w tej książce wyjaśnimy wszystkie aspekty stosowania atrybutów i technik uzyskiwania dostępu do obiektu żądania...

← Trochę standardowego kodu Javy umieszczonego w znacznikach `<% %>` (zawartość tych znaczników często określa się mianem kodu skryptletu).

Wdrażanie strony JSP

Nie kompilujemy strony JSP (zrobi to kontener zaraz po otrzymaniu pierwszego żądania dotyczącego tej strony). *Musimy* jednak wykonać następujące kroki:

1. Nadać jej nazwę `wynik.jsp`.
2. Zapisać ją w katalogu `/web/` środowiska wytwarzania.
3. Przenieść kopię tego pliku do katalogu `/Piwo-w1/` środowiska wdrożenia.

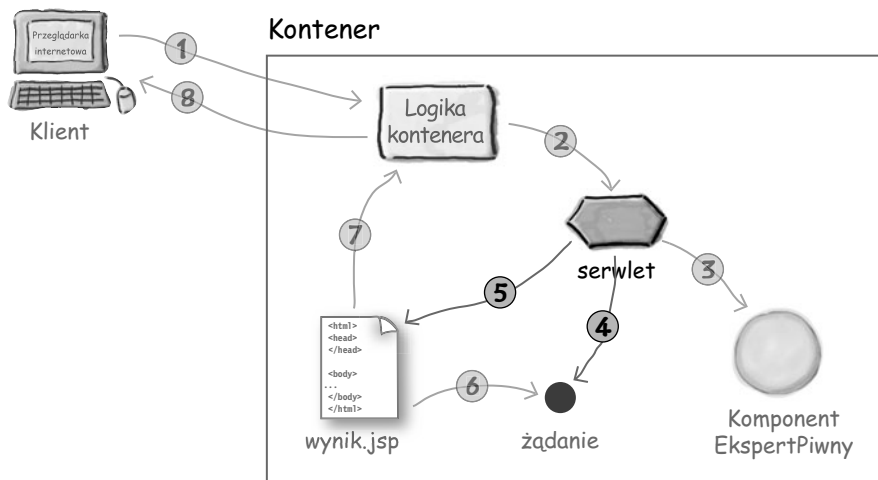


Rozszerzenie serwletu o „wywołanie” strony JSP (wersja trzecia)

W tym kroku mamy zamiar tak zmodyfikować nasz serwer, aby „wywoływał” stronę JSP wyświetlającą dane wyjściowe (widok). Kontener zapewnia mechanizm nazywany *przekazywaniem żądań*, który umożliwia komponentom zarządzanym przez kontener wywoływanie innych komponentów. Właśnie na tym zadaniu skoncentrujemy się w tym podrozdziale — serwlet pobierze niezbędne informacje z modelu, zapisze je w obiekcie żądania i *przekaze to żądanie do odpowiedniej strony JSP*.

Najważniejsze zmiany, które musimy wprowadzić w serwlecie:

1. Dodanie odpowiedzi uzyskanej z komponentu modelu do obiektu żądania, aby odpowiednia strona JSP miała do niej dostęp (patrz krok 4. na rysunku).
2. Wymuszenie na kontenerze przekazania żądania stronie *wynik.jsp* (patrz krok 5.).



1. Przeglądarka wysyła do kontenera dane żądania.
2. Kontener odnajduje (w oparciu o użyty adres URL) właściwy serwlet i przekazuje do niego otrzymane wcześniej żądanie.
3. Serwlet wywołuje do pomocy komponent EkspertPiwny.
4. Klasa eksperta zwraca odpowiedź, którą serwlet dołącza następnie do obiektu żądania.
5. Serwlet przekazuje żądanie do strony JSP.
6. Strona JSP pobiera odpowiedź z obiektu żądania.
7. Strona JSP generuje stronę HTML dla kontenera.
8. Kontener zwraca szczęśliwemu użytkownikowi gotową stronę.

Trzecia wersja kodu serwletu

Poniżej przedstawiono zmodyfikowany kod serwletu, który od tej pory będzie dodawał do obiektu żądania odpowiedź uzyskaną z komponentu modelu (aby była dostępna z poziomu strony JSP) oraz wymuszał na kontenerze przekazywanie obiektu żądania do właściwej strony JSP.

```
package com.example.web;

import com.example.model.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;

public class WyborPiwa extends HttpServlet {

    public void doPost(HttpServletRequest request,
                       HttpServletResponse response)
        throws IOException, ServletException {

        String c = request.getParameter("kolor");
        EkspertPiwny be = new EkspertPiwny();
        List wynik = be.getMarki(c);

        // response.setContentType("text/html");
        // PrintWriter out = response.getWriter();
        // out.println("Porada piwna<br>");

        request.setAttribute("styles", wynik);

        RequestDispatcher view = request.getRequestDispatcher("wynik.jsp");

        view.forward(request, response);
    }
}
```

Skoro teraz to strona JSP ma generować dane wyjściowe, możemy usunąć z kodu serwletu wyrażenia odpowiedzialne za ich testowe wyświetlanie. Pozostawiliśmy te wyrażenia w formie komentarzy, aby nadal były widoczne w kodzie naszego serwletu.

Dodajemy do obiektu żądania atrybut, który będzie wykorzystywany przez docelową stronę JSP. Zwróć uwagę na fakt, iż wspomniana strona JSP będzie szukała atrybutu styles.

Tworzymy obiekt przekazujący żądanie do właściwej strony JSP.

Wykorzystujemy obiekty klasy RequestDispatcher do wymuszania na kontenerze zaangażowania wskazanej strony JSP (w tym przekazania tej stronie obiektów żądania i odpowiedzi).

Kompilacja, wdrożenie i przetestowanie ostatecznej wersji aplikacji!

W tym rozdziale zbudowaliśmy całą (choć bardzo niewielką) aplikację MVC składającą się ze strony HTML, serwletów oraz stron JSP. Możesz tę aplikację uwzględnić w swoim CV.

Kompilowanie serwletu

Użyjemy tego samego polecenia kompilatora, którego użyliśmy podczas kompilacji dwóch poprzednich wersji naszego serwletu.

Plik Edycja Okno Pomoc UciekajToPułapka

```
% cd piwoW1
% javac -classpath /Users/bert/Applications2/tomcat/common/lib/
servlet-api.jar:classes:. -d classes src/com/example/web/WyborPiwa.java
```

Wdrażanie i testowanie aplikacji internetowej

Czas ponownie wdrożyć nasz serwlet.

1. Przenieś kopię pliku `.class` serwletu do katalogu `../Piwo-w1/WEB-INF/classes/com/example/web/` (także tym razem należy **zastąpić** poprzednią wersję pliku klasy serwletu).
2. Zatrzymaj i uruchom ponownie Tomcata.
3. Przetestuj aplikację za pośrednictwem strony internetowej `form.html`.

Plik Edycja Okno Pomoc RatujSię

```
% cd tomcat
% bin/shutdown.sh
% bin/startup.sh
```



Taką stronę powinieneś zobaczyć w swojej przeglądarce! →



Dobrze, wiemy już, że potrafi robić aplikacje MVC, ale nadal nie ma pojęcia, jak używać języka wyrażeń JSP, jak korzystać z biblioteki JSTL, jak pisać własne znaczniki oraz jak używać filtrów. Co więcej, przytapałem go na słuchaniu płyty grupy Weezer i to już PO wydaniu zielonego albumu. Cóż, ten chłopak musi się jeszcze sporo nauczyć...



Pozostało nam jeszcze mnóstwo zagadnień do omówienia

Zabawa się skończyła. Zapoznałeś się już z treścią trzech rozdziałów, napisałeś trochę kodu, miałeś okazję przeanalizować procesy przetwarzania żądań i odpowiedzi protokołu HTTP.

Warto jednak pamiętać, że pozostało nam jeszcze około 200 przykładowych pytań egzaminacyjnych sformułowanych w tej książce — odpowiedzi na te pytania można znaleźć w treści następnego i kolejnych rozdziałów. Jeśli nie dysponujesz wystarczającą wiedzą na temat wytwarzania i wdrażania serwetów, nie powinieneś tak naprawdę przewracać tej strony (być może lepszym wyjściem będzie powtórne *zapoznanie* się z materiałem tego rozdziału).

Nie myśl tylko, że próbujemy wywołać u Ciebie wyrzuty sumienia lub jakiegokolwiek poczucie winy...

4. Żądanie i odpowiedź

Być serwiletem

Użył żądania GET do zaktualizowania bazy danych. Kara musi być jak najsurowsza... żadnych zajęć z jogi przez najbliższych 90 dni.



Serwleły istnieją po to, by obsługiwać klientów. Zadaniem serwletu jest obsługa *żądań* klientów i odsyłanie im odpowiednich *odpowiedzi*. Żądanie może być zupełnie proste, np. „*prześlij mi stronę powitalną*”, lub znacznie bardziej skomplikowane, np. „*wygeneruj zamówienie na podstawie zawartości mojego koszyka*”. Żądanie obejmuje kluczowe dane, a kod Twojego serwletu musi wiedzieć, jak należy te dane *odszukać* i jak można ich *użyć*. Odpowiedź musi zawierać informacje niezbędne do wizualizacji strony (lub pobrania bajtów) przez przeglądarkę, zatem kod Twojego serwletu musi wiedzieć, jak te informacje wysłać. Okazuje się jednak, że może być *inaczej*... Twój serwlet może decydować o przekazaniu żądania zupełnie *gdzie indziej* — do innej strony, serwletu lub JSP.



Model technologii serwletów

- 1.1.** Dla każdej z metod przesyłania żądań protokołu HTTP (takich jak GET, POST, HEAD itp.) opisz jej przeznaczenie i charakterystyki techniczne, wymień czynniki, które mogą decydować o wyborze danej metody przez klienta (zazwyczaj przeglądarkę internetową), oraz zidentyfikuj metodę klasy `HttpServlet` właściwą danej metodzie protokołu HTTP.
- 1.2.** Korzystając z interfejsu `HttpServletRequest`, napisz kod odczytujący z żądania protokołu HTTP parametry formularza HTML, odczytujący informacje zawarte w nagłówku protokołu HTTP lub odczytujący z żądania znaczniki kontekstu (ciasteczka, cookies) klienta.
- 1.3.** Korzystając z interfejsu `HttpServletResponse`, napisz kod ustawiający nagłówki odpowiedzi protokołu HTTP, ustawiający typ zawartości odpowiedzi, uzyskujący dostęp do strumienia tekstowego odpowiedzi, uzyskujący dostęp do strumienia binarnego odpowiedzi, przekierowujący żądanie HTTP na inny adres URL lub dodający do odpowiedzi ciasteczka klienta*.
- 1.4.** Opisz znaczenie i sekwencję zdarzeń składających się na cykl życia serwletu: (1) wczytanie klasy serwletu, (2) stworzenie obiektu (konkretyzacja) klasy serwletu, (3) wywołanie metody `init()`, (4) wywołanie metody `service()` oraz (5) wywołanie metody `destroy()`.

* Celom egzaminu związanym ze znacznikami kontekstu klienta (tzw. ciasteczkami) poświęcimy więcej czasu dopiero w rozdziale dotyczącym sesji.

Wdrażanie aplikacji internetowych

Wszystkie wymienione obok cele egzaminu zostaną dokładnie omówione jeszcze w tym rozdziale (wyjątkiem jest fragment celu 1.3 poświęcony ciasteczkom). O znacznej części zagadnień opisywanych w tym rozdziale wspominaliśmy już w rozdziale 2., gdzie jednak zasugerowaliśmy Czytelnikowi, że nie musi zapamiętywać prezentowanych treści.

W tym rozdziale **MUSISZ** nieco zwolnić i przystąpić do naprawdę uważnego studiowania i zapamiętywania prezentowanego materiału. Pamiętaj, że do szczegółów wymienionych obok celów egzaminu nie będziemy wracali w kolejnych rozdziałach, zatem masz ostatnią szansę ich dogłębnej analizy.

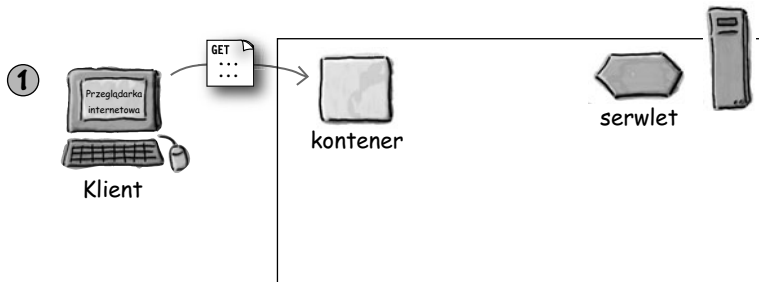
Wykonuj wszystkie ćwiczenia opisywane w tym rozdziale, dokładnie zapoznając się z całym materiałem, i w skupieniu spróbuj odpowiedzieć na pytania pierwszego egzaminu próbnego na końcu rozdziału. Jeśli odsetek prawidłowych odpowiedzi nie przekroczy 80%, **ZANIM** przystąpisz do lektury kolejnego (piątego) rozdziału, wróć do treści rozdziału i jeszcze raz zapoznaj się z materiałem, którego nieznanomość została ujawniona w trakcie testu.

Niektóre z próbnych pytań egzaminacyjnych, które dotyczą wymienionych obok celów, zostały przeniesione do rozdziałów 5. i 6., ponieważ odpowiedź na nie wymaga dodatkowej wiedzy w zakresie omawianym dopiero w tamtych rozdziałach. Oznacza to, że po przeczytaniu tego rozdziału będziesz musiał odpowiedzieć na mniejszą liczbę pytań egzaminacyjnych niż w późniejszych rozdziałach (w ten sposób uniknęliśmy testowania wiedzy, której nie miałeś jeszcze okazji osiąść).

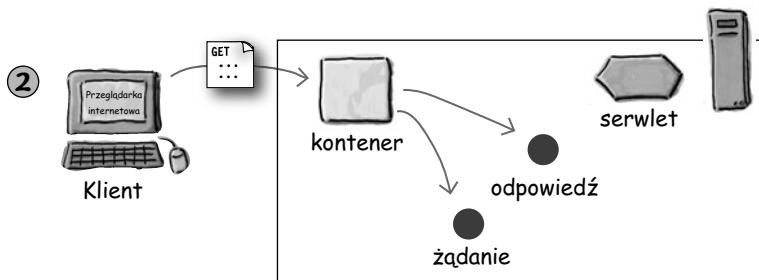
Ważna uwaga: o ile w pierwszych trzech rozdziałach omawialiśmy przede wszystkim informacje stanowiące tło dla właściwej treści tej książki, o tyle już od następnej strony niemal cały materiał będzie bezpośrednio związany z konkretnymi fragmentami egzaminu.

Serwlety są kontrolowane przez kontener

W rozdziale drugim pobieżnie omówiliśmy rolę kontenera w życiu serwletu — tworzy on obiekty żądania i odpowiedzi, tworzy lub przydziela nowy wątek serwletu oraz wywołuje metodę `service()` serwletu (przekazując za pośrednictwem jej argumentów referencje do obiektów żądania i odpowiedzi). Oto krótkie przypomnienie...

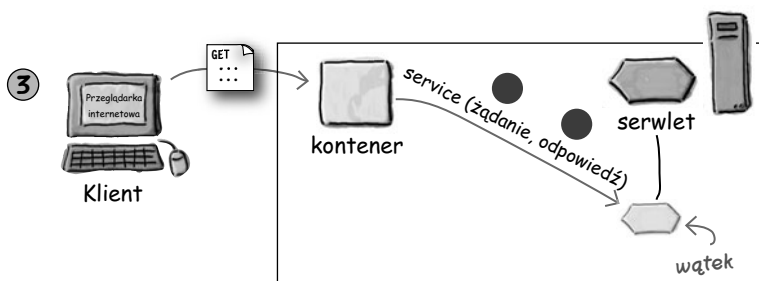


Użytkownik klika łącze z adresem URL serwletu.



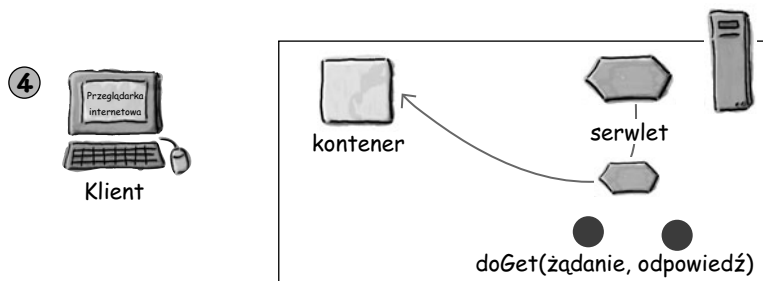
Kontener „widzi”, że żądanie jest kierowane do serwletu, zatem tworzy dwa niezbędne obiekty:

1. `HttpServletResponse`
2. `HttpServletRequest`



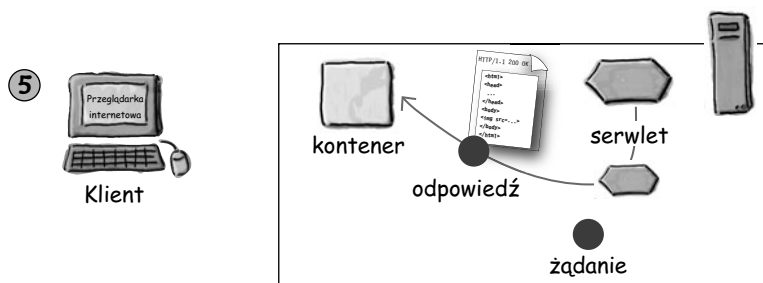
Kontener odnajduje odpowiedni serwlet na podstawie adresu URL dołączonego do żądania, tworzy lub przydziela wątek obsługujący to żądanie oraz wywołuje metodę `service()` serwletu i przekazuje w formie jej argumentów obiekty reprezentujące żądanie i odpowiedź.

Dalszy ciąg historii życia serwletu...

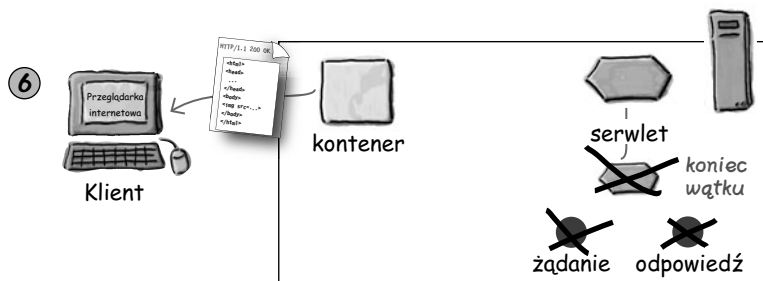


Metoda `service()` określa na podstawie przysłanej przez klienta metody protokołu HTTP (GET, POST itp.), którą metodę serwletu należy wykonać.

W tym przypadku klient wysłał żądanie HTTP GET, zatem metoda `service()` wywołuje metodę `doGet()` z obiektem żądania i obiektem odpowiedzi przekazanymi w formie argumentów.



Serwlet wykorzystuje obiekt odpowiedzi do zapisania swojej odpowiedzi dla klienta. Odpowiedź jest odsyłana do klienta za pośrednictwem kontenera.

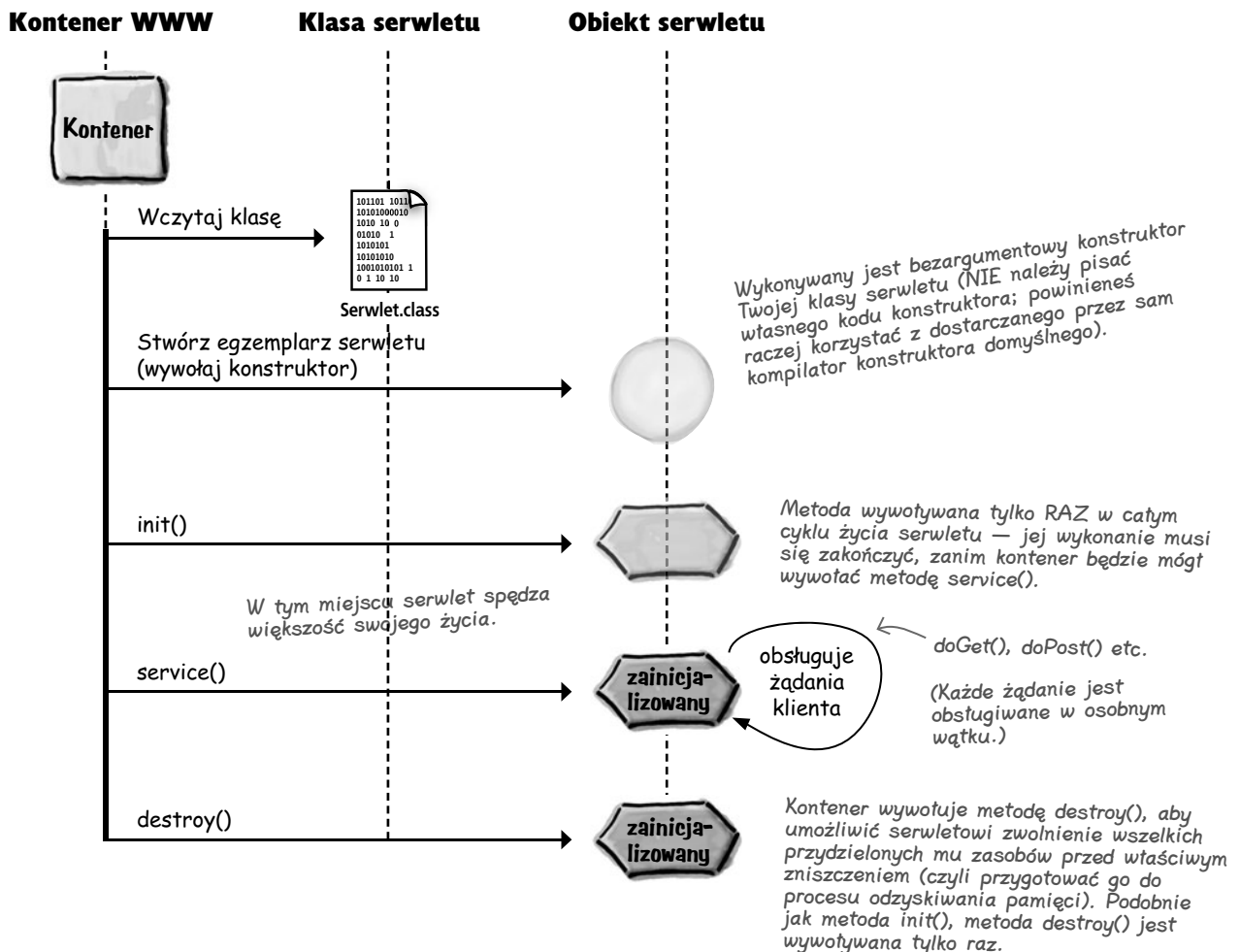
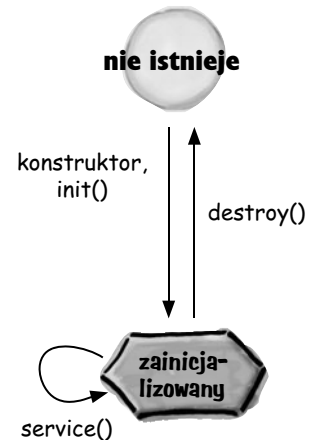


Wykonywanie metody `service()` kończy się, zatem odpowiedni wątek jest albo zabijany, albo zwracany do zarządzanej przez kontener puli wątków. Referencje do obiektów reprezentujących żądanie i odpowiedź wypadają poza bieżący zakres, zatem zajmowana przez nie pamięć może zostać zwolniona (w procesie odświeżania pamięci).

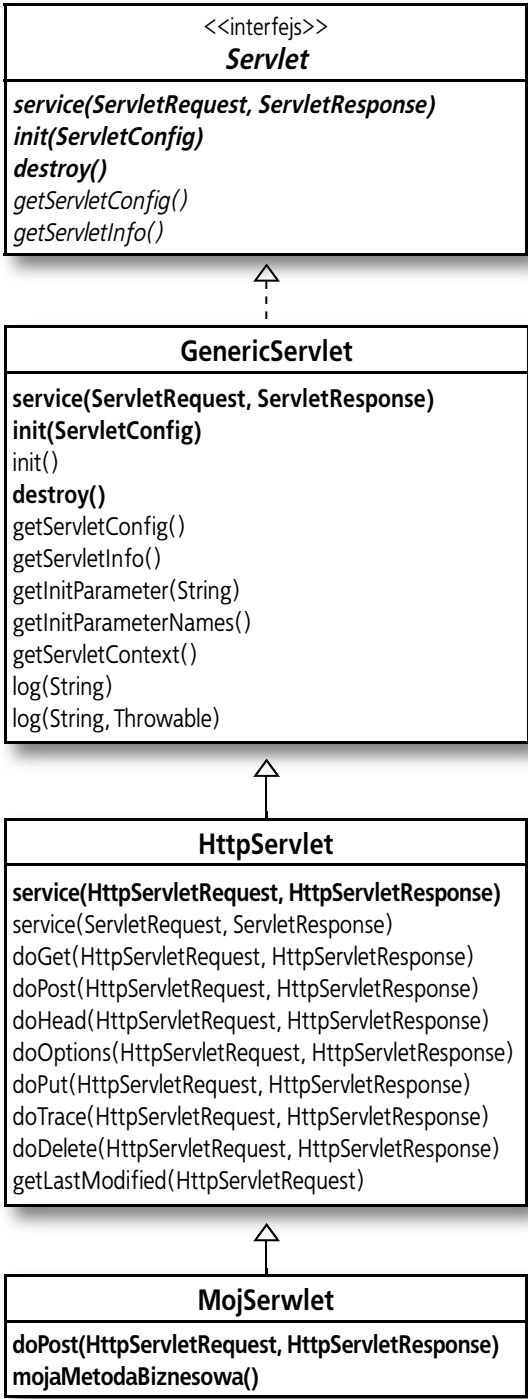
Życie serwletu toczy się jednak dalej

W naszych rozważaniach gładko przeszliśmy do środka życia serwletu, nadal jednak pozostawiliśmy bez odpowiedzi wiele istotnych pytań. Kiedy klasa serwletu jest wczytywana? Kiedy jest wywoływany konstruktor serwletu? Jak długo żyje obiekt serwletu? Kiedy powinniśmy inicjalizować zasoby naszego serwletu? Kiedy należy te zasoby zwolnić?

Cykl życia serwletu jest prosty; istnieje tylko jeden stan — *zainicjalizowany*. Jeśli serwlet nie jest zainicjalizowany, musi się znajdować w stanie *jest inicjalizowany* (jeśli w danej chwili wykonywany jest jego konstruktor lub metoda `init()`), w stanie *jest niszczone* (jeśli w danym momencie wykonywana jest jego metoda `destroy()`) lub po prostu w stanie *nie istnieje*.



Twój serwlet dziedziczy metody cyklu życia



Interfejs Servlet

(javax.servlet.Servlet)

Interfejs Servlet określa, że wszystkie serwlety muszą zawierać te pięć metod (trójka metod wyróżniona pogrubieniem należy do cyklu życia serwletu).

UWAGA: NIE próbuj zapamiętywać wszystkich wymienionych na tej stronie informacji! Zapoznaj się tylko ze sposobem funkcjonowania tego interfejsu API...

Klasa GenericServlet

(javax.servlet.GenericServlet)

GenericServlet jest klasą abstrakcyjną, która implementuje większość niezbędnych metod serwletów, włącznie z tymi zadeklarowanymi w interfejsie Servlet. Prawdopodobnie NIGDY nie będziesz samodzielnie rozszerzał tej klasy. „Zachowania” większości Twoich serwletów będą pochodziły właśnie z klasy GenericServlet.

Klasa HttpServlet

(javax.servlet.http.HttpServlet)

Klasa HttpServlet (także abstrakcyjna) implementuje metodę service(), która jest niezbędna do zapewnienia zgodności serwletu z protokołem HTTP — metoda service() nie otrzymuje na wejściu ŻADNYCH obiektów żądań i odpowiedzi samego serwletu, tylko właśnie żądanie i odpowiedź HTTP.

Klasa MojSerwlet

(com.wickedlysmart.foo)

Większość serwletowości Twojej aplikacji jest obsługiwana przez metody dziedziczone po rozszerzanych nadklasach. W swojej klasie serwletu musisz napisać tylko te metody HTTP, których potrzebujesz.

Trzy najważniejsze momenty w cyklu życia serwletu

1

init()

Kiedy jest wywoływana?

Kontener wywołuje metodę `init()` obiektu serwletu *po* jego utworzeniu, ale *zanim* dany serwlet będzie mógł obsłużyć jakiegokolwiek żądania klientów.

Do czego służy?

Daje programiście możliwość inicjalizowania serwletu jeszcze przed przystąpieniem do obsługi żądań klientów.

Czy będziesz ją nadpisywał?

Być może.

Jeśli tworzysz kod inicjalizujący aplikację internetową (np. przez utworzenie połączenia z bazą danych lub zarejestrowanie innych obiektów), musisz nadpisać metodę `init()` w klasie swojego serwletu.

2

service()

Kiedy jest wywoływana?

Kiedy kontener otrzymuje pierwsze żądanie klienta, uruchamia nowy wątek serwletu lub przydziela do nowego zadania wątek już istniejący w puli wątków, po czym wymusza wywołanie metody `service()` danego serwletu.

Do czego służy?

Metoda analizuje otrzymane żądanie, określa metodę protokołu HTTP (GET, POST itp.) i wywołuje odpowiednią metodę (`doGet()`, `doPost()` itp.) obiektu serwletu.

Czy będziesz ją nadpisywał?

Nie, to bardzo mało prawdopodobne.

NIE powinieneś nadpisywać metody `service()`. Twoim zadaniem jest nadpisywanie metody `doGet()` i (lub) metody `doPost()` i pozostawienie odpowiedzialności za wywoływanie właściwej części kodu oryginalnej metodzie `service()` klasy `HttpServlet`.

3

**doGet()
i (lub)
doPost()**

Kiedy są wywoływane?

Metoda `service()` wywołuje metodę `doGet()` lub `doPost()` w zależności od metody protokołu HTTP (GET, POST itp.) otrzymanej w ramach żądania od klienta.

(W tym miejscu wspominamy wyłącznie o metodach `doGet()` i `doPost()`, ponieważ są to prawdopodobnie jedyne tego typu metody, z których kiedykolwiek będziesz korzystał).

Do czego służą?

Właśnie w tym miejscu zaczyna się *Twój* kod! Metoda `doGet()` lub `doPost()` odpowiada za realizację właściwych ZADAŃ Twojej aplikacji internetowej.

Możesz oczywiście wywoływać inne metody należące do innych obiektów, jednak wszelkie tego typu działania mają swój początek właśnie w tym miejscu.

Czy będziesz je nadpisywał?

ZAWSZE, przynajmniej JEDNĄ z nich (doGet() lub doPost())!

To, którą metodę (lub które metody) nadpisujesz, jest sygnałem dla kontenera, które typy żądań będą obsługiwane w Twoim serwlecie. Jeśli na przykład nie nadpiszesz metody `doPost()`, tym samym dasz kontenerowi jasno do zrozumienia, że Twój serwlet nie obsługuje żądań POST protokołu HTTP.

Myślę, że już wiem, o co chodzi... zatem kontener wywołuje metodę `init()` mojego serwletu, ale jeśli tej metody nie nadpiszę w swoim kodzie, zostanie wykonana domyślna metoda `init()` odziedziczona po klasie `GenericServlet`. Następnie, kiedy do serwera dociera żądanie klienta, kontener uruchamia nowy wątek lub wykorzystuje jeden z wątków dostępnych w puli, po czym wywołuje metodę `service()`, której nie nadpisuję (zatem faktycznie wykonywana jest metoda `service()` z klasy `HttpServlet`). Metoda `service()` klasy `HttpServlet` wywołuje następnie nadpisaną przeze mnie metodę `doGet()` lub `doPost()`. Oznacza to, że każde wywołanie mojej metody `doGet()` lub `doPost()` jest realizowane w odrębnym wątku.

Metoda `service()` zawsze jest wywoływana w ramach jej własnego stosu...



Inicjalizacja serwletu

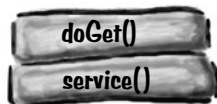


Wątek A

Kontener wywołuje metodę `init()` obiektu serwletu, oczywiście już po utworzeniu tego obiektu, ale *zanim* serwlet będzie mógł obsłużyć jakiegokolwiek żądania klienta.

Jeśli tworzysz kod inicjalizujący aplikację internetową (np. przez utworzenie połączenia z bazą danych lub rejestrację innych obiektów), musisz nadpisać metodę `init()` w klasie swojego serwletu. W przeciwnym razie zostanie wykonana metoda `init()` odziedziczona po klasie `GenericServlet`.

Pierwsze żądanie klienta

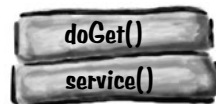


Wątek B

Kiedy do serwera WWW dociera pierwsze żądanie klienta, kontener uruchamia (lub znajduje w puli) wątek i wymusza wywołanie metody `service()` serwletu wskazanego w żądaniu.

Zazwyczaj NIE będziesz nadpisywał metody `service()`, zatem w większości przypadków wykonywana będzie metoda `service()` należąca do klasy bazowej `HttpServlet`. Metoda `service()` określa, której metody protokołu HTTP (GET, POST itp.) użyto w żądaniu, i wywołuje odpowiednią metodę serwletu (odpowiednio `doGet()` lub `doPost()`). Zdefiniowane w klasie `HttpServlet` metody `doGet()` i `doPost()` nie podejmują żadnych działań, zatem przynajmniej jedną z tych metod musimy nadpisać w kodzie naszego serwletu. Kiedy działanie metody `service()` się kończy, odpowiedni wątek jest zabijany lub zwracany do puli zarządzanej przez kontener.

Drugie żądanie klienta



Wątek C

Kiedy do serwera WWW dotrze drugie (i każde kolejne) żądanie klienta, kontener ponownie utworzy lub wyszuka w puli wątek i wymusi wywołanie metody `service()` należącej do odpowiedniego serwletu.

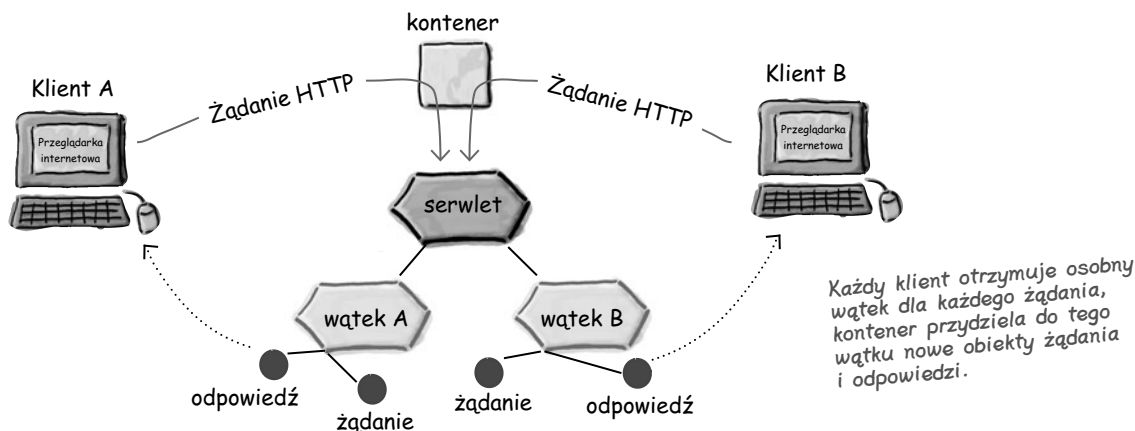
Oznacza to, że sekwencja metod `servlet()` i `doGet()` jest wykonywana za każdym razem, gdy serwer otrzymuje żądanie klienta. W dowolnym momencie istnieje co najmniej tyle wykonywalnych wątków, ile żądań klientów aktualnie obsługiwanych przez Twoją aplikację internetową — ich liczba może być ograniczana przez dostępne zasoby lub strategię (ustawienia konfiguracyjne) zastosowane w kontenerze (możesz np. korzystać z kontenera, oferującego możliwość określania maksymalnej liczby jednocześnie utrzymywanych wątków i — w razie wyczerpania puli wątków — mechanizm kolejki żądań oczekujących na swoją kolej).

Każde żądanie jest realizowane w osobnym wątku!

Być może miałeś okazję usłyszeć wyrażenie „każdy obiekt tego serwletu...”, takie określenie jest jednak *niepoprawne*. Nie istnieje wiele *obiektów* (egzemplarzy) żadnej klasy serwletu z wyjątkiem bardzo specyficznych przypadków (pracujących w rzadko stosowanym i niezalecanym modelu jednowątkowym), których w tej książce nie będziemy omawiali.

Kontener uruchamia wiele wątków, których zadaniem jest przetworzenie wielu żądań adresowanych do pojedynczego serwletu.

Każde żądanie klienta generuje także nową parę obiektów reprezentujących żądanie i odpowiedź.



Nie ma niemiłych pytań

P: Wciąż mam pewne wątpliwości... na powyższym rysunku przedstawiono dwóch klientów, z których każdy korzysta z własnego wątku. Co się stanie, jeśli *ten sam* klient wygeneruje wiele żądań? Czy kontener stworzy wówczas po jednym wątku dla każdego klienta czy po jednym wątku dla każdego żądania?

O: Po jednym wątku dla każdego żądania. Kontenera w ogóle nie interesuje, kto przysłał dane żądanie — każde otrzymane żądanie wiąże się z koniecznością stworzenia lub przydzielenia nowego wątku (stosu).

P: Jak to wszystko działa, kiedy kontener stosuje technikę łączenia w klastry i rozprasza aplikację internetową pomiędzy wiele wirtualnych maszyn Javy?

O: Wyobraź sobie, że przedstawiony powyżej rysunek dotyczy pojedynczej wirtualnej maszyny Javy i że taki sam rysunek można by sporządzić dla każdej wirtualnej maszyny systemu rozproszonego. W przypadku rozproszonej aplikacji internetowej musiałby istnieć jeden egzemplarz naszego serwletu w każdej wirtualnej maszynie Javy, ale pojedyncza wirtualna maszyna Javy nadal utrzymywałaby tylko jeden taki obiekt.

P: Zauważyłem, że klasa `HttpServlet` należy do innego pakietu niż klasa `GenericServlet`... ile wobec tego istnieje pakietów dla serwletów?

O: Wszystko, co wiąże się z serwletami (poza elementami technologii JSP), jest obsługiwane albo w pakiecie `javax.servlet`, albo w pakiecie `javax.servlet.http`. Nietrudno także opisać różnice pomiędzy tymi pakietami... kod związany z obsługą żądań protokołu HTTP należy do pakietu `javax.servlet.http`, całą resztę (a więc uniwersalne klasy i interfejsy serwletów) zakodowano w pakiecie `javax.servlet`. Technologii JSP poświęcimy kilka rozdziałów w dalszej części tej książki.

Etap pierwszy. Wczytywanie i inicjalizacja

Życie serwletu rozpoczyna się w momencie, w którym kontener (odpowiedzialny za realizację żądań klientów) odnajduje plik klasy serwletu. Klasa serwletu jest niemal zawsze odnajdywana już w chwili rozpoczęcia pracy kontenera (np. w momencie uruchomienia Tomcata). Uruchamiany kontener przegląda strukturę katalogów w poszukiwaniu wdrożonych aplikacji internetowych, po czym przystępuje do odnajdywania wszystkich plików klas serwletów, które składają się na znalezione aplikacje (proces odnajdywania serwletów omówimy bardziej szczegółowo w rozdziale poświęconym wdrażaniu aplikacji internetowych).

Samo *znalezienie* pliku klasy jest tylko pierwszym krokiem.

Wczytanie klasy jest drugim krokiem, którego realizacja ma miejsce albo podczas *uruchamiania kontenera*, albo w czasie *obsługi pierwszego żądania klienta*. Twój kontener może albo uzależniać tryb ładowania klas od ustawień konfiguracyjnych, albo wczytywać klasy w dowolnym wybranym przez siebie momencie. Niezależnie od tego, czy Twój kontener z góry przygotowuje serwlety aplikacji internetowych, czy wczytuje je na bieżąco, w odpowiedzi na potrzeby zgłaszane przez użytkownika, *metoda `service()` serwletu nigdy nie jest wywoływana przed pełną inicjalizacją serwletu*.

Twój serwlet zawsze musi zostać wczytany i zainicjalizowany ZANIM będzie mógł obsłużyć pierwsze żądania klientów.

Metoda `init()` zawsze kończy działanie przed pierwszym wywołaniem metody `service()`.



WYTEŻ UMYSŁ

Po co w ogóle istnieje metoda `init()`? Inaczej mówiąc, dlaczego sam *konstruktor* nie wystarczy do zainicjalizowania serwletu?

Jakiego rodzaju kod należy umieścić w ciele metody `init()`?

Wskazówka: Metoda `init()` otrzymuje za pośrednictwem argumentu referencję do pewnego obiektu. Jak myślisz, jaka jest rola tego argumentu metody `init()` i jak (lub do czego) mógłbyś tego argumentu użyć?

Inicjalizacja serwletu. Kiedy obiekt staje się serwletem



Początek procesu przejścia serwletu ze stanu *nie istnieje* w stan *zainicjalizowany* (który tak naprawdę oznacza *gotowość do obsługi żądań klientów*) wiąże się z wykonaniem konstruktora. Warto jednak pamiętać, że sam konstruktor tworzy jedynie *obiekt*, nie *serwlet*. Aby zostać pełnoprawnym serwletem, taki obiekt musi jeszcze otrzymać podstawowe atrybuty *serwletowości*.

Kiedy obiekt staje się serwletem, otrzymuje jednocześnie wszystkie unikatowe uprawnienia właściwe dla wszystkich serwletów, w tym możliwość wykorzystywania referencji do kontekstu `ServletContext`, za pośrednictwem którego serwlet może uzyskiwać potrzebne informacje z kontenera.

Po co w ogóle zajmujemy się szczegółami procesu inicjalizacji?

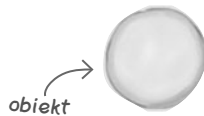
Ponieważ gdzieś pomiędzy konstruktorem a metodą `init()` serwlet znajduje się w stanie *serwletu Schrödingera*¹. Wykonywanie kodu inicjalizującego Twój serwlet (np. uzyskiwanie informacji o konfiguracji aplikacji internetowej lub wyszukiwanie referencji do innych części aplikacji) może zakończyć się **niepowodzeniem**, jeśli spróbujesz to zrobić zbyt *wcześnie* w życiu serwletu. O właściwym miejscu dla kodu inicjalizującego decyduje prosta zasada — wystarczy pamiętać, aby niczego nie umieszczać w konstruktorze serwletu!

Serwlet nie może zawierać żadnego kodu, którego wykonanie nie może czekać na wywołanie metody `init()`.

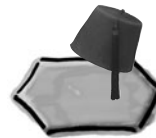
¹ Jeśli ostatnio zaniedbałeś swoją wiedzę z dziedziny mechaniki kwantowej, możesz w wyszukiwarce Google wpisać wyrażenie *Kot Schrödingera* (uwaga: miłośnicy zwierząt domowych powinni unikać tego zagadnienia). Mówiąc o *stanie Schrödingera*, mamy na myśli coś, co nie jest ani w pełni martwe, ani w pełni żywe — znajduje się w zagadkowym punkcie pomiędzy życiem a śmiercią.

Co tak naprawdę oznacza „bycie serwletem”?

Co się stanie, kiedy serwlet
przejdzie stąd...



tutaj?



oficjalny, zarejestrowany serwlet



**Nie należy mylić parametrów
obiektu ServletConfig
z parametrami obiektu
ServletContext!**

Nie mieliśmy zamiaru omawiać tych zagadnień w tym miejscu, tylko w **kolejnym** rozdziale; z drugiej strony, tak wiele osób myli ze sobą parametry obu obiektów, że warto już teraz położyć nacisk na **konieczność ich rozróżniania**.

Zacznijmy od analizy samych nazw:

W nazwie obiektu `ServletConfig` występuje słowo „config”, które reprezentuje „konfigurację” (ang. configuration). Tak rozumiana konfiguracja jest po prostu zbiorem wartości (ustawień) definiowanych dla serwletu w czasie wdrażania aplikacji (dla każdego serwletu istnieje osobny zbiór ustawień konfiguracyjnych). Konfiguracja obejmuje ustawienia, które z jednej strony mają być dostępne dla serwletu, ale z drugiej strony nie powinny być trwale kodowane w tym serwlecie — dotyczy to np. nazwy bazy danych. Parametry obiektu `ServletConfig` nie są zmieniane w całym okresie wykorzystywania serwletu po jego wdrożeniu. Aby je zmienić, będziesz musiał ponownie wdrożyć dany serwlet w swoim środowisku.

Obiekt `ServletContext` w rzeczywistości powinien nosić nazwę `AppContext` (niestety, nikt nie chce uwzględnić naszej opinii na ten temat), ponieważ dla każdej aplikacji internetowej istnieje tylko jeden taki obiekt (NIE jak w przypadku obiektu `ServletConfig`, gdzie mamy do czynienia z jednym obiektem dla każdego serwletu). Tak czy inaczej wszystkie te zagadnienia szczegółowo omówimy w następnym rozdziale — ten tekst należy traktować wyłącznie jak jego zapowiedź.

1 Obiekt ServletConfig

- Dla każdego serwletu istnieje jeden obiekt `ServletConfig`.
- Obiektu `ServletConfig` należy używać do przekazywania do serwletu informacji znanych już w czasie wdrażania aplikacji (np. nazw identyfikujących bazy danych lub komponenty EJB), których nie chcemy trwale kodować w samym serwlecie (np. w postaci jego parametrów inicjalizacji).
- Obiektu `ServletConfig` należy używać także do uzyskiwania dostępu do kontekstu serwletu (obiektu `ServletContext`).
- Parametry tego obiektu są konfigurowane w deskrypcorze wdrożenia (DD).

2 Obiekt ServletContext

- Dla każdej aplikacji internetowej istnieje dokładnie jeden obiekt `ServletContext` (należałoby więc nazywać te obiekty kontekstami aplikacji lub np. `AppContext`).
- Obiektu `ServletContext` należy używać do uzyskiwania dostępu do parametrów aplikacji internetowej (także skonfigurowanych w deskrypcorze wdrożenia).
- Obiekt `ServletContext` można również wykorzystywać w roli komunikatora, za pomocą którego możesz tworzyć komunikaty (nazywane atrybutami) udostępniane pozostałym częściom aplikacji internetowej (więcej informacji na temat tego typu rozwiązań znajdziesz w kolejnym rozdziale).
- Obiektu `ServletContext` należy używać także do uzyskiwania informacji o serwerze, włącznie z nazwą i wersją wykorzystywanego kontenera oraz wersją obsługiwanego interfejsu API.

RZECZYWISTYM zadaniem serwletu jest jednak obsługa żądań. Dopiero wtedy życie serwletu ma jakiś sens

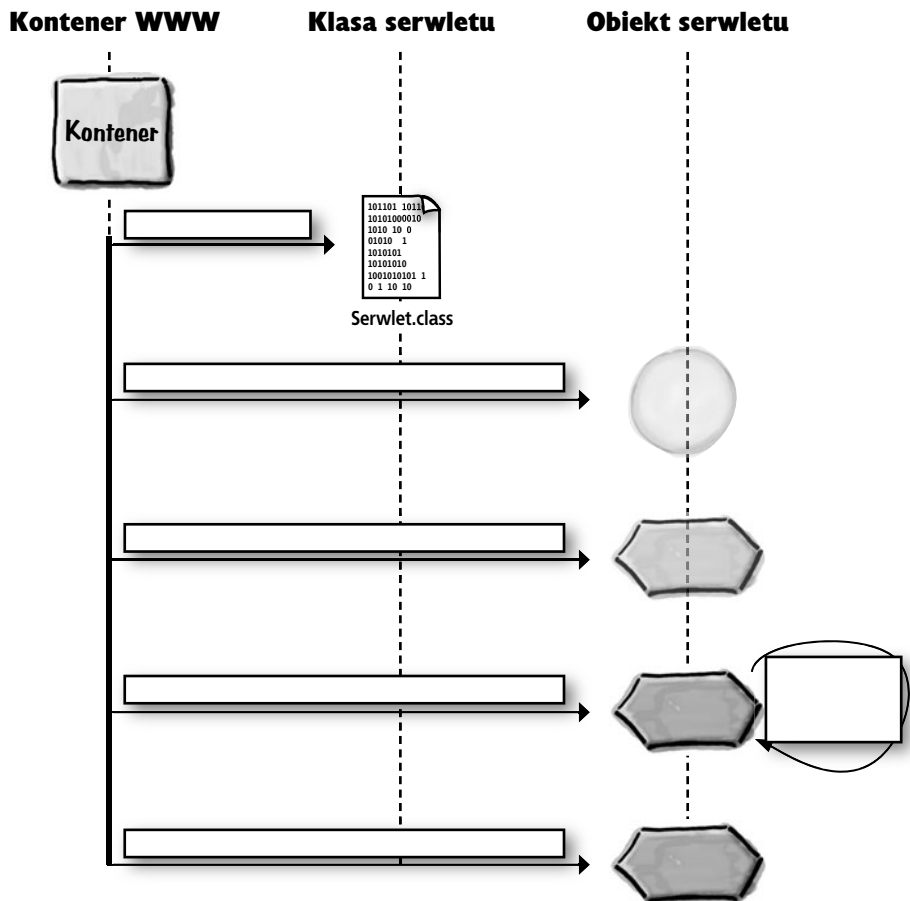
W kolejnym rozdziale zajmiemy się obiektami `ServletConfig` i `ServletContext`, na razie musimy jednak szczegółowo przeanalizować funkcjonowanie mechanizmu obsługującego żądania i odpowiedzi. Warto pamiętać, że obiekty `ServletConfig` i `ServletContext` istnieją tylko po to, by umożliwić naszemu serwletowi realizację Jego Najważniejszego Zadania — obsługę żądań klientów! Zanim przystąpimy do omawiania sposobu, w jaki nasze obiekty kontekstu i konfiguracji mogą pomóc w wykonywaniu tego zadania, musimy wrócić do podstaw przetwarzania żądań i odpowiedzi.

Wiemy już, że obiekty reprezentujące żądanie i odpowiedź są przekazywane do naszego serwletu w postaci argumentów metody `doGet()` lub `doPost()`, warto się jednak zastanowić, co *szczególnego* wynika z dostępu do tego typu obiektów. Co możemy zrobić z tymi obiektami i dlaczego przywiązujemy do nich tak dużą wagę?

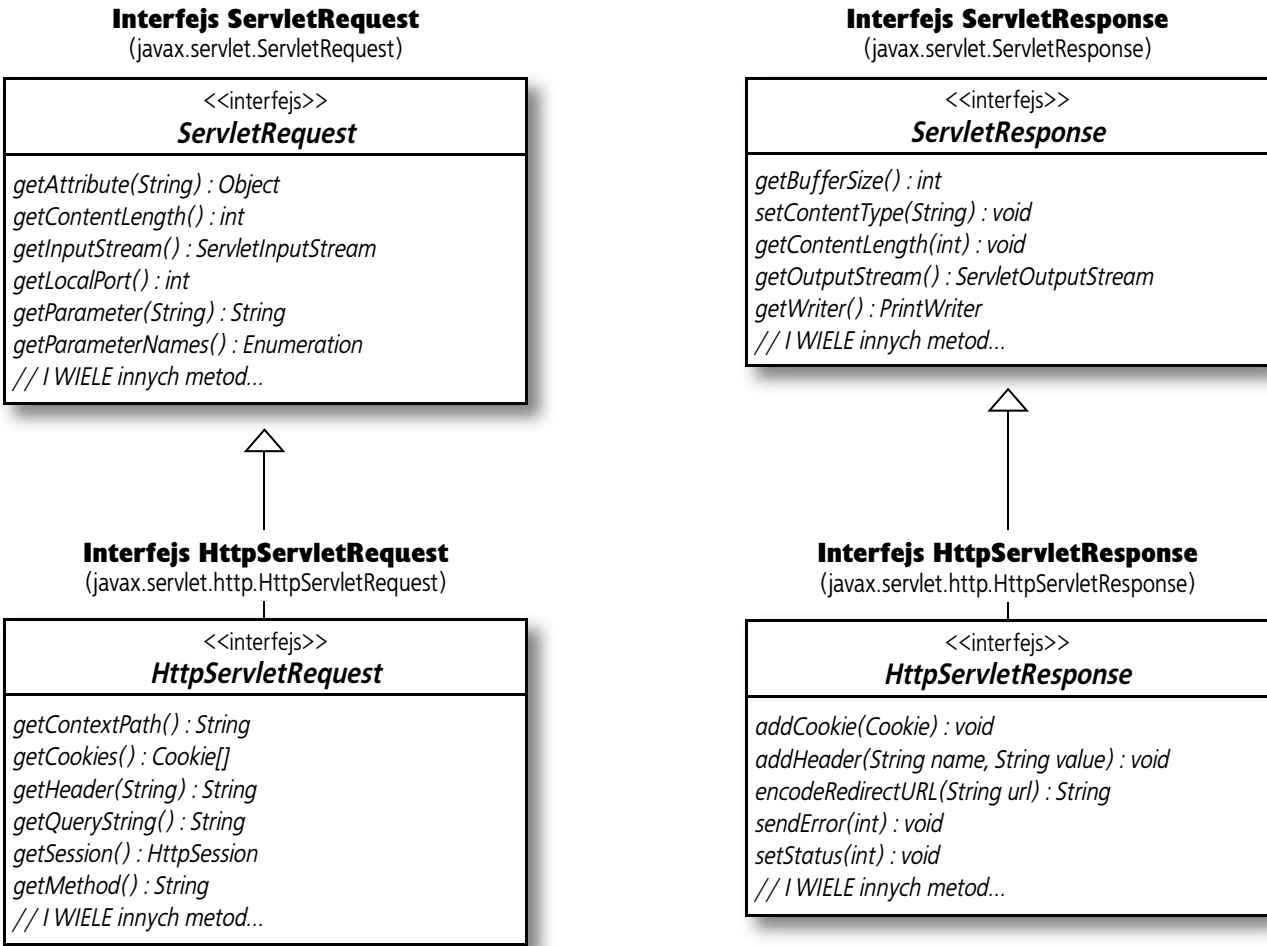
Zaostrz ołówek

Oznacź etykietami brakujące elementy (puste pola) przedstawionego diagramu cyklu życia serwletu (porównaj swoje odpowiedzi ze schematem przedstawionym we wcześniejszej części tego rozdziału).

Dodaj własne komentarze, które pozwolą Ci lepiej zapamiętać szczegóły tego schematu.



Żądanie i odpowiedź — kluczowe obiekty
będące także argumentami metody service()*



Metody interfejsu `HttpServletRequest` obsługują takie elementy żądań protokołu HTTP jak znaczniki kontekstu klienta (tzw. ciasteczka), nagłówki i sesje. `HttpServletRequest` interface... — Interfejs `HttpServletRequest` dodaje metody związane z protokołem HTTP, które Twój serwet wykorzystuje do zapewniania komunikacji pomiędzy klientem a serwerem.

Ta para interfejsów z pewnością ma związek z odpowiedzią... interfejs `HttpServletResponse` dodaje metody niezbędne podczas obsługi takich elementów protokołu HTTP jak błędy, ciasteczka czy nagłówki.

* Obiekty reprezentujące żądanie i odpowiedź są także argumentami *innych* nadpisywanych przez nas metod klasy bazowej `HttpServlet` — `doGet()`, `doPost()` itp.

Nie ma niemądrych pytań

P: Kto implementuje interfejsy `HttpServletRequest` i `HttpServletResponse`?
Czy odpowiednie klasy należą do interfejsu programowego API?

U: Pierwsza odpowiedź brzmi „kontener”; druga — „nie”. Odpowiednie klasy nie należą do interfejsu programowego API, ponieważ ich implementację pozostawiono producentom oprogramowania kontenerów. Niewątpliwie dobrą wiadomością dla Ciebie jest brak konieczności samodzielnego implementowania tych klas. Możesz przyjąć, że w momencie wywołania metody `service()` Twojego serwletu otrzymasz referencje do niezawodnych obiektów klas *implementujących* interfejsy `HttpServletRequest` i `HttpServletResponse`. W żadnym razie nie musisz się zajmować faktyczną implementacją nazwy ani typu klasy — Twoje zadanie sprowadza się do właściwego korzystania z funkcjonalności oferowanej przez obiekty implementujące interfejsy `HttpServletRequest` i `HttpServletResponse`.

Innymi słowy, do programisty serwletu należy jedynie znajomość *metod, które może wywoływać* za pośrednictwem obiektów przekazanych przez kontener w postaci części żądania! Tworzenie rzeczywistej klasy implementującej te metody nie należy do Twoich zadań — do obiektów reprezentujących żądanie i odpowiedź odwołujesz się *wyłącznie* za pośrednictwem odpowiedniego typu interfejsu.

P: Czy właściwie odczytałem przedstawione diagramy UML?
Czy wspomniane interfejsy rozszerzają interfejsy bazowe?

U: Tak. Pamiętaj, że interfejsy mogą tworzyć własne drzewa dziedziczenia. Jeśli jeden interfejs *rozszerza* inny interfejs (to wszystko, co *może* robić interfejs — interfejsy z natury rzeczy nie mogą *implementować* innych interfejsów), każda klasa implementująca ten interfejs musi jednocześnie implementować *wszystkie* metody zdefiniowane zarówno w danym interfejsie, jak i we *wszystkich* jego nadinterfejsach. Oznacza to, że każda klasa implementująca np. interfejs `HttpServletRequest` musi zawierać implementacje wszelkich metod, które zadeklarowano zarówno w tym interfejsie, jak i w rozszerzonym przez niego nadinterfejsie `ServletRequest`.

P: Nadal nie jest dla mnie jasne, dlaczego istnieje zarówno interfejs `GenericServlet`, interfejs `ServletRequest`, jak i interfejs `ServletResponse`? Skoro poza serwletami HTTP nikt nic nie robi, po co aż tyle tych interfejsów?

U: Nie powiedzieliśmy, że nikt nic nie robi. Można sobie wyobrazić, że ktoś gdzieś wykorzystuje model technologii serwletów bez protokołu HTTP. Chodzi nam wyłącznie o to, że nigdy nikogo takiego nie spotkaliśmy i nigdy o nikim takim nie słyszeliśmy. To wszystko. Model oparty na serwletach został zaprojektowany w sposób na tyle elastyczny, że istnieje możliwość komunikowania się z serwletami np. za pośrednictwem protokołu SMTP lub nawet protokołu opracowanego specjalnie z myślą o danej aplikacji internetowej. Warto jednak pamiętać, że omawiany interfejs programowy API ma wbudowaną wyłącznie obsługę protokołu HTTP i że właśnie z tego protokołu korzystają niemal wszyscy programiści serwletów.



Relax

Na egzaminie
nikt nie będzie od
Ciebie wymagał
opracowywania serwletów
innych niż serwlety HTTP.

Nikt nie oczekuje od Ciebie pełnej znajomości technik tworzenia i wykorzystywania serwletów z protokołem innym niż HTTP. Powinieneś jednak mieć pewne rozeznanie w sposobie funkcjonowania istniejącej hierarchii klas. **MUSISZ** na przykład wiedzieć, że interfejsy `HttpServletRequest` i `HttpServletResponse` rozszerzają odpowiednio interfejsy `ServletRequest` i `ServletResponse` oraz że większa część implementacji interfejsu `HttpServletRequest` faktycznie pochodzi z abstrakcyjnej klasy `GenericServlet`.

To wszystko — twórcy egzaminu zakładają, że jesteś programistą serwletów dziedziczącym po klasie `HttpServlet`.

Od metody żądania protokołu HTTP zależy, czy zostanie wywołana metoda doGet(), czy metoda doPost()

Jak zapewne pamiętasz, żądanie klienta zawsze zawiera konkretną metodę protokołu HTTP. Jeśli tą metodą jest GET, metoda service() wywoła metodę doGet(). Jeśli natomiast wskazaną metodą protokołu HTTP jest POST, metoda service() wywoła metodę doPost().

Cały czas
mówimy tylko o metodach
doGet() i doPost() tak jakby nie
istniały żadne inne... WIEM przecież,
że w sumie istnieje aż osiem metod
standardu HTTP 1.1.

9



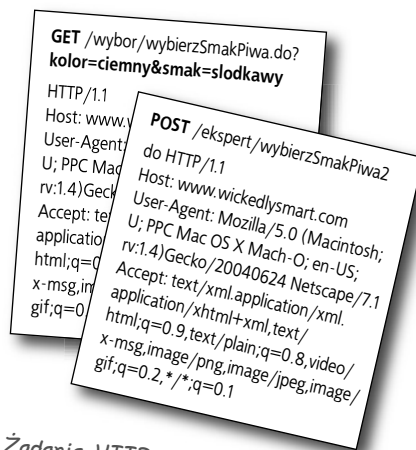
Prawdopodobnie nie będą Cię interesowały żadne metody protokołu HTTP poza najpopularniejszymi GET i POST

To prawda, poza GET i POST istnieją także *inne* metody protokołu HTTP 1.1. Protokół HTTP dodatkowo obsługuje takie metody jak HEAD, TRACE, OPTIONS, PUT, DELETE czy CONNECT.

Okazuje się, że dla aż siedmiu spośród wymienionych metod protokołu HTTP istnieją odpowiednie metody doXXX() w klasie HttpServlet, zatem poza wielokrotnie wspomnianymi metodami doGet() i doPost() mamy do dyspozycji metody doOptions(), doHead(), doTrace(), doPut() oraz doDelete(). Interfejs programowy API dla serwletów nie przewiduje mechanizmów obsługujących metodę doConnect(), zatem metoda ta nie jest częścią klasy HttpServlet.

O ile jednak wymienione metody protokołu HTTP mogą być interesujące np. dla programistów *serwerów WWW*, o tyle programiści *serwletów* bardzo rzadko korzystają z czegokolwiek innego niż popularne metody GET i POST.

W większości (być może nawet we *wszystkich*) procesów wytwarzania serwletów będziesz używał albo metody doGet() (dla prostych żądań), albo metody doPost() (do przyjmowania i przetwarzania danych z formularza), zatem nie będziesz nawet myślał o pozostałych dostępnych metodach.



Żądania HTTP

Jeśli pozostałe metody protokołu HTTP nie są dla mnie ważne... Z PEWNOŚCIĄ część pytań egzaminacyjnych będzie poświęcona właśnie tym metodom.



Przykład odpowiedzi dla żądania OPTIONS protokołu HTTP.



```
Http/1.1 200 OK
Server: Apache-Coyote/1.1
Date: Thu, 20 Apr 2004
16:20:00 GMT
Allow: OPTIONS, TRACE,
GET, HEAD, POST
Content-Length: 0
```

W rzeczywistości jedna lub kilka pozostałych metod protokołu HTTP może (na moment) pojawić się na egzaminie...

Jeśli przygotowujesz się do egzaminu, powinieneś potrafić prawidłowo rozpoznać wszystkie wymienione metody i choćby w skrócie wyjaśnić, do czego służą i co je różni. Nie trać jednak zbyt wiele czasu na omawiane tutaj zagadnienia — ich udział w pytaniach egzaminacyjnych będzie marginalny.

W świecie rzeczywistych serwletów interesują nas wyłącznie metody GET i POST

W świecie egzaminów powinniśmy nieco uwagi poświęcić także pozostałym metodom protokołu HTTP.

- GET** Żąda zwrócenia zasobu (pliku) znajdującego się pod podanym adresem URL.
- POST** Żąda od serwera *przyjęcia* informacji dołączonych do żądania i przekazania ich pod wskazany adres URL. Metodę POST można traktować jak rozszerzoną metodę GET... jak metodę GET z dodatkowymi informacjami wysyłanymi wraz z żądaniem.
- HEAD** Żąda zwrócenia wyłącznie części *nagłówkowej* tego, co zostałoby zwrócone w odpowiedzi na analogiczne żądanie GET. Metoda HEAD jest więc żądaniem GET pomijającym właściwe informacje w zwracanej komunikacji. Pozwala uzyskać informacje na temat wskazanego adresu URL bez konieczności zwracania *zawartości* generowanej odpowiedzi.
- TRACE** Żąda odesłania komunikatu żądania, dzięki czemu klient może się łatwo przekonać, co dotarło na drugi koniec — tego typu funkcjonalność wykorzystuje się najczęściej podczas testowania aplikacji i usuwania błędów.
- PUT** Żąda *umieszczenia* osadzonych informacji (ciała) pod wskazanym adresem URL.
- DELETE** Żąda *usunięcia* zasobu (pliku) znajdującego się pod wskazanym adresem URL.
- OPTIONS** Żąda *listy* metod protokołu HTTP, na które zasób znajdujący się pod wskazanym adresem URL może odpowiedzieć.
- CONNECT** Wnosi o nawiązanie *połączenia* celem tunelowania protokołów.

Różnica pomiędzy metodą GET a metodą POST

Żądanie **POST** zawiera ciało. To kluczowa różnica. Zarówno metoda GET, jak i metoda POST przewiduje możliwość przesyłania parametrów, ale w przypadku metody GET dane zawarte w tych parametrach podlegają takim samym ograniczeniom jak wiersz żądania.

Wiersz żądania.

Metoda protokołu HTTP.

Ścieżka do zasobu na serwerze WWW.

W żądaniu GET parametry (jeśli w ogóle istnieją) są dotychczas do adresu URL tego żądania.

Żądana przez przeglądarkę wersja protokołu.

GET /wybor/wybierzSmakPiwa.do?kolor=ciemny&smak=słodkawy HTTP/1.1

Nagłówki żądania.

Host: www.wickedlysmart.com

User-Agent: Mozilla/5.0 (Macintosh; U; PPC Mac OS X Mach-O; en-US; rv:1.4)Gecko/20040624 Netscape/7.1

Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,video/x-msg,image/png,image/jpeg,image/gif;q=0.2,*/*;q=0.1

Accept-Language: en-us,en;q=0.5

Accept-Encoding: gzip,deflate

Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7

Keep-Alive: 300

Connection: keep-alive

BRAK ciała... tylko informacje nagłówka.

Wiersz żądania.

Metoda protokołu HTTP.

Ścieżka.

BRAK parametrów żądania w tym miejscu.

Protokół.

POST /ekspert/wybierzSmakPiwa.do HTTP/1.1

Nagłówki żądania.

Host: www.wickedlysmart.com

User-Agent: Mozilla/5.0 (Macintosh; U; PPC Mac OS X Mach-O; en-US; rv:1.4)Gecko/20040624 Netscape/7.1

Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,video/x-msg,image/png,image/jpeg,image/gif;q=0.2,*/*;q=0.1

Accept-Language: en-us,en;q=0.5

Accept-Encoding: gzip,deflate

Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7

Keep-Alive: 300

Connection: keep-alive

Ciało komunikatu, nazywane także „ładunkiem”.

kolor=ciemny&smak=słodkawy

Tym razem parametry umieszczono w ciele komunikatu; w ten sposób udało nam się uniknąć ograniczeń typowych dla metody GET, która wymaga przekazywania parametrów w wierszu żądania.

Wygląda na to, że jedyne różnice dzielące metody GET i POST sprowadzają się do maksymalnego rozmiaru przesyłanych danych parametrów.

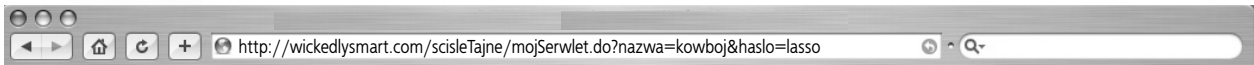


Nie, nie chodzi tylko o rozmiar

Pamiętasz, jak w pierwszym rozdziale wspominaliśmy o pozostałych cechach metody GET?

Kiedy używasz metody GET, dane parametrów są wyświetlane w polu adresu okna przeglądarki internetowej, bezpośrednio za właściwym adresem URL (obie składowe tego adresu są oddzielone jedynie znakiem zapytania). Nietrudno wyobrazić sobie scenariusz, w którym programista nie chce, by parametry były znane użytkownikowi.

Można więc przyjąć, że kwestie bezpieczeństwa stanowią kolejną różnicę pomiędzy obiema metodami.



Innym elementem decydującym o odmienności metod GET i POST jest możliwość dodawania żądanych stron do listy ulubionych adresów użytkownika końcowego. O ile do takiej listy można dodawać żądania GET, o tyle w przypadku żądań POST podobne działania są niemożliwe. Może to być szczególnie istotne w przypadku strony, na której użytkownicy mogą np. określać kryteria przeszukiwania dostępnych zasobów. Użytkownicy mogą być zainteresowani powrotem na taką stronę po tygodniu i wykonaniem tej samej operacji przeszukiwania ponownie, aby sprawdzić, czy na przykład na serwerze nie pojawiły się nowe dane.

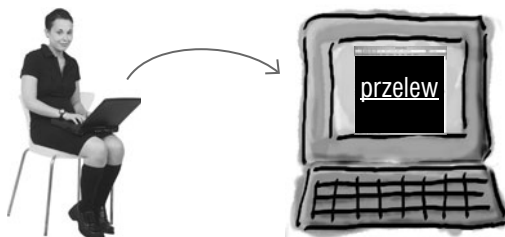
Jednak *oprócz* rozmiaru, bezpieczeństwa i listy ulubionych stron internetowych, istnieje jeszcze inna kluczowa cecha odróżniająca metodę GET od metody POST — jest to sposób, w jaki obie metody mają *w założeniu* być wykorzystywane. Metodę GET zaprojektowano z myślą o *uzyskiwaniu* dostępu do żądanych zasobów. Kropka. Jej celem jest otrzymywanie. Możesz oczywiście wykorzystywać dodatkowe parametry, które pomagają w określaniu tego, co serwer powinien Ci odesłać, jednak z reguły żądania GET nie powinny być źródłem zmian po stronie serwera! Metoda POST ma w założeniu służyć do *przesyłania danych przeznaczonych do przetworzenia*. Dane dołączone do żądania POST mogą co prawda mieć postać prostych parametrów zapytania używanych do precyzyjnego określania zasobów interesujących użytkownika (a więc podobnie jak w przypadku metody GET), jednak prawdziwym powodem zaprojektowania tej metody jest *aktualizowanie* danych po stronie serwera. Należy przyjąć, że dane z ciała żądania POST mają na celu *wprowadzenie jakichś zmian na serwerze*.

W naturalny sposób doszliśmy do kolejnego zagadnienia... czy dane żądanie może być *powtarzalne* (*idempotentne*). Jeśli nie, możemy popaść w tarapaty, których nie rozwiąże mała, niebieska pigułka. Jeśli po raz pierwszy zetknąłeś się z pojęciem *powtarzalności* w świecie aplikacji internetowych, czytaj dalej...

Historia pewnego niepowtarzalnego żądania

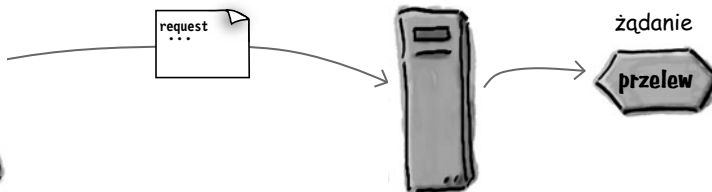
Diane ma problem. Próbuje znaleźć w księgarni internetowej książkę z serii *Head First* poświęconą dziwiarstwu; nie wie jednak, że prace nad witryną internetową Wickedly Smart wciąż nie zostały ukończone. Diane nie ma zbyt dużo pieniędzy — uzbierała na swoim koncie dokładnie tyle, ile będzie potrzebne na *jedną* książkę. Diane rozważała co prawda zakup książki bezpośrednio w księgarni internetowej Amazon lub na witrynie wydawnictwa, jednak ostatecznie zdecydowała się na zakup egzemplarza z *autografem* autora, dostępnego wyłącznie na witrynie Wickedly Smart. Tego wyboru już wkrótce będzie żałowała...

- ❶ Diane klika przycisk przelew (informacje o numerze konta bankowego przesłała w jednym z wcześniejszych formularzy).



Przeglądarka wysyła do serwera żądanie HTTP z informacją o zamówieniu książki i numerem klienta przypisanym wcześniej Diane.

Kontener wysyła żądanie do przetworzenia przez serwet przelew.



Serwer (kontener) WWW księgarni internetowej Wickedly Smart.

- ❷ Serwet dokonuje elektronicznego obciążenia konta bankowego Diane.



obciąża



Zdalny serwer
kont bankowych

- ❸ Serwet aktualizuje bazę danych (usuwa jeden egzemplarz zamawianej książki z bazy danych magazynu, tworzy nowy rekord zamówienia itp.).



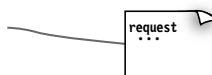
aktualizuje



- ❹ Serwet NIE odesłał jednoznacznej odpowiedzi, a Diane cały czas widzi w oknie przeglądarki tę samą stronę koszyka z zakupami i nabiera pewnych podejrzeń...



Przeglądarka wysyła do serwera żądanie HTTP z informacją o zamówieniu książki i numerem klienta przypisanym wcześniej Diane.

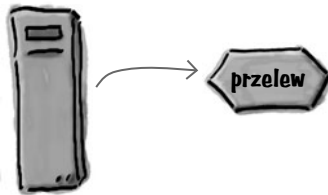


żądanie

Serwer (kontener) WWW księgarni internetowej Wickedly Smart.

Dalszy ciąg naszej historii...

- 5 Kontener przekazuje żądanie do przetworzenia przez serwet przelew.

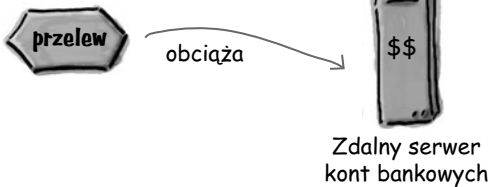


Serwer (kontener) WWW księgarni internetowej Wickedly Smart.

- 6 Serwet nie widzi problemu w tym, że Diane kupuje tę samą książkę, którą kupiła przed chwilą.

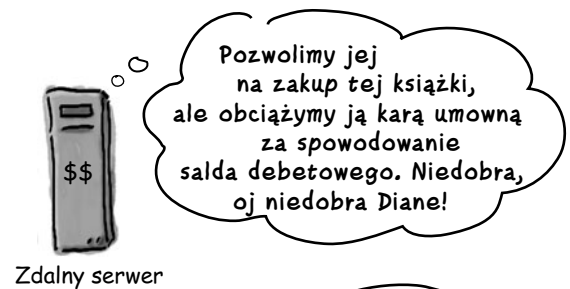


- 7 Serwet dokonuje elektronicznego obciążenia konta bankowego Diane.



Zdalny serwer kont bankowych

- 8 Bank obsługujący konto Diane przyjmuje polecenie przelewu i obciąża ją surową karą za doprowadzenie do salda debetowego.



- 9 Diane ostatecznie przechodzi do strony stanu zamówienia i przekonuje się, że wysłała aż DWA zamówienia dotyczące tej samej książki...



- 10 Halo, bank? Ten głupi programista księgarni internetowej wszystko pomylił...





Zaostrz ołówek

Która z metod protokołu HTTP jest (lub powinna być) Twoim zdaniem powtarzalna (idempotentna)? Swoją odpowiedź możesz oprzeć na samym znaczeniu słowa powtarzalność i (lub) analizie przedstawionego przed chwilą przykładu podwójnego zakupu dokonanego przez Diane. Prawidłowe odpowiedzi przedstawiono na dole tej strony.

- ☐ GET
- ☐ POST
- ☐ PUT
- ☐ HEAD

(Celowo pomieiliśmy metodę CONNECT, ponieważ nie jest ona obsługiwana w klasie `HttpServlet`.)



WYTEŻ UMYŚL

Co było źródłem problemów związanych z transakcją Diane?

(Zaistniałe nieporozumienie nie było efektem tylko JEDNEJ usterki — prawdopodobnie istnieje kilka problemów, które muszą zostać wyeliminowane przez programistę.)

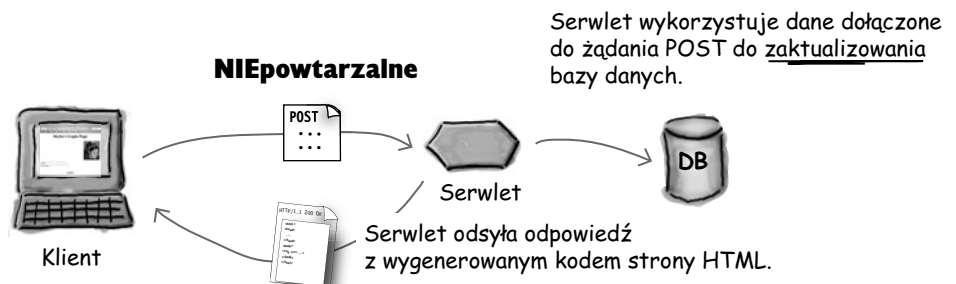
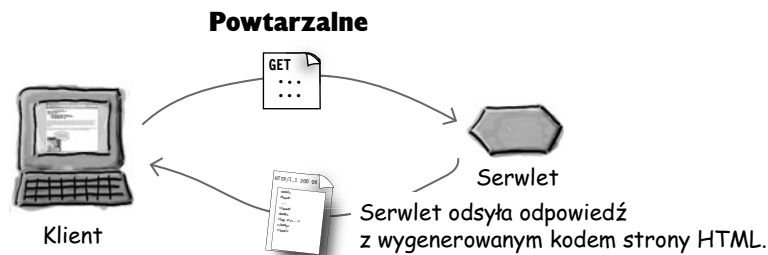
Jakie są możliwości ograniczania przez programistę ryzyka związanego z podobnymi niedociągnięciami w kodzie aplikacji?

(Wskazówka: nie wszystkie rozwiązania w tym obszarze muszą mieć charakter działań programistycznych).

Specyfikacja standardu HTTP 1.1 deklaruje metody GET, HEAD i PUT jako powtarzalne, chociaż teoretycznie ISTNIEJE możliwość samodzielnego napisania niepowtarzalnej metody doGet() (nie należy jednak z tej możliwości korzystać). Specyfikacja HTTP 1.1 nie przewiduje powtarzalności metody POST.



Bycie powtarzalnym ma swoje PLUSY. Oznacza bowiem, że możesz wykonywać te same czynności wielokrotnie bez niepożądanych efektów ubocznych!



Żądanie POST nie jest idempotentne

Żądanie GET protokołu HTTP ma w założeniu służyć do *uzyskiwania* dostępu do zasobów, nie do *zmieniania* czegokolwiek na serwerze. Żądanie GET jest więc (zgodnie z definicją i specyfikacją standardu HTTP) powtarzalne (idempotentne). Identyczne żądanie tego typu można wykonać więcej niż raz bez obaw o jakiegokolwiek niekorzystne efekty uboczne.

Żądanie POST *nie* jest powtarzalne — dane osadzone w ciele żądania POST mogą być kierowane do nieodwracalnej transakcji. Oznacza to, że musisz bardzo uważać podczas opracowywania funkcjonalności swojej metody doPost().

Żądanie GET jest, żądanie POST nie jest idempotentne. To, czy logika Twojej aplikacji internetowej w każdej sytuacji będzie przygotowana do właściwego obsłużenia scenariusza podobnego do zakupów Diane (gdzie do serwera dociera więcej niż jedno identyczne żądanie POST), zależy tylko od Ciebie.

Co może
mnie powstrzymać przed
stosowaniem parametrów metody
GET do aktualizowania danych
składowanych na serwerze?



**W standardzie HTTP 1.1
metoda GET zawsze jest
uważana za powtarzalną...**

...nawet jeśli na egzaminie
występują żądania GET z parametrami prowadzącymi
do rozmaitych skutków ubocznych po stronie serwera!
Innymi słowy, metoda GET **jest idempotentna**
zgodnie ze specyfikacją protokołu HTTP. Nie
ma jednak żadnych ograniczeń, które mogłyby Cię
odwieść od implementowania nieidempotentnych
metod doGet() w Twoich serwletach. Generowane
przez klienta żądanie GET zawsze jest traktowane
jak działanie powtarzalne, niezależnie od tego,
czy zaimplementowany przez CIEBIE mechanizm
przetwarzania tych danych wiąże się z jakimiś
skutkami ubocznymi. Zawsze musisz mieć na uwadze
istotną różnicę pomiędzy metodą GET protokołu HTTP
a metodą doGet() Twojego serwletu.

Uwaga: istnieje wiele różnych zastosowań słowa „idempotentny”; w tej książce używamy tego słowa wyłącznie w odniesieniu do protokołu HTTP i technologii serwletów — żądanie „idempotentne” to takie, które może być wykonywane dwukrotnie bez negatywnych konsekwencji dla serwera. *Nie* twierdzimy natomiast, że „idempotencja” oznacza zwracanie takich samych odpowiedzi na następujące po sobie identyczne żądania; NIE mówimy także, że przetwarzanie tych żądań nie prowadzi do żadnych skutków ubocznych.

Co sprawia, że przeglądarka wysyła albo żądanie GET, albo żądanie POST?

GET

Proste hipertącze zawsze reprezentuje żądanie GET.

Kliknij tutaj

POST

Jeśli OKREŚLISZ wprost: method="POST", o dziwo zostanie użyta metoda POST.

<form method="POST" action="WybierzPiwo.do">

Wybierz właściwości piwa<p>

Kolor:

<select name="kolor" size="1">

<option>jasny

<option>bursztynowy

<option>brązowy

<option>ciemny

</select>

<center>

<input type="SUBMIT">

</center>

</form>

Kiedy użytkownik kliknie przycisk Wyślij zapytanie, parametry zostaną przesłane w ciele żądania POST. W tym przypadku będzie istniał tylko jeden taki parametr (nazwany „kolor”) reprezentujący wybrany przez użytkownika kolor piwa (jasny, bursztynowy, brązowy lub ciemny).

Co będzie, jeśli w znaczniku < form > NIE określimy, że interesuje nas metoda POST (method="POST")?

<form ✓ action="WybierzPiwo.do">

Wybierz właściwości piwa<p>

Kolor:

<select name="kolor" size="1">

<option>jasny

<option>bursztynowy

<option>brązowy

<option>ciemny

</select>

<center>

<input type="SUBMIT">

</center>

</form>

Tym razem nie użyliśmy wyrażenia method="POST".

Co TERAZ stanie się z parametrami, jeśli użytkownik kliknie przycisk Wyślij zapytanie (przecież formularz nie zawiera już wyrażenia method="POST")?

POST NIE jest metodą domyślną!

Jeśli w znaczniku `<form>` nie umieścisz atrybutu `method="POST"`, zostanie zastosowane domyślne żądanie GET protokołu HTTP. Oznacza to, że przeglądarka wyśle Twoje parametry w ramach nagłówka żądania, ale nie to jest naszym zasadniczym problemem. Jeśli bowiem do serwera dotrze żądanie GET, a w swoim serwlecie nie zdefiniujesz metody `doGet()`, wpadniesz w poważne tarapaty dopiero podczas wykonywania tak skonstruowanej aplikacji internetowej.

Jeśli zrobisz to:

Brak atrybutu "method=POST" w formularzu HTML.

`<form action="WybierzPiwo.do">`

Po czym zrobisz to:

```
public class WyborPiwa extends HttpServlet {  
  
    public void doPost(HttpServletRequest request, HttpServletResponse response)  
        throws IOException, ServletException {  
        // tutaj powinien się znajdować Twój kod  
    }  
}
```

Brak metody doGet() w serwlecie.

Otrzymasz to:

BŁĄD! Jeśli Twój formularz HTML stosuje metodę GET zamiast metody POST, koniecznie MUSISZ zdefiniować w swojej klasie serwletu metodę doGet(). Domyślną metodą formularzy HTML jest GET.

P: Co powinienem zrobić, jeśli chcę w pojedynczym serwlecie obsługiwać zarówno żądania GET, jak i żądania POST?

U: Programiści, którzy chcą obsługiwać obie metody protokołu HTTP, umieszczają zwykle odpowiednią logikę w ciele metody `doGet()`, a jeśli tego wymaga sytuacja — konstruują metodę

`doPost()` w taki sposób, aby delegowała wywołanie do metody `doGet()`:

```
public void doPost(...)  
    throws ... {  
    doGet(request, response);  
}
```

Przesyłanie i wykorzystywanie pojedynczego parametru

Formularz HTML

```
<form method="POST" action="WybierzPiwo.do">
  Wybierz właściwości piwa<p>
  Kolor:
  <select name="kolor" size="1">
    <option>jasny
    <option>bursztynowy
    <option>brązowy
    <option>ciemny
  </select>
  <center>
    <input type="SUBMIT">
  </center>
</form>
```

Przeglądarka wyśle w ciełe żądania jedną z czterech dostępnych opcji parametru "kolor". Może to być np. wartość "kolor=bursztynowy".

Żądanie POST protokołu HTTP

POST /wybor/wybierzPiwo.do HTTP/1.1

Host: www.wickedlysmart.com

User-Agent: Mozilla/5.0 (Macintosh; U; PPC Mac OS X Mach-O; en-US; rv:1.4)Gecko/20040624 Netscape/7.1

Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,video/x-msg,image/png,image/jpeg,image/gif;q=0.2,*/*;q=0.1

Accept-Language: en-us,en;q=0.5

Accept-Encoding: gzip,deflate

Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7

Keep-Alive: 300

Connection: keep-alive

Pamiętaj, że to przeglądarka generuje żądanie, zatem nie musisz się martwić o jego tworzenie; w tym miejscu przedstawiliśmy możliwy kształt takiego żądania w formie, w jakiej dociera do serwera WWW...

kolor=ciemny

Klasa serwletu

```
public void doPost(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException {
    String kolorParam = request.getParameter("kolor");
    // tutaj jest Twój niesamowity kod...
}
```

(W prezentowanym przykładzie tańcuch kolorParam ma wartość "ciemny".)

Ten tańcuch odpowiada nazwie atrybutu formularza HTML.

Przesyłanie i wykorzystywanie DWÓCH parametrów

Formularz HTML

```
<form method="POST" action="WybierzPiwo.do">
  Wybierz właściwości piwa<p>
  Kolor:
  <select name="kolor" size="1">
    <option>jasny
    <option>bursztynowy
    <option>brązowy
    <option>ciemny
  </select>
  Moc:
  <select name="moc" size="1">
    <option>lekkie
    <option>ciężkie
    <option>mocne
  </select>
  <center>
    <input type="SUBMIT">
  </center>
</form>
```

Przeglądarka wyśle w ciele żądania jedną z czterech dostępnych opcji przypisanych do parametru nazwanego „kolor”.

Przeglądarka wyśle w ciele żądania jedną z trzech dostępnych opcji przypisanych do parametru nazwanego „body”.

Żądanie POST protokołu HTTP

```
POST /wybor/wybierzPiwo.do HTTP/1.1
Host: www.wickedlysmart.com
User-Agent: Mozilla/5.0 (Macintosh; U; PPC Mac OS X Mach-O; en-US; rv:1.4)Gecko/20040624 Netscape/7.1
Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,video/x-mpeg;q=0.3,image/jpeg;q=0.1,*/*;q=0.0
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300

Connection: keep-alive
```

kolor=ciemny&moc=mocne

Żądanie POST zawiera teraz oba parametry (oddzielone znakiem &).

Klasa serwletu

```
public void doPost(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException {
    String kolorParam = request.getParameter("kolor");
    String mocParam = request.getParameter("moc");
    // jeszcze więcej kodu w tym miejscu...
}
```

Zmienna łańcuchowa kolorParam zawiera teraz wartość „ciemny”, natomiast zmienna mocParam reprezentuje wartość „mocne”.



Dla pojedynczego parametru może istnieć wiele wartości! Oznacza to, że będziemy potrzebowali metody `getParameterValues()`, która zwróci tablicę tych wartości (zamiast metody `getParameter()` zwracającej pojedynczy łańcuch).

Niektóre typy komponentów formularza HTML (np. zbiór pól wyboru) mogą reprezentować więcej niż jedną wartość. Oznacza to, że pojedynczy parametr (w poniższym przykładzie nazwany "sizes") będzie miał wiele wartości w zależności od wyboru dokonanego przez użytkownika (liczby zaznaczonych pól wyboru). Formularz, w którym użytkownik może wybrać więcej niż jedną pojemność butelki lub puszki piwa (np. aby określić, że jest zainteresowany WSZYSTKIMI rozmiarami), może mieć następującą postać:

```
<form method="POST" action="WybierzPiwo.do">
  Wybierz właściwości piwa<p>
  Rozmiary puszek lub butelek: <p>
    <input type="checkbox" name="sizes" value="330ml">330 ml<br>
    <input type="checkbox" name="sizes" value="500ml">500 ml<br>
    <input type="checkbox" name="sizes" value="700ml">700 ml<br>
  <br><br>
  <center>
    <input type="SUBMIT">
  </center>
</form>
```

W kodzie naszego serwletu użyjemy metody `getParameterValues()`, która zwróci tabelę wartości:

```
String one = request.getParameterValues("sizes")[0];
```

```
String [] sizes = request.getParameterValues("sizes");
```

Gdybyśmy chcieli przejrzeć kolejne elementy uzyskanej w ten sposób tablicy (w celach testowych lub tylko dla zabawy), powinniśmy użyć pętli w następującej postaci:

```
String [] sizes = request.getParameterValues("sizes");
for (int x=0; x < sizes.length; x++) {
    out.println("<br>rozmiar: " + sizes[x]);
}
```

(przyjmijmy, że `out` jest obiektem klasy `PrintWriter` uzyskanym z obiektu odpowiedzi)

Co poza parametrami można odczytać z obiektu żądania?

Interfejsy `ServletRequest` i `HttpServletRequest` oferują mnóstwo metod, które możesz wywoływać, ale których nie musisz koniecznie pamiętać. Z drugiej strony, z *pewnością* warto się uważnie przyjrzeć pełnemu API interfejsów `javax.servlet.ServletRequest` i `javax.servlet.http.HttpServletRequest`. `HttpServletRequest` — w tym rozdziale wspominamy tylko o tych metodach, które programiści serwetów stosują najczęściej (i które mogą się pojawić w pytaniach egzaminacyjnych).

W praktyce będziesz miał przyjemność (lub, w zależności od Twojego punktu widzenia, nieprzyjemność) stosowania nieco ponad 15% interfejsu API żądań. *Nie martw się, jeśli nie dysponujesz jeszcze wystarczającą wiedzą na temat technik wykorzystywania każdego z elementów tego interfejsu*, z analizą części z nich (w szczególności tych związanych ze znacznikami kontekstu klienta, tzw. ciasteczkami) będziesz miał okazję zapoznać się w dalszej części tej książki.

Informacje o platformie i przeglądarce klienta

```
String client = request.getHeader("User-Agent");
```

Ciasteczka związane z danym żądaniem

```
Cookie[] cookies = request.getCookies();
```

Sesja związana z danym klientem

```
HttpSession session = request.getSession();
```

Użyta w żądaniu metoda protokołu HTTP

```
String theMethod = request.getMethod();
```

Strumień wejściowy z danego żądania

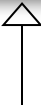
```
InputStream input = request.getInputStream();
```

Interfejs `ServletRequest` (`javax.servlet.ServletRequest`)

<<interfejs>>

ServletRequest

`getAttribute(String)`
`getContentTypeLength()`
`getInputStream()`
`getLocalPort()`
`getRemotePort()`
`getServerPort()`
`getParameter(String)`
`getParameterValues(String)`
`getParameterNames()`
// I WIELE innych metod...



Interfejs `HttpServletRequest` (`javax.servlet.http.HttpServletRequest`)

<<interfejs>>

HttpServletRequest

`getContextPath()`
`getCookies()`
`getHeader(String)`
`getIntHeader(String)`
`getMethod()`
`getQueryString()`
`getSession()`
// I WIELE innych metod...

Nie ma niemądrych pytań

P: Po co miałbym *kiedykolwiek* używać obiektu `InputStream` zawartego w żądaniu?

U: Wraz z żądaniem GET otrzymujemy wyłącznie dane zawarte w jego nagłówku. Innymi słowy, nie mamy do czynienia z ciałem żądania. Okazuje się JEDNAK, że zupełnie inaczej jest w przypadku żądania POST protokołu HTTP, gdzie oprócz nagłówka otrzymujemy ciało. Przez większość czasu będziemy zainteresowani właśnie wyodrębnianiem wartości parametrów z tego ciała (np. "color=ciemny") za pomocą prostej metody `request.getParameter()`, ale warto pamiętać, że takie wartości mogą być całkiem duże. Istnieje też możliwość stworzenia serwletu przetwarzającego żądania w formacie generowanym i zrozumiałym dla komputera, gdzie ciało zawiera dane tekstowe lub binarne wymagające dalszego przetworzenia. W takim przypadku należy użyć odpowiednio metody `getReader()` lub `getInputStream()`. Strumienie zwracane przez te metody zawierają wyłącznie ciało żądania protokołu HTTP, nigdy nagłówki tego żądania.

P: Jaka jest różnica pomiędzy metodami `getHeader()` i `getIntHeader()`? Z tego co wiem, nagłówki żądań HTTP zawsze są łańcuchami! Skoro nawet metoda `getIntHeader()` otrzymuje na wejściu łańcuch reprezentujący nazwę nagłówka, co właściwie oznacza wyraz *Int* w nazwie tej metody?

U: Nagłówki reprezentują zarówno nazwy (np. "User-Agent" czy "Host"), jak i wartości (odpowiednio "Mozilla/5.0 (Macintosh; U; PPC Mac OS X Mach-O; en-US; rv:1.4) Gecko/20040624 Netscape/7.1" i "www.wickedlysmart.com"). Co prawda wartości przesyłane w nagłówkach żądań zawsze mają postać łańcuchów, jednak niektóre łańcuchy reprezentują pewne liczby. Przykładowo, nagłówek "Content-Length" zwraca liczbę bajtów zawartych w ciele komunikatu; nagłówek "Max-Forwards" zwraca liczbę całkowitą określającą maksymalną liczbę routerów na drodze żądania (możemy wykorzystać ten nagłówek do śledzenia trasy żądania, o którym sądzimy, że mógł utknąć w jakiejś pętli).

Wartość nagłówka "Max-Forwards" możemy łatwo uzyskać za pomocą metody `getHeader()`:

```
String forwards = request.getHeader("Max-Forwards");
```

```
int forwardsNum = Integer.parseInt(forwards);
```

Takie rozwiązanie działa bez zarzutu. Skoro jednak *wiemy*, że wartość tego nagłówka powinna być reprezentowana w postaci liczby całkowitej, możemy *dla wygody* od razu użyć metody `getIntHeader()`, dzięki której unikniemy dodatkowego kroku konwersji łańcucha na liczbę całkowitą:

```
int forwardsNum = request.getIntHeader("Max-Forwards");
```



Oglądaj to!

Mylą mi się metody `getServerPort()`, `getLocalPort()` i `getRemotePort()`!

Znaczenie metody `getServerPort()` powinno być oczywiste... przynajmniej do momentu, w którym zapytasz, co oznacza metoda `getLocalPort()`. Przejdźmy więc do wyjaśniania pierwszych niejasności — metody `getRemotePort()`. Twoje pierwsze pytanie powinno brzmieć: „zdalny dla kogo?”. Ponieważ w tym przypadku to pytanie zadaje serwer, zdalny port musi się odnosić do portu po stronie KLIENTA. Klient jest z punktu widzenia serwera zdalnym węzłem, zatem metoda `getRemotePort()` musi oznaczać „zwróć port klienta”. Innymi słowy, metoda zwraca numer portu, z którego klient wysłał dane żądanie. Pamiętaj, że jeśli jesteś serwerem, określenie zdalny (ang. *remote*) dotyczy klienta.

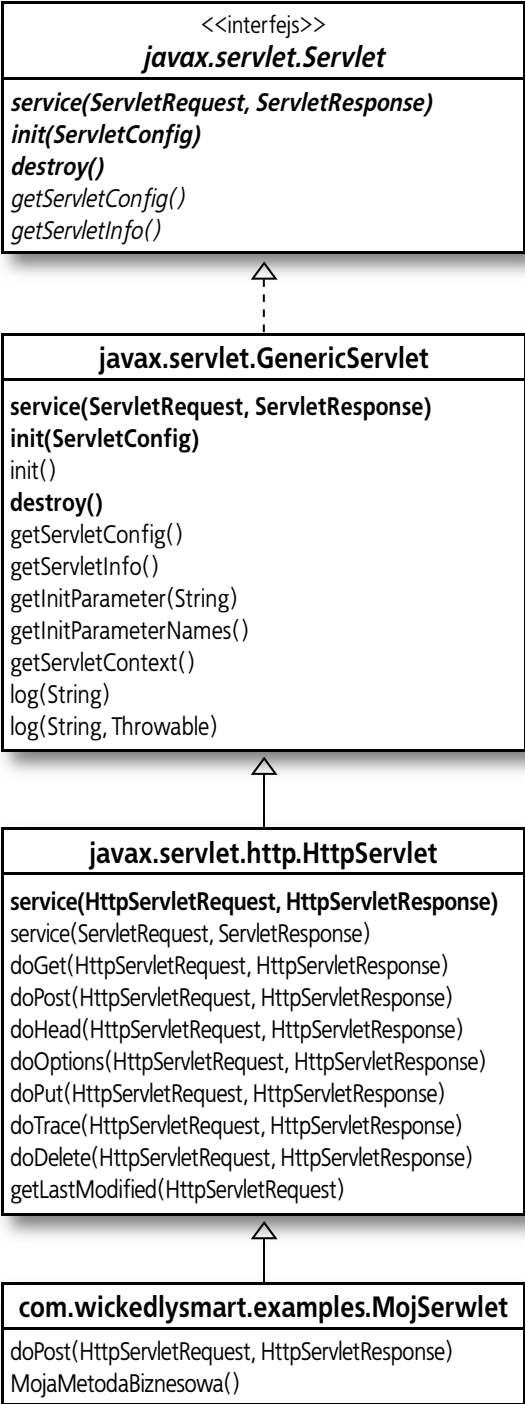
Różnica pomiędzy metodą `getLocalPort()` a metodą `getServerPort()` jest bardziej subtelna — metoda `getServerPort()` odpowiada na pytanie: „na który port dane żądanie zostało WYŚŁANE?”, natomiast metoda `getLocalPort()` odpowiada na pytanie: „na który port dane żądania ostatecznie DOTARŁY?”. Tak, istnieje pewna różnica, ponieważ choć żądania są wysyłane na pojedynczy port (na którym nasłuchuje nasz serwer), odbierający je serwer przydziela inny port lokalny dla każdego wątku serwletu, dzięki czemu aplikacja może jednocześnie obsługiwać wiele klientów.

Przypomnienie. Cykl życia serwletu i interfejs API

KLUCZOWE ZAGADNIENIA



- Kontener inicjalizuje serwlet przez wczytanie jego klasy, wywołanie bezargumentowego konstruktora tej klasy i wywołanie metody `init()` serwletu.
- Metoda `init()` (która może zostać nadpisana przez programistę) jest wywoływana tylko raz w całym cyklu życia serwletu; ma to miejsce zawsze przed obsługą przez serwlet jakichkolwiek żądań klientów.
- Metoda `init()` zapewnia serwletowi dostęp do obiektów `ServletConfig` i `ServletContext`, które są niezbędne do uzyskiwania informacji na temat konfiguracji serwletu i jego macierzystej aplikacji internetowej.
- Kontener kończy życie serwletu przez wywołanie jego metody `destroy()`.
- Znaczną część swojego życia serwlet poświęca na wykonywaniu metody `service()` obsługującej żądania klientów.
- Każde obsługiwane przez serwlet żądanie jest wykonywane w osobnym wątku! Może istnieć co najwyżej jeden egzemplarz każdej z klas serwletów składających się na aplikację internetową.
- Twoje serwlety niemal zawsze będą rozszerzały klasę bazową `javax.servlet.http.HttpServlet`, po której będą dziedziczyły metodę `service()` otrzymującą na wejściu obiekty klas `HttpServletRequest` i `HttpServletResponse`.
- Klasa `HttpServlet` rozszerza klasę abstrakcyjną `javax.servlet.GenericServlet`, która z kolei implementuje większość podstawowych metod serwletu.
- Klasa `GenericServlet` implementuje interfejs `Servlet`.
- Wszystkie klasy wykorzystywane w serwletach (z wyjątkiem tych związanych ze stronami JSP) należą do jednego z dwóch pakietów: `javax.servlet` lub `javax.servlet.http`.
- W swoim kodzie klasy serwletu możesz nadpisać metodę `init()` i musisz nadpisać przynajmniej jedną z metod obsługujących żądania (`doGet()`, `doPost()` itp.).



Przypomnienie. Protokół HTTP i klasa `HttpServletRequest`

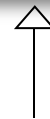
KLUCZOWE ZAGADNIENIA



- Metody `doGet()` i `doPost()` klasy `HttpServletRequest` otrzymują na wejściu (w formie argumentów) po jednym obiekcie klas `HttpServletRequest` i `HttpServletResponse`.
- Metoda `service()` określa na podstawie metody użytej w żądaniu protokołu HTTP (GET, POST itp.), czy w serwlecie należy wywołać metodę `doGet()` czy metodę `doPost()`.
- Żądania POST zawierają ciała, których nie stosuje się w żądaniach GET; żądania GET mogą jednak zawierać parametry dołączane do adresów URL (nazywane niekiedy łańcuchami zapytań).
- Żądania GET są z natury rzeczy (zgodnie ze specyfikacją protokołu HTTP) powtarzalne (idempotentne). Powinny umożliwiać wielokrotne wykonywanie tych samych działań bez jakichkolwiek negatywnych skutków dla funkcjonowania serwera. Żądania GET nie powinny zmieniać czegokolwiek po stronie serwera, co wcale nie oznacza, że nie jest możliwe napisanie złej, nieidempotentnej metody `doGet()`.
- Żądania POST są z natury rzeczy niepowtarzalne (nieidempotentne), zatem tylko od Ciebie zależy, czy projektując i kodując swoją aplikację, uwzględniłeś możliwość omyłkowego wysłania dwóch identycznych żądań przez klienta.
- Jeśli formularz HTML nie zawiera zdefiniowanego wprost atrybutu `method="POST"`, do serwletu zostanie wysłane żądanie GET, a nie POST. Jeśli Twój serwlet nie zawiera metody `doGet()`, takie żądanie spowoduje błąd.
- Parametry z obiektu reprezentującego żądanie możesz odczytywać za pomocą metody `getParameter("nazwaParametru")`. Metoda zawsze zwraca łańcuch.
- Jeśli dla jednej nazwy parametru może jednocześnie istnieć wiele wartości, powinieneś użyć metody `getParameterValues("nazwaParametru")`, która zwraca tablicę łańcuchów.
- Z obiektu reprezentującego żądanie można odczytywać także inne przydatne dane, włącznie z nagłówkami, ciasteczkami, sesją, łańcuchem zapytania oraz strumieniem wejściowym.

Interfejs `ServletRequest` (`javax.servlet.ServletRequest`)

<<interfejs>>
<i>ServletRequest</i>
<code>getAttribute(String)</code> <code>getContentTypeLength()</code> <code>getInputStream()</code> <code>getLocalPort()</code> <code>getRemotePort()</code> <code>getServerPort()</code> <code>getParameter(String)</code> <code>getParameterValues(String)</code> <code>getParameterNames()</code> <i>// I WIELE innych metod...</i>



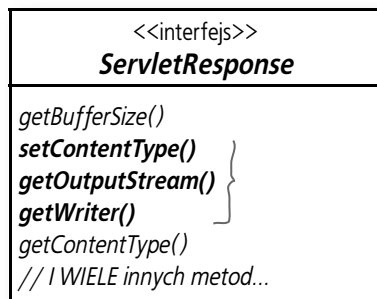
Interfejs `HttpServletRequest` (`javax.servlet.http.HttpServletRequest`)

<<interfejs>>
<i>HttpServletRequest</i>
<code>getContextPath()</code> <code>getCookies()</code> <code>getHeader(String)</code> <code>getIntHeader(String)</code> <code>getMethod()</code> <code>getQueryString()</code> <code>getSession()</code> <i>// I WIELE innych metod...</i>

Dobrze, wiemy już, do czego służy klasa `Request`... przyjrzymy się teraz klasie `Response`

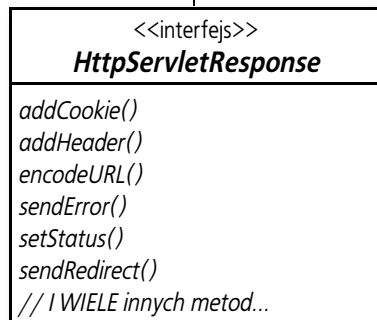
Odpowiedź jest tym, co serwlet odsyła klientowi. Odesłana odpowiedź dociera następnie do przeglądarki internetowej, która poddaje ją analizie składniowej i wizualizuje dla użytkownika. Obiektu odpowiedzi używa się zwykle do uzyskiwania strumienia wyjściowego (zazwyczaj obiektu `Writer`), w którym serwlet zapisuje odsyłany do użytkownika kod HTML (lub dowolny inny rodzaj zawartości). Poza typowymi operacjami wejścia-wyjścia na danych wyjściowych, obiekt odpowiedzi udostępnia także inne metody — niektóre z nich omówimy bardziej szczegółowo w dalszej części tego rozdziału.

Interfejs `ServletResponse` (`javax.servlet.ServletResponse`)



Są to tylko niektóre z najczęściej wykorzystywanych metod.

Interfejs `HttpServletResponse` (`javax.servlet.http.HttpServletResponse`)



Niekiedy będziesz korzystać także z tych metod.

Przez większość czasu będziesz używał obiektu klasy `Response` wyłącznie do odsyłania danych do klienta.

W pierwszej kolejności musisz wywołać dwie metody obiektu odpowiedzi: `setContentType()` oraz `getWriter()`.

Bezpośrednio potem możesz wykonywać proste operacje wejścia-wyjścia prowadzące się do zapisywania w strumieniu wyjściowym kodu HTML (lub dowolnej innej zawartości).

Obiektu odpowiedzi można jednak używać także do ustawiania innych nagłówków, wysyłania informacji o błędach oraz dodawania znaczników kontekstu (ciasteczek) klienta.

Chwileczkę... nie sądziłem, że będziemy wysyłać kod HTML z poziomu naszych serwletów, ponieważ formatowanie tego kodu w strumieniu wyjściowym jest wyjątkowo niepraktyczne.



Używanie obiektu odpowiedzi do operacji wejścia-wyjścia

To prawda, zamiast wysyłać do klienta kod HTML w konstruowanym na poziomie serwletu strumieniu wyjściowym obiektu odpowiedzi, powinniśmy raczej używać stron JSP. Formatowanie stron HTML w strumieniu wyjściowym za pomocą metody `println()` faktycznie bywa bardzo *kłopotliwe*.

Nie oznacza to jednak, że nigdy nie zetkniesz się z koniecznością pracy ze strumieniem wyjściowym na poziomie kodu serwletu.

Dlaczego?

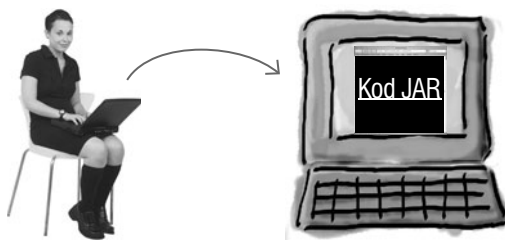
- 1) Firma prowadząca twój serwer WWW może nie obsługiwać stron JSP. Istnieje wiele starszych serwerów i kontenerów WWW, które co prawda obsługują serwlety, ale nie obsługują stron JSP, co musimy uwzględnić w projektowanych rozwiązaniach.
- 2) Opcja przewidująca użycie stron JSP może nie mieć zastosowania także z innych powodów; np. dlatego, że jesteś zmuszony do współpracy z wyjątkowo głupim kierownikiem projektu, który nie godzi się na stosowanie technologii JSP, ponieważ w 1998 roku szwagier powiedział mu, że miał w swojej pracy niedobre doświadczenia ze stronami JSP.
- 3) Kto powiedział, że kod *HTML* jest jedynym rodzajem zawartości, który możemy odsyłać w ramach odpowiedzi kierowanej do klienta? Moglibyśmy przecież równie dobrze odsyłać klientowi coś zupełnie *innego*. Coś, czego obsługa z wykorzystaniem obiektu strumienia wyjściowego jest zupełnie naturalna.

Na następnej stronie znajdziesz odpowiedni przykład...

Wyobraź sobie, że chcesz odesłać klientowi plik JAR...

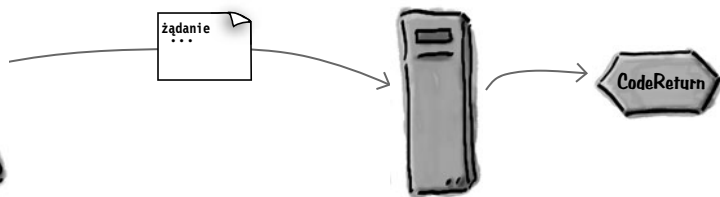
Przypuśćmy, że opracowałeś stronę pobierania plików, gdzie klient ma dostęp do kodu zawartego w plikach JAR. Zamiast odsyłać klientowi stronę HTML, obiekt odpowiedzi powinien zawierać bajty reprezentujące wybierane pliki JAR. W tej sytuacji powinieneś *odczytywać* bajty z pliku JAR, po czym *zapisywać* je w strumieniu wyjściowym obiektu odpowiedzi.

- ① Diane desperacko szuka sposobu pobrania pliku JAR z kodem dla książki, z której uczy się pisać serwlety i strony JSP. Otwiera witrynę internetową tej książki i klika łącze Kod JAR wskazujące na serwlet nazwany Code.do.



Przeglądarka wysyła do serwera żądanie HTTP z nazwą interesującego użytkownika serwletu (Code.do).

Kontener przekazuje otrzymane żądanie do przetworzenia przez serwlet CodeReturn (którego nazwa jest odwzorowywana w deskrytorze wdrożenia w nazwę Code.do).



- ② Rozpoczyna się pobieranie pliku JAR do komputera klienta. Diane jest bardzo zadowolona.



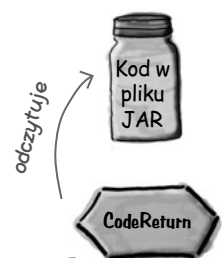
Odpowiedź HTTP zawiera teraz bajty reprezentujące plik JAR.

Serwlet CodeReturn pobiera bajty z przechowywanego na serwerze pliku JAR i uzyskuje strumień wyjściowy z otrzymanego obiektu odpowiedzi, po czym zapisuje w tym obiekcie bajty reprezentujące żądany plik JAR.



bajty z pliku JAR

odpowiedź



odczytuje

zapisuje

Kod serwletu obsługującego pobieranie pliku JAR

// w tym miejscu znajduje się garść operacji importujących

```
public class CodeReturn extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException {

        response.setContentType("application/jar");

        ServletContext ctx = getServletContext();
        InputStream is = ctx.getResourceAsStream("/bookCode.jar");

        int read = 0;
        byte[] bytes = new byte[1024];

        OutputStream os = response.getOutputStream();
        while ((read = is.read(bytes)) != -1) {
            os.write(bytes, 0, read);
        }
        os.flush();
        os.close();
    }
}
```

Chcemy, aby przeglądarka wiedziała, że otrzyma bajty pliku JAR, a nie kod HTML, zatem ustawiamy inny niż zazwyczaj typ zawartości: „application/jar”.

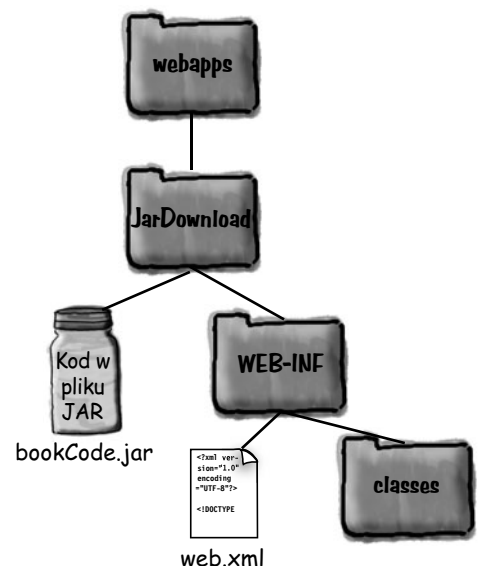
W ten sposób określamy jedynie, że chcemy uzyskać dostęp do strumienia wejściowego dla zasobu nazwanego bookCode.jar.

Oto kluczowa część tego serwletu, ale jak widać, są to tylko tradycyjne, proste operacje wejścia-wyjścia!! Nie ma w tym nic specjalnego, po prostu odczytujemy bajty pliku JAR, po czym zapisujemy je w strumieniu wyjściowym, do którego mamy dostęp za pośrednictwem obiektu odpowiedzi.

Nie ma niemądrych pytań

P: Gdzie powinien znajdować się udostępniany plik JAR (bookCode.jar)? Inaczej mówiąc, gdzie metoda `getResourceAsStream()` powinna SZUKAĆ tego pliku? Jak należy interpretować użytą ścieżkę?

U: Metoda `getResourceAsStream()` wymaga od nas użycia znaku ukośnika (/) reprezentującego katalog główny naszej aplikacji internetowej. Ponieważ nazwaliśmy naszą aplikację **JarDownload**, struktura jej katalogów powinna przypominać strukturę przedstawioną na rysunku obok. Katalog **JarDownload** znajduje się wewnątrz katalogu **webapps** (wspólnego dla wszystkich aplikacji internetowych), natomiast w samym katalogu **JarDownload** umieszczamy katalog **WEB-INF** oraz sam plik JAR. Plik *bookCode.jar* jest więc przechowywany w katalogu głównym aplikacji internetowej **JarDownload** (nie przejmuj się, wszelkie szczegóły związane ze strukturą katalogów wdrażanej aplikacji omówimy w rozdziale poświęconym wdrażaniu).

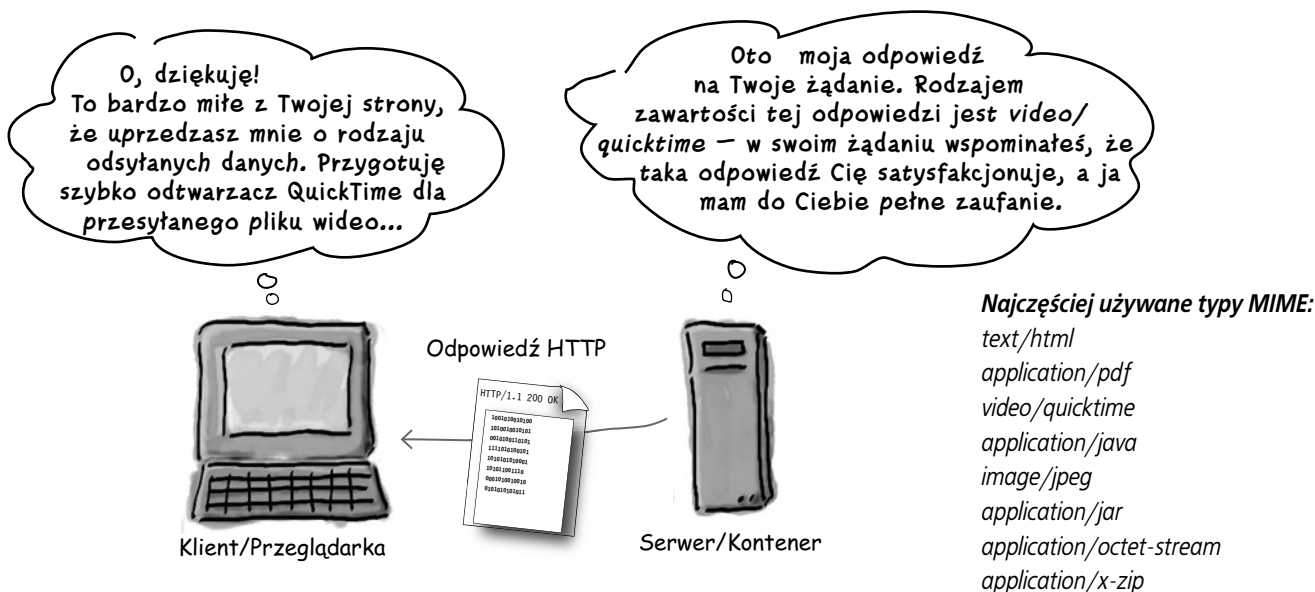


Świetnie, ale do czego służy rodzaj zawartości?

Być może zastanawiasz się, co w przedstawionym kodzie serwletu oznacza następujący wiersz:

```
response.setContentType("application/jar");
```

A przynajmniej *powinieneś się zastanawiać*. Musisz przecież przekazać przeglądarce, co do niej odsyłasz w odpowiedzi HTTP, ponieważ tylko w ten sposób możliwe będzie *zapewnienie właściwej obsługi tej odpowiedzi*: uruchomienie aplikacji „pomocniczej” (np. przeglądarki dokumentów PDF lub odtwarzacza plików wideo), wizualizacja kodu HTML, zapisanie bajtów odpowiedzi w postaci pobranego pliku etc. Skoro tak Cię to interesuje, od razu odpowiadamy: owszem, kiedy mówimy o *rodzaju zawartości*, myślimy o typie MIME. Rodzaj zawartości jest jednym z tych nagłówek protokołu HTTP, który *musi* być dołączany do każdej odpowiedzi HTTP.



Nie musisz pamiętać dostępnych rodzajów zawartości.

Powinieneś wiedzieć, co robi metoda `setContentType()` i jak należy z niej korzystać, ale poza typem MIME

- text/html, nie musisz znać na pamięć nawet najczęściej stosowanych rodzajów zawartości. Z pewnością
- powinieneś wiedzieć, jak w praktyce stosuje się metodę `setContentType()`... Przykładowo, wywołanie tej metody PO zapisaniu odpowiedzi w strumieniu wyjściowym niczego nie zmienia. To oczywiste. Ale wcale nie musi to oznaczać, że nie możesz ustawić rodzaju zawartości, zapisać jakichś danych w strumieniu wyjściowym, po czym zmienić rodzaju zawartości i zapisać w strumieniu wyjściowym innych danych. Warto się jednak zastanowić, jak tak skonstruowana odpowiedź zostanie zinterpretowana przez przeglądarkę internetową. Przeglądarka może jednocześnie obsługiwać tylko jeden rodzaj TREŚCI w otrzymanej odpowiedzi.



Relax

Upewnij się, że wszystko działa prawidłowo; dobrą praktyką (a w niektórych przypadkach absolutną koniecznością) jest wywołanie metody `setContentType()` zawsze jako pierwszej, a więc PRZED wywołaniem metody zwracającej obiekt strumienia wyjściowego (`getWriter()` lub `getOutputStream()`). W ten sposób możemy wyeliminować ryzyko konfliktów pomiędzy rodzajem zawartości a strumieniem wyjściowym.

Nie ma niemądrych pytań

P: Dlaczego musimy ustawiać rodzaj zawartości? Czy serwer nie może sam określić rozszerzenia odsyłanego pliku?

U: Większość serwerów *może*, przynajmniej dla zawartości statycznej. Przykładowo, w serwerze Apache możemy ustawić typy MIME przez odwzorowanie określonych rozszerzeń plików (.txt, .jar, etc.) na konkretne rodzaje zawartości — serwer Apache będzie wówczas automatycznie wykorzystywał te odwzorowania do ustawiania nagłówka rodzaju zawartości w odpowiedzi protokołu HTTP. Mówimy jednak o tym, co dzieje się w ramach serwletu, gdzie NIE wysyłamy pliku! Odpowiadamy wyłącznie za odesłanie odpowiedzi, a obsługujący nasz serwlet kontener nie ma pojęcia, co się w tej odpowiedzi znajduje.

P: Jak wobec tego wygląda sytuacja w ostatnim przykładzie, w którym odczytywaliśmy konkretny plik JAR? Czy kontener nie może wykryć, że odczytujemy plik JAR?

U: Nie. W naszym serwlecie jedynie odczytujemy bajty z pliku (w tym przypadku jest to akurat plik JAR), obracamy się i zapisujemy odczytane bajty w strumieniu wyjściowym. Kontener nie ma pojęcia, co robiliśmy w czasie, gdy byliśmy zajęci odczytywaniem bajtów z naszego pliku JAR. Wiedza kontenera jest bardzo ograniczona — odczytujesz pewien rodzaj danych i zapisujesz coś zupełnie innego w obiekcie odpowiedzi.

P: Gdzie mogę znaleźć nazwy najbardziej popularnych rodzajów zawartości?

U: Poszukaj ich w wyszukiwarce Google. Poważnie. Co prawda stale są dodawane nowe typy MIME, jednak ich listę można bez trudu znaleźć w internecie. Możesz także zajrzeć do ustawień swojej przeglądarki internetowej, gdzie powinna się znajdować lista typów MIME skonfigurowanych w tym programie; dobrym źródłem wiedzy na temat obsługiwanych rodzajów zawartości są również pliki konfiguracyjne Twojego serwera WWW. Znajomość tych typów nie powinna być weryfikowana na egzaminie i jest mało prawdopodobne, żebyś kiedykolwiek jej potrzebował podczas pracy nad rzeczywistymi aplikacjami internetowymi.

P: Zaczekaj... dlaczego użyliśmy serwletu do odsyłania zawartości pliku JAR, skoro mogliśmy po prostu wykorzystać serwer WWW do udostępniania i odsyłania tego pliku jako jednego z zasobów? Innymi słowy, dlaczego nie użyliśmy klikanego przez użytkownika łącza, które wskazywałoby na dany plik JAR zamiast na serwlet? Czy nie moglibyśmy tak skonfigurować naszego serwera WWW, aby odsyłał plik JAR bezpośrednio (a więc BEZ angażowania serwletu)?

U: Tak. Dobre pytanie. MOGLIBYŚMY skonfigurować serwer WWW w taki sposób, aby użytkownik klikał łącze HTML wskazujące np. na plik JAR składowany na serwerze (tak jak inne zasoby, włącznie z obrazami JPEG czy plikami tekstowymi), i aby serwer odsyłał żądany plik (zasób) w swojej odpowiedzi.

Zakładamy jednak, że w naszym serwlecie będą wykonywane dodatkowe czynności ZANIM odczytane dane zostaną odesłane w postaci strumienia wyjściowego. Możemy np. umieścić w serwlecie logikę określającą, który plik JAR należy odesłać klientowi. Możemy także odsyłać klientowi bajty tworzone „w locie”, a więc w trakcie obsługi jego żądania. Wyobraźmy sobie systemy, w którym pobieramy od użytkownika parametry wejściowe, wykorzystujemy je do dynamicznego generowania dźwięku, który następnie odsyłamy do klienta. Dźwięk przekazywany do użytkownika byłby wówczas zasobem, który w ogóle nie istniał przed otrzymaniem komunikatu. Innymi słowy, nie byłoby to zasób mający postać pliku dźwiękowego składowanego gdzieś na serwerze. Byłby to natomiast zasób stworzony specjalnie na żądanie klienta i odesłany do niego w postaci odpowiedzi HTTP.

Być może masz rację, być może nasz przykład z odsyłaniem pliku JAR przechowywanego na serwerze jest nietrafiony... jeśli jednak użyjesz wyobraźni i wzbogacisz tę aplikację o możliwie wyszukane elementy, być może okaże się, że zastosowanie serwletu było *uzasadnione*. Być może wystarczy zrobić coś tak prostego jak dodanie do serwletu kodu, który oprócz odsyłania zawartości pliku JAR zapisze w bazie danych pewne informacje o użytkowniku pobierającym ten plik. Może będziemy musieli sprawdzić, czy dany użytkownik ma dostęp do żadanego kodu JAR, w oparciu o odpowiednie informacje składowane w bazie danych.

Masz do wyboru dwa rodzaje danych wyjściowych: znaki lub bajty

Poniżej zastosowaliśmy tradycyjny, prosty kod `java.io`; jedynym wyjątkiem jest wykorzystany interfejs `ServletResponse`, który daje nam do wyboru *dwa* strumienie: `ServletOutputStream` (dla bajtów) oraz `PrintWriter` (dla danych znakowych).

► PrintWriter

Przykład

```
PrintWriter writer = response.getWriter();  
  
writer.println("jakiś tekst i kod HTML");
```

Zastosowania:

Zapisywanie danych tekstowych w strumieniu znakowym. Chociaż nadal *możemy* zapisywać tego typu dane w strumieniu `OutputStream`, w takich sytuacjach warto jednak korzystać ze strumienia `PrintWriter` stworzonego specjalnie z myślą o danych znakowych.

► OutputStream

Przykład

```
ServletOutputStream out = response.getOutputStream();  
  
out.write(obiektKlasyByteArray);
```

Zastosowania:

Zapisywanie *wszelkich innych danych!*

Do Twojej wiadomości: Strumień `PrintWriter` w rzeczywistości jest „opakowaniem” strumienia `ServletOutputStream`. Innymi słowy, klasa `PrintWriter` wykorzystuje referencję do klasy `ServletOutputStream` i właśnie do niej deleguje swoje wywołania. Istnieje oczywiście tylko **JEDEN** strumień wyjściowy docierający do klienta, ale klasa `PrintWriter` „dekoruje” ten strumień przez dodanie metod wyższego poziomu przyjaznych znakom.



Oglądaj to!

MUSISZ zapamiętać te metody

Z wymienionymi obok metodami *musisz* się dobrze zapoznać przed egzaminem. Ich zapamiętanie nie jest trudne. Zwróć uwagę, że do zapisywania danych w strumieniu `ServletOutputStream` używasz metody `write()`, ale do zapisywania danych tekstowych w strumieniu `PrintWriter` stosuje się... metodę `println()`. Mogłbyś oczywiście przyjąć naturalne założenie, że zapis w strumieniu `Writer` wymaga metody `write`, ale niestety tak nie jest. Jeśli stosowałeś już pakiet `java.io`, zapamiętanie tych metod nie powinno stanowić większego problemu. Jeśli jednak z tego typu operacjami nie miałeś wcześniej do czynienia, po prostu zapamiętaj:

`println()` dla strumienia **`PrintWriter`**
`write()` dla strumienia **`ServletOutputStream`**

Musisz także zapamiętać, że w nazwach metod zwracających zarówno obiekt strumienia znakowego, jak i obiekt strumienia bajtowego nie występuje pierwsze słowo zwracanego typu:

`ServletOutputStream` `out = response.getOutputStream();`

`PrintWriter` `writer = response.getWriter();`

Musisz także potrafić prawidłowo rozpoznawać **NIEPOPRAWNE** nazwy metod:

`getPrintWriter()`
`getResponseStream()`
`getStream()`
`getOutputStream()`

} Te metody **NIE** istnieją!

Możesz ustawiać nagłówki odpowiedzi, możesz dodawać nagłówki odpowiedzi

Być może zastanawiasz się, jaka jest różnica pomiędzy ustawianiem a dodawaniem nagłówków odpowiedzi. Przemyśl to i spróbuj wykonać poniższe ćwiczenie.

Dopasuj wywołanie metody do jej zachowania	
<code>response.setHeader("foo", "bar");</code>	Narysuj strzałkę łączącą wywołanie metody klasy <code>HttpServletResponse</code> z odpowiednim zachowaniem metody. Najbardziej oczywistego wyboru dokonaliśmy już za Ciebie.
<code>response.addHeader("foo", "bar");</code>	<i>Dodaje do odpowiedzi nowy nagłówek wraz z wartością lub dodaje nową wartość do istniejącego nagłówka.</i>
<code>response.setIntHeader("foo", 42);</code>	<i>Przydatna metoda, która zastępuje wartość istniejącego nagłówka nową wartością całkowitoliczbową lub dodaje do odpowiedzi nowy nagłówek wraz z całkowitoliczbową wartością.</i>
	<i>Jeśli nagłówek z taką nazwą już istnieje w obiekcie odpowiedzi, jego oryginalna wartość jest zastępowana nazwą przekazaną za pośrednictwem drugiego argumentu tej metody. W przeciwnym przypadku metoda dodaje do obiektu odpowiedzi nowy nagłówek wraz z wartością.</i>

Kiedy wszystkie te metody wymienimy w jednym miejscu, ich znaczenie wydaje się oczywiste.

Jednak podczas egzaminu powinieneś znać te metody na pamięć — kiedy w środku nocy z następnego wtorku na środę ktoś Cię zapyta: „Która metoda obiektu odpowiedzi umożliwia dodawanie wartości do istniejących nagłówków?”, powinieneś bez zastanowienia odpowiedzieć: „To oczywiście metoda `addHeader()`, która pobiera dwa łańcuchy reprezentujące nazwę i wartość nagłówka”. Właśnie tak powinieneś być przygotowany do egzaminu.

Zarówno metoda `setHeader()`, jak i metoda `addHeader()` dodaje do obiektu odpowiedzi nagłówek i wartość, jeśli ten obiekt jeszcze nie zawiera danego nagłówka (pierwszego argumentu metody). Różnica pomiędzy tymi metodami ujawnia się dopiero w sytuacji, gdy obiekt odpowiedzi *zawiera* wskazany nagłówek. W takim przypadku:

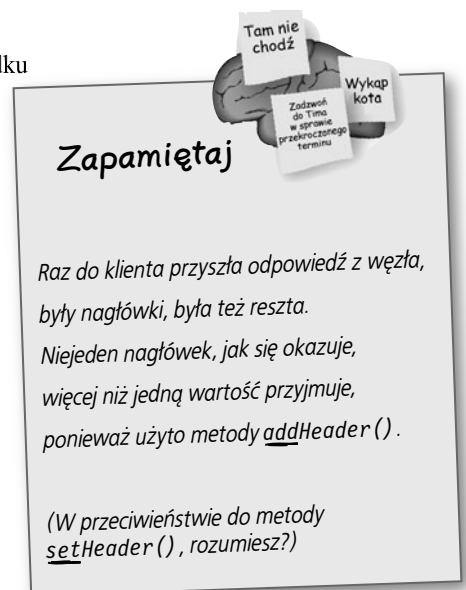
metoda `setHeader()` nadpisuje istniejącą wartość,

metoda `addHeader()` dodaje nową wartość.

Kiedy wywołujemy metodę `setContentType("text/html")`, w rzeczywistości ustawiamy odpowiedni nagłówek, a więc wywołanie to nie różni się od wywołania innej metody:

```
setHeader("content-type", "text/html");
```

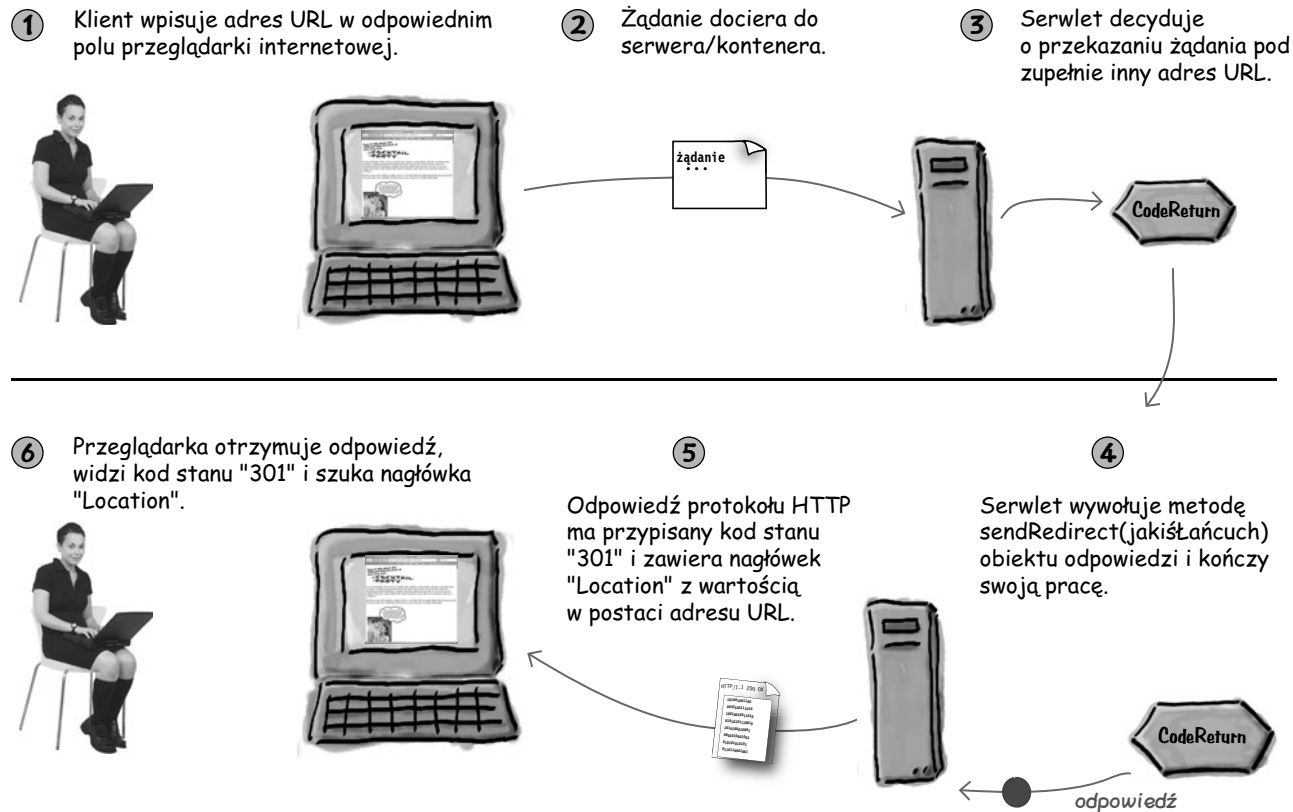
Jaka jest więc różnica pomiędzy tymi wywołaniami? Żadna, pod warunkiem, że prawidłowo wpiszesz nazwę nagłówka `"content-type"`. Metoda `setHeader()` nie zasygnalizuje błędu, jeśli popełnisz błąd w pisowni nazwy któregoś z nagłówków — po prostu założysz, że dodajesz nowy rodzaj nagłówka. Błąd wyjdzie na jaw nieco później, kiedy okaże się, że nie ustawiłeś prawidłowo rodzaju zawartości odpowiedzi!



Niekiedy po prostu nie chcemy samodzielnie operować na obiektach reprezentujących odpowiedzi...

Możesz wybrać rozwiązanie polegające na obsłudze odpowiedzi na żądanie przez inny moduł programowy. Możesz albo *przekierować* żądanie pod zupełnie inny adres URL, albo *przydzielić* żądanie do innego komponentu swojej aplikacji internetowej (zwykle jest to strona JSP).

Przekierowanie



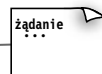
7

Przeglądarka generuje nowe żądanie w oparciu o adres URL otrzymany w nagłówku "Location" dołączonym do odpowiedzi na poprzednie żądanie. Użytkownik może zauważyć, że zmienił się adres URL wyświetlany na pasku adresu przeglądarki internetowej...



8

W nowym żądaniu nie ma nic szczególnego (fakt, iż zostało wygenerowane w odpowiedzi na przekierowanie, nie ma wpływu na jego kształt).



9

Serwer odnajduje zasób wskazywany przez żądany adres URL. Nie ma w tym działaniu niczego szczególnego.



11

Przeglądarka wizualizuje nową stronę. Użytkownik jest zaskoczony.



10

Odpowiedź HTTP nie różni się specjalnie od innych odpowiedzi tego protokołu... w tym przypadku pochodzi z innego serwera niż ten, który został początkowo wskazany przez użytkownika wpisującego adres URL.



Jak to możliwe, że otrzymałam tę stronę?



jesteś tutaj ▶ 163

Przekierowanie z poziomu serwletu zmusza przeglądarkę do dodatkowych działań

Mechanizm przekierowywania żądań umożliwia naszemu serwletowi całkowite pozbywanie się odpowiedzialności na rzecz innych składników tej samej aplikacji lub innych aplikacji. Kiedy nasz serwlet zdecyduje, że danego żądania nie może obsłużyć, może po prostu wywołać metodę `sendRedirect()`:

```
if (toZadanieDlaMnie) {  
    // obsługa żądania  
} else {  
    response.sendRedirect("http://www.helion.pl");  
}
```

Adres URL, który ma być użyty dla danego żądania przez przeglądarkę. Witryna reprezentowana przez ten adres zostanie wyświetlona w oknie przeglądarki użytkownika.

Stosowanie względnych adresów URL w wywołaniach metody `sendRedirect()`

Okazuje się, że w roli argumentu metody `sendRedirect()` możemy używać względnych adresów URL, zamiast określać całe wyrażenia "http://www...". Każdy taki adres URL może mieć jedną z dwóch postaci: rozpoczynającą się od znaku ukośnika lub pozbawioną tego znaku.

Wyobraź sobie, że klient wpisał w polu adresu przeglądarki internetowej następujący adres:

`http://www.wickedlysmart.com/myApp/cool/bar.do`

Kiedy odpowiednie żądanie dotrze do serwletu nazywanego `bar.do`, serwlet ten wywoła metodę `sendRedirect()` ze względным adresem URL, który nie będzie się rozpoczynał od znaku ukośnika:

```
sendRedirect("foo/stuff.html");
```

Kontener buduje pełny adres URL (adres w tej formie jest niezbędny, aby można go było umieścić w nagłówku "Location" odpowiedzi HTTP) na bazie adresu URL z oryginalnego żądania:

`http://www.wickedlysmart.com/myApp/cool/foo/stuff.html`

Gdyby jednak argument metody `sendRedirect()` **ROZPOCZYNAŁ** się od znaku ukośnika:

```
sendRedirect("/foo/stuff.html");
```

Kontener zbudowałby kompletny adres URL względem samego kontenera WWW, nie względem adresu URL użytego w oryginalnym żądaniu. Oznacza to, że nowy adres URL miałby postać:

`http://www.wickedlysmart.com/foo/stuff.html`

„foo” jest aplikacją internetową niezwiązaną z aplikacją „myApp”.

Kontener zna adres URL oryginalnego żądania rozpoczynający się od ścieżki `myApp/cool`, zatem jeśli nie użyjemy znaku ukośnika, ta część ścieżki zostanie dodana na początek adresu `foo/stuff.html`.

Znak ukośnika na początku adresu przekierowania oznacza, że podano ścieżkę względem katalogu głównego tego kontenera.



Oglądaj to!

Nie możesz wywoływać metody `sendRedirect()` już po zapisaniu danych w obiekcie odpowiedzi!

Prawdopodobnie powyższa zasada jest zupełnie oczywista, ale warto ją powtarzać choćby ze względu na jej niemałe ZNACZENIE.

Jeśli przyjrzyj się metodzie `sendRedirect()` w interfejsie API, z pewnością zauważysz, że w razie próby jej wywołania już po zatwierdzeniu odpowiedzi serwletu jest generowany wyjątek `IllegalStateException`.

Przez zatwierdzenie odpowiedzi rozumiemy jej **odesłanie** do klienta. Zatwierdzenie odpowiedzi oznacza po prostu zapisanie danych wyjściowych w odpowiednim strumieniu. Z praktycznego punktu widzenia prezentowana reguła określa, że **nie możesz zapisywać danych w obiekcie odpowiedzi przed wywołaniem metody `sendRedirect()`**!

Z drugiej strony, zapewne przyjdzie Ci dyskutować z przemądrzałym naukowcem, który będzie twierdził, że z technicznego punktu widzenia możesz zapisywać dane w strumieniu bez ich zatwierdzania — wówczas wywołanie metody `sendRedirect()` nie powinno doprowadzić do wyjątku. Takie działanie byłoby jednak na tyle bezsensowne, że nie będziemy tracić czasu na jego omawianie (oczywiście nie uwzględniając tego, który już poświęciliśmy temu zagadnieniu...).

W naszym serwlecie musimy wreszcie podjąć jakąś sensowną decyzję! Powinniśmy albo obsłużyć otrzymane żądanie, albo wywołać metodę `sendRedirect()`, która przekaże to żądanie komuś INNEMU.

(Nawiasem mówiąc, koncepcja ograniczonego wyboru po zaakceptowaniu danych w strumieniu wyjściowym ma zastosowanie także w przypadku ustawianych nagłówków, znaczników kontekstu klienta, kodów stanu, rodzaju zawartości itp.)



Oglądaj to!

Metoda `sendRedirect()` otrzymuje zwykły łańcuch, NIE obiekt reprezentujący adres URL!

Cóż, tak naprawdę metoda `sendRedirect()` otrzymuje łańcuch, który JEST jednocześnie adresem URL. Zasadnicze znaczenie ma jednak fakt, iż metoda `sendRedirect()` NIE otrzymuje obiektu typu URL. Przekazujemy za pośrednictwem parametru łańcuch, który jest albo kompletnym adresem URL, albo adresem względnym. Jeśli kontener nie będzie mógł zbudować pełnego adresu na bazie otrzymanego adresu względnego, zostanie wygenerowany wyjątek `IllegalStateException`. Najważniejsze, żebyś zapamiętał, że TAKIE wywołanie jest błędne: `sendRedirect(new URL("http://www.helion.pl"))`;

Nie! Wygląda dobrze, ale NA PEWNO nie zadziała. Metoda `sendRedirect()` otrzymuje łańcuch. I kropka.

Przydział żądań ma miejsce wyłącznie po stronie serwera

Na tym właśnie polega zasadnicza różnica pomiędzy przekierowywaniem żądań a ich przydzielaniem — *przekierowanie* zmusza do dodatkowych działań klienta, natomiast *przydział żądania* wymaga wykonania pewnych kroków po stronie serwera. Zapamiętaj dwie proste reguły: przekierowanie = *klient*, przydział żądania = *serwer*. Mechanizmowi przekazywania żądań poświęcimy co prawda znacznie więcej uwagi w jednym z rozdziałów w dalszej części tej książki, jednak ta i kolejna strona powinna na tym etapie w zupełności wystarczyć.

Przydział żądania

- ❶ Użytkownik wpisuje w polu przeglądarki adres URL serwletu.



- ❷ Żądanie dociera do serwera/kontenera.



- ❸ Serwlet decyduje, czy dane żądanie powinno trafić do innej części tej samej aplikacji internetowej (w tym przypadku do odpowiedniej strony JSP).



- ❺ Przeglądarka otrzymuje zwykłą odpowiedź, którą niezwłocznie wizualizuje dla użytkownika. Ponieważ zawartość paska adresu w oknie przeglądarki się nie zmieniła, użytkownik nie wie nawet, że odpowiedź została wygenerowana przez stronę JSP.



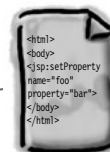
- ❹ Serwlet wywołuje metody:

```
RequestDispatcher view = request.getRequestDispatcher("wynik.jsp");  
view.forward(request, response);
```

zatem obiekt odpowiedzi przejmując teraz stronę JSP.



odpowiedź



Przekierowanie kontra przydział żądania

Nie mam na to czasu!
Czemu nie zadzwoniłeś
do Barney'a. Może on ma
czas na głupoty!

Przekierowanie



Serwlet wykonujący **przekierowanie** działa tak, jakby prosił klienta o wywołanie zamiast niego kogoś innego. W tym przypadku klientem jest przeglądarka, a nie jej użytkownik. Przeglądarka wykonuje nowe wywołanie w imieniu użytkownika bezpośrednio po tym, jak serwlet będący adresatem oryginalnego żądania stwierdził: „Przykro mi, wywołaj lepiej tego gościa...”.

Użytkownik widzi w przeglądarce nowy adres URL.

Chciałbym, żebyś mi pomogła z jednym klientem. Przekażę Ci szczegóły na temat kontaktu z nim, ale musisz się tym zająć natychmiast.

Tak, WIEM, że masz swoje zajęcia... tak, WIEM, jak ważny jest Widok we wzorcu Modelu-Widoku-Kontrolera... nie, nie mogę znaleźć do tego zadania innej strony JSP... co? odmawiasz... przepraszam – nie słyszę... tracę pakiety...

Przydział żądania

Serwlet wykonujący **przydział żądania** działa tak, jakby prosił współpracownika o przejęcie obsługi danego klienta. Współpracownik serwletu odpowiada klientowi na jego żądanie, ale klient nie wie, że nie komunikuje się ze współpracownikiem wywołanym w pierwszym żądaniu.

Użytkownik nigdy nie wie, że ktoś inny przejął jego obsługę, ponieważ adres URL wyświetlany na pasku przeglądarki internetowej pozostaje niezmienny.



Przypomnienie. `HttpServletResponse`

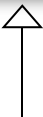
KLUCZOWE ZAGADNIENIA



- Obiektu odpowiedzi używamy do odsyłania danych do klienta.
- Do metod interfejsu `HttpServletResponse`, które będziesz wywoływał najczęściej, należą `setContentType()` i `getWriter()`.
- Zachowaj ostrożność — wielu programistów sądzi, że metoda zwracająca obiekt klasy `PrintWriter` nazywa się `getPrintWriter()`, choć jej prawdziwa nazwa to `getWriter()`.
- Metoda `getWriter()` umożliwia wykonywanie znakowych operacji wejścia-wyjścia na strumieniu (operacje te najczęściej dotyczą kodu HTML, ale mogą dotyczyć dowolnych innych danych tekstowych).
- Obiektu odpowiedzi możesz używać także do ustawiania nagłówków, wysyłania informacji o błędach oraz dodawania ciasteczek.
- W rzeczywistych zastosowaniach najprawdopodobniej będziesz używał stron JSP do przesyłania większości odpowiedzi mających postać kodu HTML, ale do odsyłania klientowi danych binarnych (np. plików JAR) nadal możesz korzystać ze strumienia danych wyjściowych.
- Do uzyskiwania obiektu reprezentującego binarny strumień wyjściowy służy metoda `getOutputStream()` obiektu odpowiedzi.
- Metoda `setContentType()` określa, w jaki sposób przeglądarka powinna obsłużyć dane otrzymywane w ramach odpowiedzi HTTP. Typowymi rodzajami zawartości tego typu odpowiedzi są: "text/html", "application/pdf" oraz "image/jpeg".
- Nie musisz uczyć się na pamięć dostępnych rodzajów zawartości (znanych także jako typy MIME).
- Nagłówki odpowiedzi możesz ustawiać za pomocą metod `addHeader()` i (lub) `setHeader()`. Różnica pomiędzy tymi metodami zależy od tego, czy dany nagłówek jest już częścią konstruowanej odpowiedzi. Jeśli tak, metoda `setHeader()` zastąpi wartość tego nagłówka, natomiast metoda `addHeader()` doda do istniejącej odpowiedzi nową wartość. Jeśli jednak danego nagłówka nie zdefiniowano jeszcze w obiekcie odpowiedzi, działanie metod `addHeader()` i `setHeader()` będzie identyczne.
- Jeśli nie chcesz odpowiadać na otrzymane żądanie, możesz je przekierować na inny adres URL. Odpowiedzialność za wysłanie nowego żądania na wskazany przez Ciebie adres spadnie wówczas na przeglądarkę internetową.
- Aby przekierować żądanie, wywołaj metodę `sendRedirect()` (łańcuch URL) obiektu odpowiedzi.
- Metody `sendRedirect()` (łańcuch URL) nie można wywoływać po zatwierdzeniu odpowiedzi! Innymi słowy, jeśli zapisałś już coś w strumieniu wyjściowym, na przekierowanie żądania klienta jest już za późno.
- Przekierowanie żądania nie jest równoznaczne z jego przydzieleniem. Przydział żądania (szczegółowo omówiony w innym rozdziale tej książki) jest realizowany w ramach serwera, natomiast przekierowanie wymaga udziału klienta. Przydział żądania w praktyce sprowadza się do jego przekazania do innego komponentu tego samego serwera, zwykle do komponentu należącego do tej samej aplikacji internetowej. Przekierowanie żądania polega natomiast na przekazaniu przeglądarce zalecenia wysłania żądania na inny adres URL.

Interfejs `ServletRequest` (`javax.servlet.ServletRequest`)

<<interfejs>> <i>ServletRequest</i>
<code>getBufferSize()</code> <code>setContentType()</code> <code>getOutputStream()</code> <code>getWriter()</code> <code>getContentType()</code> <i>// I WIELE innych metod...</i>



Interfejs `HttpServletResponse` (`javax.servlet.http.HttpServletResponse`)

<<interfejs>> <i>HttpServletResponse</i>
<code>addCookie()</code> <code>addHeader()</code> <code>encodeURL()</code> <code>sendError()</code> <code>setStatus()</code> <code>sendRedirect()</code> <i>// I WIELE innych metod...</i>



**BAR
KAWOWY**

Examin próbny

1 W jaki sposób kod serwletu zawarty w metodzie obsługującej żądania (np. `doPost()`) może uzyskać wartość nagłówka "User-Agent" z obiektu żądania? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. `String userAgent = request.getParameter("User-Agent");`
- ☐ B. `String userAgent = request.getHeader("User-Agent");`
- ☐ C. `String userAgent = request.getRequestHeader("Mozilla");`
- ☐ D. `String userAgent = getServletContext().getInitParameter("User-Agent");`

2 Które metody protokołu HTTP są wykorzystywane do sygnalizowania klientowi tego, co dociera do serwera WWW? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. `GET`
- ☐ B. `PUT`
- ☐ C. `TRACE`
- ☐ D. `RETURN`
- ☐ E. `OPTIONS`

3 Która metoda interfejsu `HttpServletResponse` służy do przekierowywania żądań protokołu HTTP pod inne adresy URL?

- ☐ A. `sendURL()`
- ☐ B. `redirectURL()`
- ☐ C. `redirectHttp()`
- ☐ D. `sendRedirect()`
- ☐ E. `getRequestDispatcher()`

-
- 4** Które metody protokołu HTTP NIE są uważane za idempotentne (powtarzalne)? (Zaznacz wszystkie prawidłowe odpowiedzi).
- ☐ A. **GET**
 - ☐ B. **POST**
 - ☐ C. **HEAD**
 - ☐ D. **PUT**
-
- 5** Wiedząc, że **req** jest obiektem klasy **HttpServletRequest**, wskaż wywołania metod, które zwrócą binarny strumień wyjściowy. (Zaznacz wszystkie prawidłowe odpowiedzi).
- ☐ A. **BinaryInputStream s = req.getInputStream();**
 - ☐ B. **ServletInputStream s = req.getInputStream();**
 - ☐ C. **BinaryInputStream s = req.getBinaryStream();**
 - ☐ D. **ServletInputStream s = req.getBinaryStream();**
-
- 6** Jak należy ustawiać nagłówek nazwany "CONTENT-LENGTH" w obiekcie **HttpServletResponse**? (Zaznacz wszystkie prawidłowe odpowiedzi).
- ☐ A. **response.setHeader(CONTENT-LENGTH, "1024");**
 - ☐ B. **response.setHeader("CONTENT-LENGTH", "1024");**
 - ☐ C. **response.setStatus(1024);**
 - ☐ D. **response.setHeader("CONTENT-LENGTH", 1024);**
-
- 7** Wskaż fragment kodu serwletu, który zwraca strumień binarny umożliwiający zapisywanie w obiekcie **HttpServletResponse** obrazów lub danych binarnych innego rodzaju.
- ☐ A. **java.io.PrintWriter out = response.getWriter();**
 - ☐ B. **ServletOutputStream out = response.getOutputStream();**
 - ☐ C. **java.io.PrintWriter out = new PrintWriter(response.getWriter());**
 - ☐ D. **ServletOutputStream out = response.getBinaryStream();**

8 Które metody są wykorzystywane w serwlecie do obsługi nadesłanych przez klienta danych z formularza? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. `HttpServletRequest.doHead()`
- ☐ B. `HttpServletRequest.doPost()`
- ☐ C. `HttpServletRequest.doForm()`
- ☐ D. `ServletRequest.doGet()`
- ☐ E. `ServletRequest.doPost()`
- ☐ F. `ServletRequest.doForm()`

9 Które z wymienionych poniżej metod zadeklarowano w interfejsie `HttpServletRequest`, nie w interfejsie `ServletRequest`? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. `getMethod()`
- ☐ B. `getHeader()`
- ☐ C. `getCookies()`
- ☐ D. `getInputStream()`
- ☐ E. `getParameterNames()`

10 Jak programiści serwletów powinni obsługiwać metodę `service()` klasy `HttpServlet` podczas jej rozszerzania we własnych klasach serwletów? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. W większości przypadków powinni nadpisywać metodę `service()`.
- ☐ B. Powinni wywoływać metodę `service()` z poziomu metody `doGet()` lub `doPost()`.
- ☐ C. Powinni wywoływać metodę `service()` z poziomu metody `init()`.
- ☐ D. Powinni nadpisywać przynajmniej jedną z metod `doXXX()` (np. metodę `doPost()`).



BAR KAWOWY

Egzamin próbny — odpowiedzi

-
- 1** W jaki sposób kod serwletu zawarty w metodzie obsługującej żądania (np. **doPost()**) może uzyskać wartość nagłówka „User-Agent” z obiektu żądania? (Zaznacz wszystkie prawidłowe odpowiedzi). (API)
- ☐ A. `String userAgent = request.getParameter("User-Agent");`
 - ☒ B. `String userAgent = request.getHeader("User-Agent");`
 - ☐ C. `String userAgent = request.getRequestHeader("Mozilla");`
 - ☐ D. `String userAgent = getServletContext().getInitParameter("User-Agent");`
- Odpowiedź B zawiera prawidłowe wywołanie metody z przekazaniem nazwy nagłówka w postaci parametru typu `String`.
-
- 2** Które metody protokołu HTTP są wykorzystywane do sygnalizowania klientowi tego, co dociera do serwera WWW? (Zaznacz wszystkie prawidłowe odpowiedzi). (Rozdział 4., metody HTTP)
- ☐ A. **GET**
 - ☐ B. **PUT**
 - ☒ C. **TRACE**
 - ☐ D. **RETURN**
 - ☐ E. **OPTIONS**
- Ta metoda jest zwykle wykorzystywana do rozwiązywania problemów, a nie do zwykłego przetwarzania żądań.
-
- 3** Która metoda interfejsu **HttpServletResponse** służy do przekierowywania żądań protokołu HTTP pod inne adresy URL? (API)
- ☐ A. `sendURL()`
 - ☐ B. `redirectURL()`
 - ☐ C. `redirectHttp()`
 - ☒ D. `sendRedirect()`
 - ☐ E. `getRequestDispatcher()`
- Odpowiedź D jest w tym przypadku prawidłowa, ponieważ metoda `sendRedirect()` jako jedyna z wymienionych należy do interfejsu `HttpServletResponse`.

- 4 Które metody protokołu HTTP NIE są uważane za idempotentne (powtarzalne)? (Zaznacz wszystkie prawidłowe odpowiedzi). (Rozdział 4., żądania idempotentne)

- ☐ A. GET
- ☒ B. POST — Zgodnie ze specyfikacją protokołu HTTP, metoda POST ma w założeniu służyć przesyłaniu żądań aktualizujących stan serwera. Ogólnie, ta sama operacja aktualizująca nie powinna być stosowana więcej niż raz.
- ☐ C. HEAD
- ☐ D. PUT

- 5 Wiedząc, że **req** jest obiektem klasy **HttpServletRequest**, wskaż wywołania metod, które zwrócą binarny strumień wyjściowy. (Zaznacz wszystkie prawidłowe odpowiedzi). (API)

- ☐ A. `BinaryInputStream s = req.getInputStream();`
- ☒ B. `ServletInputStream s = req.getInputStream();` — Odpowiedź B zawiera zarówno wywołanie właściwej metody, jak i prawidłowy typ zwracanego obiektu.
- ☐ C. `BinaryInputStream s = req.getBinaryStream();`
- ☐ D. `ServletInputStream s = req.getBinaryStream();`

- 6 Jak należy ustawiać nagłówek nazwany „CONTENT-LENGTH” w obiekcie **HttpServletResponse**? (Zaznacz wszystkie prawidłowe odpowiedzi). (API)

- ☐ A. `response.setHeader(CONTENT-LENGTH, "1024");`
- ☒ B. `response.setHeader("CONTENT-LENGTH", "1024");` — Odpowiedź B demonstruje właściwy sposób ustawiania nagłówka protokołu HTTP z dwoma parametrami łańcuchowymi: jednym reprezentującym nazwę nagłówka i drugim reprezentującym wartość.
- ☐ C. `response.setStatus(1024);`
- ☐ D. `response.setHeader("CONTENT-LENGTH", 1024);`

- 7 Wskaż fragment kodu serwletu, który zwraca strumień binarny umożliwiający zapisywanie w obiekcie **HttpServletResponse** obrazów lub danych binarnych innego rodzaju. (API)

- ☐ A. `java.io.PrintWriter out = response.getWriter();`
- ☒ B. `ServletOutputStream out = response.getOutputStream();` — Odpowiedź A nie jest prawidłową odpowiedzią, ponieważ wykorzystuje obiekt klasy `PrintWriter` stworzony z myślą o danych tekstowych.
- ☐ C. `java.io.PrintWriter out = new PrintWriter(response.getWriter());`
- ☐ D. `ServletOutputStream out = response.getBinaryStream();`

8 Które metody są wykorzystywane w serwlecie do obsługi nadesłanych przez klienta danych z formularza? (Zaznacz wszystkie prawidłowe odpowiedzi). (API)

- ☐ A. `HttpServletRequest.doHead()`
 - ☒ B. `HttpServletRequest.doPost()`
 - ☐ C. `HttpServletRequest.doForm()`
 - ☐ D. `ServletRequest.doGet()`
 - ☐ E. `ServletRequest.doPost()`
 - ☐ F. `ServletRequest.doForm()`
- Odpowiedzi C – F są błędne, ponieważ wymienione w nich metody w ogóle nie istnieją.

9 Które z wymienionych poniżej metod zadeklarowano w interfejsie `HttpServletRequest`, nie w interfejsie `ServletRequest`? (Zaznacz wszystkie prawidłowe odpowiedzi). (API)

- ☒ A. `getMethod()`
 - ☒ B. `getHeader()`
 - ☒ C. `getCookies()`
 - ☐ D. `getInputStream()`
 - ☐ E. `getParameterNames()`
- Odpowiedzi A, B i C mają związek ze składnikami żądań protokołu HTTP.

10 Jak programiści serwletów powinni obsługiwać metodę `service()` klasy `HttpServlet` podczas jej rozszerzania we własnych klasach serwletów? (Zaznacz wszystkie prawidłowe odpowiedzi). (API)

- ☐ A. W większości przypadków powinni nadpisywać metodę `service()`.
- ☐ B. Powinni wywoływać metodę `service()` z poziomu metody `doGet()` lub `doPost()`.
- ☐ C. Powinni wywoływać metodę `service()` z poziomu metody `init()`.
- ☒ D. Powinni nadpisywać przynajmniej jedną z metod `doXXX()` (np. metodę `doPost()`).

— Odpowiedź D jest w tym przypadku prawidłowa, ponieważ programiści zazwyczaj skupiają się na metodach `doGet()` i `doPost()`.

5. Atrybuty i obiekty nasłuchujące

Być aplikacją internetową



Żaden serwet nie działa samodzielnie. We współczesnych aplikacjach internetowych osiągnięcie zamierzonego celu jest możliwe dzięki współpracy wielu komponentów. Aplikacje składają się nie tylko z modeli, kontrolerów i widoków, ale także klas pomocniczych. Warto się jednak zastanowić nad sposobem łączenia tych wszystkich składników w jedną całość. Jak zmusić komponenty do *dzielenia się* informacjami? Jak *ukrywać* pewne informacje? Jak *zapewnić bezpieczeństwo informacji podczas wykonywania wątków*? Od odpowiedzi na te pytania może zależeć Twoje życie, więc zanim przystąpisz do czytania tego rozdziału, dobrze sprawdź, czy masz przygotowaną odpowiednią ilość herbaty. *I żeby mi to nie była ta beznadziejna herbatka ziołowa.*



Model kontenera WWW

- 3.1.** Dla parametrów inicjalizujących serwet i obiekt ServletContext napisz kod serwletu uzyskujący dostęp do tych parametrów i przygotuj odpowiednie elementy deskryptora wdrożenia, które będą deklarowały te parametry inicjalizujące.
- 3.2.** Dla podstawowych zasięgów atrybutów serwletu (żądania, sesji i kontekstu) napisz kod serwletu, który będzie te atrybuty dodawał, odczytywał i usuwał; mając dany scenariusz użycia, zidentyfikuj zarówno właściwy zasięg dla wskazanego atrybutu, jak i ewentualne problemy związane z przetwarzaniem wielowątkowym w poszczególnych zasięgach.
- 3.3.** *Opisz następujące elementy modelu przetwarzania żądań w kontenerze WWW: filtry, łańcuchy filtrów, opakowania żądań i odpowiedzi oraz zasoby WWW (serwetów i stron JSP).*
- 3.4.** Opisz model zdarzeń składających się na cykl życia kontenera WWW ze szczególnym uwzględnieniem roli żądań, sesji i aplikacji internetowych; opracuj i skonfiguruj klasy nasłuchujące dla cyklu życia w każdym zasięgu; przygotuj i skonfiguruj klasy nasłuchujące dla atrybutu zasięgu; dla podanego scenariusza zidentyfikuj właściwą klasę nasłuchującą atrybutów.
- 3.5.** Opisz mechanizm RequestDispatcher; napisz kod serwletu, który będzie przekazywał żądanie; napisz kod serwletu, który będzie przekazywał lub dołączał docelowy zasób, oraz zidentyfikuj dodatkowe atrybuty zasięgu żądania, które kontener udostępnia dla zasobów docelowych .

Wdrażanie aplikacji internetowych

Wszystkie wymienione obok cele egzaminu zostaną dokładnie omówione jeszcze w tym rozdziale (wyjątkiem jest cel 3.3, który omówimy w rozdziale poświęconym filtrom).

O większości zagadnień omówionych w tym rozdziale będziemy wspominać także w innych rozdziałach tej książki, jeśli jednak przygotowujesz się do egzaminu, właśnie TEN rozdział powinieneś dokładnie przestudiować pod kątem wymienionych tematów egzaminacyjnych.

Omówione w rozdziale poświęconym filtrom.



Chcę, żeby mój adres poczty elektronicznej był widoczny na generowanej przez mój serwet stronie internetowej na temat piwa... podejrzewam jednak, że mój adres będzie się zmieniał, a z pewnością nie będę miał w przyszłości możliwości wielokrotnego kompilowania kodu mojego serwletu...



Kim chce skonfigurować swój adres poczty elektronicznej w deskrytorze wdrożenia, zamiast kodować ten adres na stałe w klasie serwletu

Oto, czego Kim *nie* powinien robić w kodzie swojego serwletu:

```
PrintWriter out = response.getWriter();
out.println("blooper@wickedlysmart.com");
```

Kodowanie adresu na stałe jest w tym przypadku **NIEPOŻĄDANE!**

Co będzie, kiedy jego adres poczty elektronicznej się zmieni? Będzie musiał ponownie skompilować serwlet...

Znacznie lepszym rozwiązaniem będzie umieszczenie adresu poczty elektronicznej Kima w *deskrytorze wdrożenia* (pliku *web.xml*), ponieważ serwlet będzie wówczas mógł „odczytywać” ten adres z deskryptora już po wdrożeniu aplikacji internetowej w docelowym środowisku. W ten sposób Kim uniknie konieczności trwałego kodowania swojego adresu w klasie serwletu, a ewentualne zmiany będą wymagały tylko wprowadzania odpowiednich modyfikacji w pliku *web.xml*, a więc w *żaden sposób nie będą związane z właściwym kodem źródłowym serwletu*.

Wybawieniem są parametry inicjalizacji

Mieliśmy już do czynienia z parametrami żądania, które pośrednio można przekazywać do metod `doGet()` i `doPost()`; okazuje się jednak, że serwlety mogą korzystać także z parametrów inicjalizacji (inicjalizujących).


W pliku deskryptora wdrożenia (**web.xml**):

```
<servlet>
  <servlet-name>TestParamPiwa</servlet-name>
  <servlet-class>TestParamInicjal</servlet-class>

  <init-param>
    <param-name>emailAdmina</param-name>
    <param-value>cokolwiek@wickedlysmart.com</param-value>
  </init-param>

</servlet>
```

Nadaliśmy naszemu parametrowi inicjalizacji zarówno nazwę, jak i wartość. To proste. Wystarczy, że odpowiednie wyrażenia znajdą się **WEWNĄTRZ** elementu `<servlet>` deskryptora wdrożenia.



W kodzie serwletu:

```
out.println(getServletConfig().getInitParameter("emailAdmina"));
```

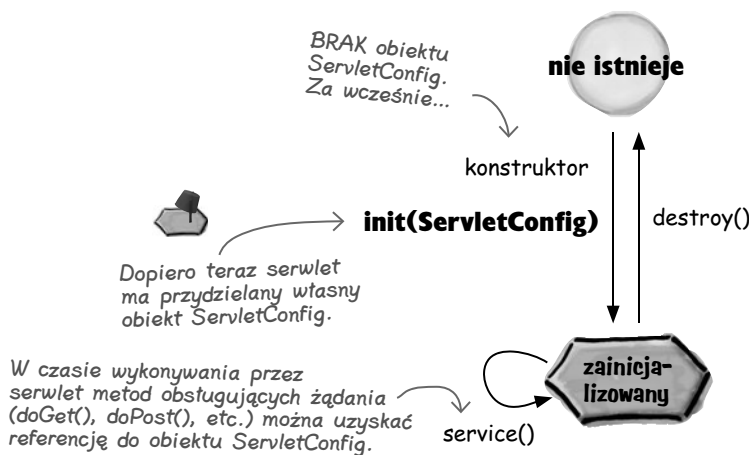
Każdy serwlet dziedziczy metodę `getServletConfig()`.



Metoda `getServletConfig()` zwraca... zaczekajmy z tym... obiekt klasy `ServletConfig`. Co ciekawe, jedną z metod tej klasy jest `getInitParameter()`.

Parametrów inicjalizacji serwletu nie możesz wykorzystywać do czasu, aż ten serwlet zostanie zainicjalizowany

Miałeś już okazję przekonać się, że Twoje serwlety dziedziczą metodę `getServletConfig()`, zatem możesz uzyskać referencję do obiektu `ServletConfig` przez wywołanie tej metody z poziomu dowolnej metody swojego serwletu. Dopiero po uzyskaniu tej referencji będziesz mógł wywoływać metodę `getInitParameter()`. Pamiętaj jednak, że *nie możesz tej metody wywoływać z poziomu swojego konstruktora!* Takie działanie byłoby przedwcześnie w życiu serwletu... do czasu wywołania przez kontener metody `init()` Twój obiekt nie ma wszystkich niezbędnych atrybutów serwletowości.



Kontener inicjalizujący serwlet tworzy dla niego unikatowy obiekt `ServletConfig`.

Kontener „odczytuje” parametry inicjalizacji serwletu z deskryptora wdrożenia, przekazuje je do obiektu `ServletConfig`, po czym przekazuje ten obiekt do metody `init()` serwletu.

Nie ma niemądrych pytań

P: W poprzednim rozdziale wspomniano, że zanim obiekt stanie się pełnoprawnym serwletem, niezbędne jest wykonanie DWÓCH kroków. Była mowa zarówno o obiekcie konfiguracji serwletu (`ServletConfig`), jak i obiekcie kontekstu (`ServletContext`).

U: Tak, to prawda — obiektami `ServletContext` zajmiemy się dosłownie kilka stron dalej. Na razie omawiamy wyłącznie znaczenie obiektu `ServletConfig`, ponieważ właśnie stamtąd można odczytywać parametry inicjalizacji serwletu.

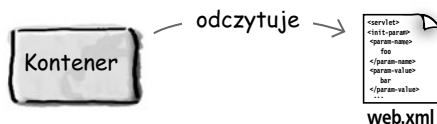
P: Chwileczkę! W ostatnim rozdziale napisaliście, że istnieje możliwość nadpisywania metody `init()` i nikt się nawet nie zająknął o obiekcie `ServletConfig` w roli argumentu tej metody!

U: Nie mówiliśmy, że metoda `init()` otrzymuje na wejściu obiekt `ServletConfig`, ponieważ *nie robi tego nadpisywana przez Ciebie wersja tej metody*. Twoja nadklasa zawiera dwie wersje metody `init()` — jedna z nich otrzymuje w formie jedyne go argumentu obiekt `ServletConfig`, druga (bardziej wygodna) jest metodą bezargumentową. Okazuje się, że dziedziczona po nadklasie metoda `init(ServletConfig)` wywołuje bezargumentową metodę `init()`, zatem jedyną wersją, którą musisz nadpisywać, jest właśnie wersja bezargumentowa. Nie ma reguły, która zabraniałaby Ci nadpisywania metody otrzymującej na wejściu obiekt `ServletConfig`, jeśli jednak to ZROBISZ, lepiej wywołaj metodę `super.init(ServletConfig)`! Warto przy tym pamiętać, że NIE istnieją żadne przesłanki do nadpisywania metody `init(ServletConfig)`, ponieważ zawsze możesz uzyskać dostęp do obiektu `ServletConfig`, wywołując po prostu odziedziczoną metodę `getServletConfig()`.

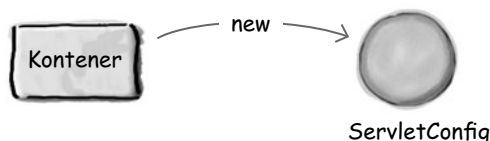
Parametry inicjalizacji serwletu są odczytywane tylko raz — w czasie inicjalizowania serwletu przez kontener

Kiedy w odpowiedzi na żądanie klienta kontener uruchamia serwlet, w pierwszej kolejności odczytuje zawartość deskryptora wdrożenia i generuje pary nazwa-wartość dla nowo utworzonego egzemplarza klasy ServletConfig. Kontener nigdy nie odczytuje parametrów inicjalizacji danego serwletu więcej niż raz! Kiedy parametry zostaną już dołączone do obiektu ServletConfig, kontener nie będzie ich ponownie odczytywał aż do momentu ponownego wdrożenia serwletu. *Pamiętaj o tym.*

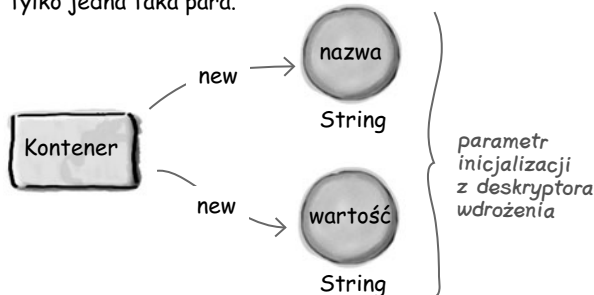
- ❶ Kontener odczytuje zapisy deskryptora wdrożenia dla danego serwletu włącznie z jego parametrami inicjalizacji (zdefiniowanymi w elementach <init-param>).



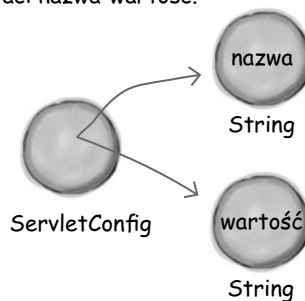
- ❷ Kontener tworzy dla danego serwletu nowy obiekt klasy ServletConfig.



- ❸ Kontener tworzy dla każdego parametru inicjalizacji serwletu parę łańcuchów nazwa-wartość. W tym przypadku zakładamy, że istnieje tylko jedna taka para.



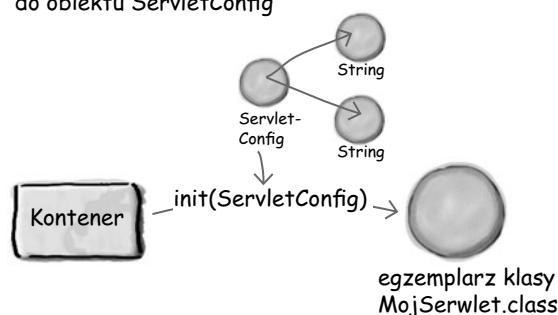
- ❹ Kontener dołącza do obiektu ServletConfig referencje do parametrów inicjalizacji w postaci nazwa-wartość.



- ❺ Kontener dołącza do obiektu ServletConfig referencje do parametrów inicjalizacji w postaci nazwa-wartość



- ❻ Kontener wywołuje metodę init() serwletu, przekazując na jej wejściu referencję do obiektu ServletConfig





Skoro kontener odczytuje parametry inicjalizacji serwletu tylko raz, nadal nie możesz zmienić swojego adresu poczty elektronicznej w czasie życia serwletu. Przedstawione rozwiązanie jest więc bezcelowe.



Mimo wszystko lepsze to niż umieszczanie mojego adresu w kodzie źródłowym serwletu. Teraz, aby uaktualnić swój adres poczty elektronicznej, muszę tylko zmienić plik XML i kliknąć przycisk uruchom ponownie, a nowy adres będzie umieszczony w ServletConfig.

Nie ma niemądrych pytań

P: No dobrze, ale gdzie w kontenerze Tomcat można znaleźć przycisk uruchom ponownie?

U: Tomcat *nie* zawiera jednoprzyciskowego, naprawdę prostego narzędzia administracyjnego, które umożliwiałoby łatwe wdrażanie i ponowne uruchamianie aplikacji internetowych (choć zawiera pewne narzędzia administracyjne). Pomyśl tylko, co jest najgorsze w konieczności modyfikowania parametrów inicjalizacji serwletu. Wprowadzasz niemal natychmiastowe zmiany w pliku *web.xml*, kończysz pracę Tomcata (*bin/shutdown.sh*) i uruchamiasz go ponownie (*bin/startup.sh*). Podczas ponownego uruchamiania Tomcat przeszukuje swój katalog *aplikacji internetowych* i wykorzystuje wszystkie znalezione tam informacje.

P: Zamknięcie i ponowne uruchomienie faktycznie jest bardzo proste, ale co należy

zrobić z działającymi aplikacjami internetowymi? Musimy na chwilę przerwać działanie wszystkich aplikacji!

U: Z technicznego punktu widzenia tak. Wstrzymywanie pracy wszystkich aplikacji internetowych tylko z powodu ponownego uruchamiania pojedynczego serwletu jest rzecz jasna dosyć kłopotliwe, w szczególności jeśli Twoja witryna internetowa obsługuje wielu klientów. Dlatego właśnie większość współczesnych kontenerów WWW oferuje możliwość **ponownego wdrażania w czasie pracy**, które nie wymaga ani ponownego uruchamiania serwera, ani przerywania pracy pozostałych aplikacji internetowych. Także Tomcat zawiera narzędzie *menadżera*, które umożliwia nam uruchamianie, wstrzymywanie i ponowne uruchamianie całych aplikacji internetowych *bez konieczności* ponownego uruchamiania samego kontenera.

Właśnie z tego typu mechanizmów należy korzystać w docelowym środowisku pracy aplikacji. Z drugiej strony, podczas testowania

znacznie prostszym rozwiązaniem będzie ponowne uruchamianie całego Tomcata. Informacje na temat narzędzia zarządzania Tomcatem można znaleźć na następującej stronie internetowej:

<http://tomcat.apache.org/tomcat-6.0-doc/manager-howto.html>

W świecie prawdziwych aplikacji ponowne wdrażanie pozostaje Poważnym Problemem, mimo mechanizmu ponownego wdrażania „na gorąco” — zatrzymywanie i ponowne uruchamianie pojedynczej aplikacji z powodu zmienionej wartości jednego z jej parametrów inicjalizacji nigdy nie jest dobrym rozwiązaniem. Jeśli wartości Twoich parametrów inicjalizacji będą często modyfikowane, lepszym wyjściem będzie zakodowanie w metodach serwletu mechanizmów odczytyjących te wartości z pliku lub bazy danych, jednak takie podejście oznacza zwykle znaczne wydłużenie czasu wykonywania kodu serwletu (w porównaniu z jednorazowym odczytywaniem parametrów w czasie inicjalizacji).

Testowanie obiektu ServletConfig

Głównym zadaniem obiektu ServletConfig jest zapewnianie programiście dostępu do parametrów inicjalizacji. Ten sam obiekt może Ci udostępniać także referencję do obiektu ServletContext, chociaż my będziemy uzyskiwali tę referencję w nieco inny sposób, a metoda getServletName() w niewielu przypadkach ma praktyczne uzasadnienie.

W pliku deskryptora wdrożenia (web.xml):

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
  version="2.4">
  <servlet>
    <servlet-name>TestParamPiwa</servlet-name>
    <servlet-class>com.example.TestParamInicjal</servlet-class>
    <init-param>
      <param-name>emailAdmina</param-name>
      <param-value>cokolwiek@wickedlysmart.com</param-value>
    </init-param>
    <init-param>
      <param-name>emailGlowny</param-name>
      <param-value>cokolwiek_innego@wickedlysmart.com</param-value>
    </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>TestParamPiwa</servlet-name>
    <url-pattern>/Tester.do</url-pattern>
  </servlet-mapping>
</web-app>
```

W klasie serwletu:

```
package com.example;
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class TestParamInicjal extends HttpServlet {
    public void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws IOException, ServletException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("test parametrów inicjalizacji<br>");
        Enumeration e = getServletConfig().getInitParameterNames();
        while(e.hasMoreElements()) {
            out.println("<br>nazwa parametru = " + e.nextElement() + "<br>");
        }
        out.println("główny adres poczty: " + getServletConfig().getInitParameter("emailGlowny"));
        out.println("<br>");
        out.println("adres poczty administratora: " + getServletConfig().getInitParameter("emailAdmina"));
    }
}
```

javax.servlet.ServletConfig

<<interfejs>>	
ServletConfig	
getInitParameter(String)	
Enumeration getInitParameterNames()	
getServletContext()	
getServletName()	←

Większość programistów
nigdy nie używa tej metody.

Ach. Właśnie zdałem sobie sprawę, że przecież w mojej aplikacji używam stron JSP do wizualizacji generowanej odpowiedzi. Czy taka strona JSP może wobec tego „widzieć” parametry inicjalizacji mojego serwletu?



Jak strona JSP może uzyskać dostęp do parametrów inicjalizacji serwletu?

Obiekt *ServletConfig* reprezentuje wyłącznie konfigurację *serwletu* (nie mówi nic o konfiguracji wykorzystywanej strony JSP). Jeśli więc chcesz, aby *pozostałe* składniki Twojej aplikacji internetowej wykorzystywały te same informacje, które definiujesz za pomocą parametrów inicjalizacji serwletu w deskrypcorze wdrożenia, musisz zrobić coś więcej.

A co z rozwiązaniem stosowanym do tej pory w aplikacji porad piwnych? Przekazaliśmy informacje modelu do strony JSP za pośrednictwem atrybutu żądania...

```
// wewnątrz metody doPost()
```

```
String kolor = request.getParameter("kolor");
```

Pamiętasz? Wraz z żądaniem otrzymaliśmy wybrany przez klienta kolor.

```
EkspertPiwny ep = new EkspertPiwny();
```

```
List wynik = ep.getMarki(kolor);
```

Następnie stworzyliśmy obiekt i użyliśmy *MODELU* do uzyskania informacji niezbędnych dla *WIDOKU*.

```
request.setAttribute("styles", wynik);
```

Dopiero potem ustawiamy „atrybut” w obiekcie żądania; strona JSP, do której prześlemy to żądanie, będzie mogła łatwo odczytać wartość tego atrybutu.

Moglibyśmy to zrobić w następujący sposób. Obiekt żądania umożliwia nam ustawianie *atrybutów* (należy je traktować jak zwykle pary nazwa-wartość, gdzie wartością może być dowolny obiekt), które mogą być następnie wykorzystywane przez wszystkie serwlety lub strony JSP otrzymujące dane żądanie. Oznacza to, że atrybut będzie dostępny dla każdego serwletu lub strony JSP, do której tak zmodyfikowane żądanie zostanie przekazane za pomocą obiektu *RequestDispatcher*. Obiekt *RequestDispatcher* szczegółowo omówimy na końcu tego rozdziału, jednak na razie interesuje nas wyłącznie przekazywanie danych (w tym przypadku adresu poczty elektronicznej) pomiędzy składnikami aplikacji internetowej (a więc nie tylko do pojedynczego serwletu).

Ustawianie atrybutów żądania działa... ale tylko w przypadku strony JSP, do której przekazałeś żądanie

W przypadku naszej aplikacji porad piwnych umieszczanie informacji modelu dla żądania klienta w *obiekcie żądania* było o tyle uzasadnione, że bezpośrednio potem następowało *przekazywanie* tego żądania do strony JSP odpowiedzialnej za tworzenie widoku. Ponieważ wywołana w ten sposób strona JSP potrzebowała danych z modelu i ponieważ te dane były związane tylko z tym konkretnym żądaniem, nasza aplikacja działała bez zarzutu.

Taki mechanizm nie rozwiązuje jednak naszego problemu z adresem poczty elektronicznej, ponieważ adres ten być może będzie nam potrzebny w wielu różnych składnikach aplikacji internetowej! *Istnieje* co prawda sposób odczytywania parametrów inicjalizacji przez serwlet i ich zapisywanie w miejscu, z którego będą dostępne dla pozostałych komponentów, jednak z jednej strony takie rozwiązanie wymaga od nas wiedzy, *który* serwlet zawsze będzie uruchamiany jako pierwszy po wdrożeniu aplikacji, z drugiej strony jakakolwiek zmiana w aplikacji internetowej może w takim przypadku wszystko zakłócić. Nie, to też nie jest dobry pomysł.



Zastanawiam się, czy
istnieje coś na kształt parametrów
inicjalizacji serwletów, tyle że
dla całej aplikacji.

Ale naprawdę bardzo mi
zależy na tym, żeby WSZYSTKIE składniki
mojej aplikacji internetowej miały dostęp
do adresu poczty elektronicznej. Stosując
parametry inicjalizacji serwletu, muszę
konfigurować odpowiednie dane w deskrytorze
wdrożenia dla każdego serwletu, a następnie
udostępniać te dane stronom JSP za pośrednictwem
każdego z tych serwletów. Jakże to nudne.
I piekielnie trudne w konserwacji.
Potrzebuję czegoś bardziej globalnego.



Wybawieniem są parametry inicjalizacji kontekstu

Parametry inicjalizacji *kontekstu* bardzo przypominają parametry inicjalizacji *serwletu* — jedyna różnica polega na tym, że parametry kontekstu są dostępne dla całej aplikacji internetowej, a nie dla pojedynczego serwletu. Oznacza to, że każdy serwlet i każda strona JSP należące do danej aplikacji internetowej automatycznie uzyskuje dostęp do parametrów inicjalizacji kontekstu, zatem nie musimy się martwić o ich konfigurowanie w deskrytorze wdrożenia dla każdego serwletu z osobna, a w przypadku zmiany wartości odpowiednie modyfikacje należy wprowadzić tylko w jednym miejscu!

W pliku deskryptora wdrożenia (web.xml):

```
<servlet>
  <servlet-name>TestParamPiwa</servlet-name>
  <servlet-class>TestParamInicjal</servlet-class>
</servlet>

<context-param>
  <param-name>emailAdmina</param-name>
  <param-value>adresklienta@wickedlysmart.com</param-value>
</context-param>
```

Wyciągnęliśmy element `<init-param>` poza element `<servlet>`.

UWAGA!! Element `<context-param>` jest definiowany dla CAŁEJ aplikacji, zatem nie zagnieżdża się go w ramach pojedynczego elementu `<servlet>`!! Zawsze umieszczaj element `<context-param>` wewnątrz elementu `<web-app>`, ale też POZA deklaracją któregośkolwiek z elementów `<servlet>`.

Zdefiniowaliśmy dla parametru inicjalizacji kontekstu elementy `param-name` i `param-value` dokładnie w taki sam sposób jak dla parametrów inicjalizacji serwletu (z tą różnicą, że tym razem wspomniane elementy zadeklarowano w ramach elementu `<context-param>`, a nie `<init-param>`).

W kodzie serwletu:

```
out.println(getServletContext().getInitParameter("emailAdmina"));
```

Każdy serwlet dziedziczy metodę `getServletContext()` (także strony JSP mają specjalny dostęp do obiektu kontekstu).

Metoda `getServletContext()` zwraca (o dziwo) obiekt `ServletContext`, a jedną z metod tego obiektu jest `getInitParameter()`.

LUB:

```
ServletContext context = getServletContext();
out.println(context.getInitParameter("emailAdmina"));
```

W tym przypadku rozbijamy kod na DWA kroki — w pierwszym uzyskujemy referencję do obiektu `ServletContext`, w drugim wywołujemy metodę `getInitParameter()`

Pamiętaj o różnicach dzielących parametry inicjalizacji serwletu od parametrów inicjalizacji kontekstu

W tym miejscu warto przypomnieć najważniejsze różnice pomiędzy parametrami inicjalizacji *kontekstu* a parametrami inicjalizacji *serwletu*. Zwróć szczególną uwagę na fakt, iż w odniesieniu do obu struktur używa się określenia *parametry inicjalizacji*, chociaż tylko dla parametrów inicjalizacji serwletu posługujemy się słowem "init" w konfiguracji deskryptora wdrożenia.

Parametry inicjalizacji kontekstu

Parametry inicjalizacji serwletu

Deskryptor wdrożenia

Wewnątrz elementu <web-app>, ale NIE w ramach konkretnego elementu <servlet>:

```
<web-app ...>
  <context-param>
    <param-name>foo</param-name>
    <param-value>bar</param-value>
  </context-param>

  <!-- pozostałe elementy (włącznie
       z deklaracjami serwletów) -->
</web-app>
```

Zwróć uwagę, że (w przeciwieństwie do przedstawionej obok deklaracji parametru inicjalizacji serwletu) w przedstawionym fragmencie deskryptora wdrożenia w ogóle nie wspominamy o inicjalizacji, chociaż deklarujemy parametry inicjalizacji kontekstu.

Wewnątrz elementu <servlet> właściwego dla konkretnego serwletu aplikacji:

```
<servlet>
  <servlet-name>
    TestParamPiwa
  </servlet-name>
  <servlet-class>
    TestParamInicjal
  </servlet-class>

  <init-param>
    <param-name>foo</param-name>
    <param-value>bar</param-value>
  </init-param>

  <!-- pozostałe elementy -->
</servlet>
```

Kod serwletu

```
getServletContext().getInitParameter("foo");
```

```
getServletConfig().getInitParameter("foo");
```

Wykorzystujemy dwie tak samo nazwane metody!

Dostępność

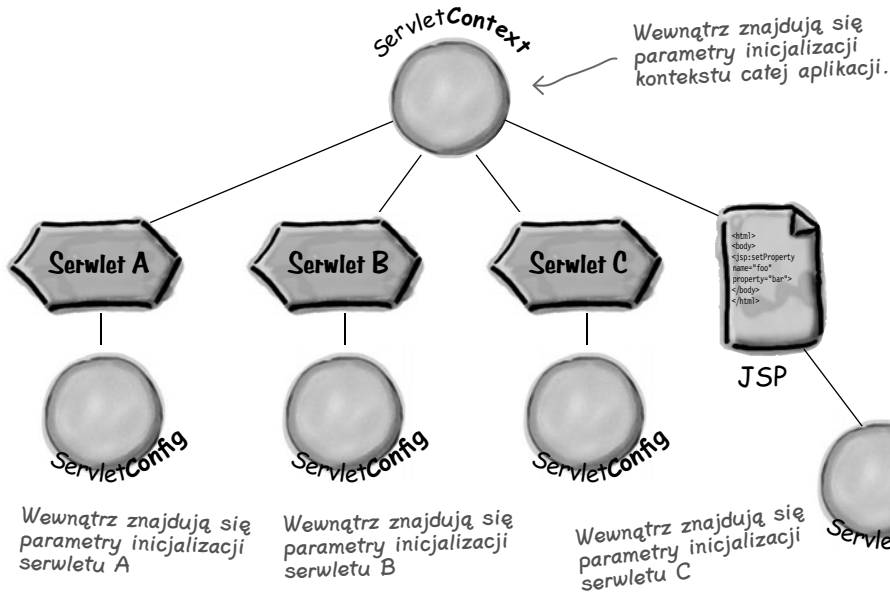
We wszystkich serwletach i stronach JSP, które należą do danej aplikacji internetowej.

Tylko w serwlecie, dla którego skonfigurowano dany element <init-param>.

(Chociaż nasz serwlet może udostępniać te dane innym komponentom aplikacji przez ich umieszczanie w atrybutach).

Dla pojedynczego serwletu może istnieć jeden obiekt `ServletConfig`; dla pojedynczej aplikacji może istnieć jeden obiekt `ServletContext`

Istnieje tylko jeden obiekt `ServletContext` dla całej aplikacji internetowej — wszystkie składniki tej aplikacji współużytkują ten obiekt. Każdy serwlet tej aplikacji ma jednak przypisany własny obiekt `ServletConfig`. Kontener już w momencie uruchamiania aplikacji internetowej tworzy i udostępnia wszystkim serwletom i stronom JSP (które stają się serwletami) obiekt `ServletContext`.



Inicjalizacja aplikacji internetowej:

- Kontener odczytuje zawartość deskryptora wdrożenia i tworzy po jednej parze łańcuchów nazwa-wartość dla każdego znanego elementu `<context-param>`.
- Kontener tworzy nowy obiekt `ServletContext`.
- Kontener tworzy w nowo konstruowanym obiekcie `ServletContext` referencję do każdej pary nazwa-wartość reprezentującej parametr inicjalizacji kontekstu.
- Każdy serwlet i każda strona JSP będąca częścią tej samej aplikacji internetowej ma dostęp do tego samego obiektu `ServletContext`.

Tak, strony JSP są przekształcane w serwlety, zatem także one otrzymują własne obiekty `ServletConfig`.



Oglądaj to!

Parametrów obiektu `ServletConfig` nie należy mylić z parametrami obiektu `ServletContext`!

Zanim przystąpisz do egzaminu, musisz naprawdę dobrze zrozumieć różnice pomiędzy tymi parametrami, a jest to dosyć skomplikowana materia. MUSISZ wiedzieć, że zarówno obiekt `ServletConfig`, jak i obiekt `ServletContext` zawiera parametry inicjalizacji oraz że oba obiekty udostępniają tak samo nazwaną metodę zwracającą parametry inicjalizacji: `getInitParameter()`. ALE... musisz także wiedzieć, że o ile parametry inicjalizacji kontekstu są definiowane wewnątrz elementów `<context-param>` (nie wewnątrz elementów `<servlet>`), o tyle parametry inicjalizacji serwletu definiuje się wewnątrz podelementów `<init-param>` elementów `<servlet>` deskryptora wdrożenia.



Oglądaj to!

W przypadku aplikacji rozproszonych stosuje się po jednym obiekcie `ServletContext` na każdą wirtualną maszynę Javy!

Jeśli Twoja aplikacja jest rozproszona pomiędzy wiele serwerów (być może w środowisku podzielonym na klastry), w praktyce MOŻLIWE jest wykorzystywanie więcej niż jednego obiektu `ServletContext`. **Zasada pojedynczego obiektu `ServletContext` dla każdej aplikacji internetowej ma zastosowanie tylko w przypadku pojedynczej wirtualnej maszyny Javy!** W środowisku rozproszonym będziesz dysponował **po jednym obiekcie `ServletContext` dla każdej wirtualnej maszyny Javy**. Takie rozwiązanie nie musi oczywiście być źródłem problemów, ale zanim przystąpisz do projektowania rozproszonej aplikacji internetowej, powinieneś dokładnie przeanalizować konsekwencje stosowania różnych kontekstów w poszczególnych wirtualnych maszynach Javy.

Nie ma niemądrych pytań

P: Skąd wziąć się brak konsekwencji w stosowanych schematach nazewnictwa? Jak to się stało, że w deskrytorze wdrożenia definiujemy różne elementy `<context-param>` i `<init-param>`, które są następnie odczytywane w kodzie serwletu za pomocą TEJ SAMEJ metody `getInitParameter()`?

U: Po prostu autorzy interfejsu nie poprosili nas w porę o pomoc w opracowywaniu schematów nazewnictwa. Gdyby to zrobili, z pewnością powiedzielibyśmy im, że należałoby raczej użyć metod `getInitParameter()` i `getContextParameter()`, które w większym stopniu odpowiadałyby elementom XML definiowanym w deskrytorze wdrożenia. Autorzy interfejsu mogli także użyć dwóch różnych elementów XML — np. `<servlet-init-param>` i `<context-init-param>`. Ale nie, to rozwiązanie mogłoby zniweczyć wszystkie dotychczasowe wysiłki na rzecz zachowania maksymalnej prostoty deskryptorów wdrożeń.

P: Po co w ogóle miałbym używać elementu `<init-param>`? Czy nie byłoby lepszym rozwiązaniem stosowanie we wszystkich przypadkach elementu `<context-param>`, dzięki któremu pozostałe składniki mojej aplikacji mogłyby wielokrotnie wykorzystywać zdefiniowane w ten sposób wartości, a ja nie musiałbym powielać kodu XML dla każdej deklaracji serwletu?

U: Wszystko zależy od tego, która część Twojej aplikacji internetowej ma mieć dostęp do tych wartości. Logika aplikacji może wymagać pewnych ograniczeń (np. do jednego serwletu) w dostępie do określonych danych. Zwykle jednak programiści uważają wykorzystywane na poziomie całej aplikacji parametry inicjalizacji *kontekstu* za znacznie bardziej przydatne od związanych z pojedynczymi serwletami parametrów inicjalizacji *serwletów*. Prawdopodobnie najbardziej popularnym zastosowaniem parametrów kontekstu jest składowanie nazw przeszukiwania baz danych. W takim przypadku wszystkie komponenty Twojej aplikacji powinny mieć dostęp do właściwej nazwy, a w przypadku konieczności jej zmodyfikowania niezbędne zmiany powinny się ograniczać do jednego miejsca (deskryptora wdrożenia).

P: Co stanie się, jeśli nadam parametrowi inicjalizacji kontekstu taką samą nazwę jak używanemu w tej samej aplikacji internetowej parametrowi inicjalizacji serwletu?

U: Miniaturowa czarna dziura uzyskana w cudowny sposób w laboratoriach badawczych w New Jersey wymknie się spod kontroli naukowców, opadnie do jądra Ziemi i zniszczy całą planetę.

Być może nic takiego nie będzie miało miejsca, ponieważ nie występuje konflikt przestrzeni nazw (parametry są przecież udostępniane za pośrednictwem dwóch różnych obiektów: `ServletContext` i `ServletConfig`).

P: Jeśli zmodyfikujesz plik XML i zmienisz wartość któregoś z parametrów inicjalizacji (serwletu lub kontekstu), kiedy wprowadzone zmiany będą widoczne dla serwletu lub pozostałych składników aplikacji internetowej?

U: DOPIERO po ponownym wdrożeniu danej aplikacji internetowej. Pamiętaj (o czym już wspominaliśmy w tym rozdziale), że serwlet jest inicjalizowany tylko raz, na początku swojego życia, oraz że ta jednorazowa inicjalizacja obejmuje między innymi przydział obiektów `ServletConfig` i `ServletContext`. Kontener odczytuje potrzebne wartości z deskryptora wdrożenia w czasie tworzenia obu obiektów.

P: Czy nie można rozwiązać tego problemu inaczej, np. ustawiając wartości w czasie wykonywania aplikacji? Musi przecież istnieć jakiś interfejs API, który umożliwi mi dynamiczne modyfikowanie tych wartości...

U: Nie, taki interfejs nie istnieje. Wystarczy się przyjrzeć metodom interfejsów `ServletContext` i `ServletConfig`; znajdziesz tam metodę zwracającą parametry (`getInitParameter()`), ale z pewnością nie znajdziesz metod ustawiających parametry. Nie istnieje metoda `setInitParameter()`.

P: To fatalnie.

U: Cały czas mówimy o parametrach *inicjalizacji*. Dopiero kiedy zaczniesz je traktować jak typowe stałe czasu wdrażania aplikacji, będziesz ten mechanizm obserwował z właściwej perspektywy. Opisywana reguła jest na tyle ważna, że warto ją jeszcze raz powtórzyć i zapisać grubszą czcionką:

Traktuj parametry inicjalizacji jak stałe czasu wdrażania aplikacji!

Możesz te parametry odczytywać w czasie działania aplikacji, ale nie możesz ich zmieniać. Nie istnieje metoda `setInitParameter()`.



Ćwiczenie



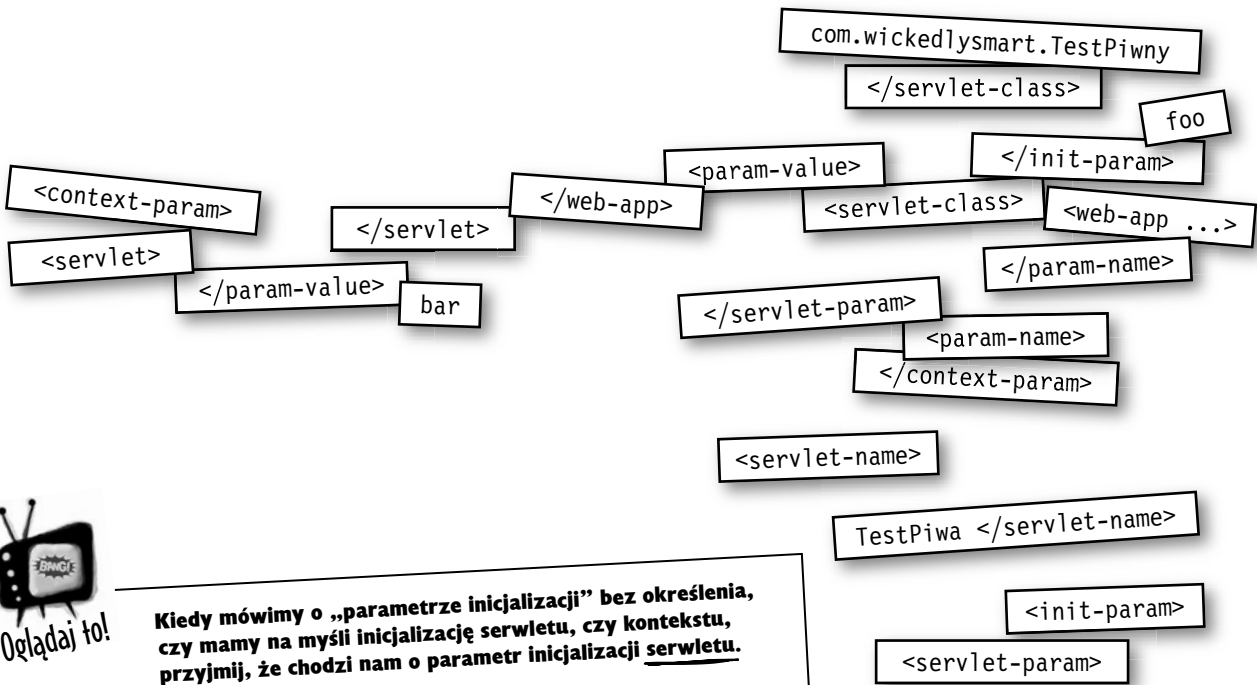
Magnesiki z kodem

Poprządkuj magnesiki na lodówce w taki sposób, aby utworzyły deskryptor wdrożenia deklarujący parametr, który będzie odpowiadał następującemu fragmentowi kodu serwletu:

```
getServletContext().getInitParameter("foo");
```

Nie będziesz potrzebował wszystkich magnesików!

(Uwaga: kiedy widzisz znacznik `<web-app ...>`, pamiętaj, że jest to skrócona wersja tego elementu, dzięki której oszczędzamy mnóstwo miejsca na stronie. Oczywiście nie będziesz mógł wdrożyć takiego deskryptora wdrożenia, dopóki w znaczniku `<web-app>` nie umieścisz wszystkich wymaganych atrybutów.)



Oglądaj to!

Kiedy mówimy o „parametrze inicjalizacji” bez określenia, czy mamy na myśli inicjalizację serwletu, czy kontekstu, przyjmij, że chodzi nam o parametr inicjalizacji serwletu.

Niektórzy programiści używają pojęcia „parametr inicjalizacji” w znaczeniu „parametru inicjalizacji serwletu” oraz terminu „parametr kontekstu” lub nawet „parametr aplikacji” do określania „parametru inicjalizacji kontekstu”. Mimo że OBA pojęcia odnoszą się do parametrów inicjalizacji oraz oba są zwracane przez metodę `getInitParameter()`, musisz pamiętać, że tylko parametry inicjalizacji SERWLETU konfiguruje się w deskrytorze wdrożenia właśnie jako parametry inicjalizacji, zatem wyrażenie „parametr inicjalizacji” oznacza w domyśle „parametr inicjalizacji serwletu”. Wiemy, że jako programista będziesz bardziej precyzyjny od reszty populacji i zawsze będziesz jasno deklarował, czy chodzi Ci o parametr inicjalizacji serwletu, czy może o parametr inicjalizacji kontekstu.

Do czego jeszcze możemy wykorzystać interfejs **ServletContext**?

Interfejs **ServletContext** służy do łączenia stron JSP i (lub) serwletów zarówno z kontenerem WWW, jak i z pozostałymi komponentami tej samej aplikacji internetowej. Poniżej przedstawiono kilka wybranych metod tego interfejsu. Metody, na które powinieneś zwrócić szczególną uwagę przed przystąpieniem do egzaminu, oznaczyliśmy pogrubieniem.



Zwracają parametry inicjalizacji oraz umożliwiają ustawianie i zwracanie atrybutów.

Zwracają informacje na temat serwera (kontenera).

Dopisuje tańcuch do pliku dziennika serwera (w zależności od jego producenta) lub przekazuje ten tańcuch na standardowe wyjście.

```
<<interfejs>>
ServletContext

getInitParameter(String)
getInitParameterNames()
getAttribute(String)
getAttributeNames()
setAttribute(String, Object)
removeAttribute(String)
-----
getMajorVersion()
getServerInfo()
-----
getRealPath(String)
getResourceAsStream(String)
getRequestDispatcher(String)
-----
log(String)
// więcej metod
```

Różnice pomiędzy parametrami a atrybutami omówimy kilka stron dalej.

← Klasę **RequestDispatcher** omówimy w dalszej części tego rozdziału.

javax.servlet.ServletContext



Oglądaj to!

Możesz uzyskać dostęp do obiektu `ServletContext` na dwa różne sposoby...

Wykorzystywany w serwlecie obiekt `ServletConfig` zawsze zawiera referencję do właściwego dla tego serwletu obiektu `ServletContext`. Nie powinieneś się więc zdziwić, jeśli na egzaminie będziesz miał do czynienia z następującym fragmentem kodu serwletu:

```
getServletConfig().getServletContext().getInitParameter()
```

Takie wywołanie nie tylko jest prawidłowe, ale także jest równoważne poleceniu:

```
this.getServletContext().getInitParameter()
```

Jedyny przypadek, w którym **POTRZEBUJEMY** w kodzie serwletu pośrednictwa obiektu `ServletConfig` w dostępie do obiektu `ServletContext`, ma miejsce wtedy, gdy programujemy klasę serwletu, która nie rozszerza ani klasy `HttpServlet`, ani klasy `GenericServlet` (metodę `getServletContext()` dziedziczymy właśnie po klasie bazowej `GenericServlet`). Warto jednak pamiętać, że szanse na to, by **KTOKOLWIEK** stosował inne rozwiązanie niż serwlet HTTP... no cóż, asymptotycznie dążą do zera. W zdecydowanej większości przypadków wystarczy więc po prostu wywołać metodę `getServletContext()`; nie dziw się jednak, jeśli spotkasz w kodzie fragment wykorzystujący obiekt `ServletConfig` w roli pośrednika w przekazywaniu referencji do obiektu kontekstu.

Jak należy postępować w sytuacji, gdy dany kod znajduje się wewnątrz jakiejś klasy, która **NIE** jest serwletem (np. klasy pomocniczej)? Programista może przecież przekazać obiekt `ServletConfig` do klasy, w której zostanie użyta metoda `getServletContext()` do uzyskania referencji do obiektu `ServletContext`.

P: W jaki sposób wszystkie składniki aplikacji internetowej uzyskują dostęp do ich własnych obiektów `ServletContext`?

U: W przypadku serwletów odpowiedź już teraz powinna Ci być znana: wystarczy wywołać odziedziczoną metodę `getServletContext()`.

W przypadku stron JSP procedura wygląda nieco inaczej — strony te zawierają coś, co programiści nazywają „obektami niejawnymi” (jednym z takich obiektów jest właśnie `ServletContext`). Więcej szczegółów związanych z rzeczywistymi technikami wykorzystywania obiektu `ServletContext` przez strony JSP znajdziesz w rozdziałach poświęconych technologii JSP.

P: Zatem uzyskujesz dostęp do wbudowanego mechanizmu rejestrowania zdarzeń za pośrednictwem obiektu kontekstu? Przypnij, że brzmi to **BARDZO** obiecująco!

U: Hm, nie do końca. Nie jest to możliwe, chyba że masz bardzo małą i prostą aplikację internetową. Istnieją znacznie lepsze sposoby obsługi mechanizmu rejestrowania zdarzeń w dzienniku. Najbardziej popularnym i jednocześnie dosyć zaawansowanym mechanizmem tego typu jest Log4j — można go znaleźć na witrynie internetowej projektu Apache:

http://jakarta.apache.org/log4j

W swoich aplikacjach możesz także używać dostępnego w pakiecie `java.util.logging` (dodanym do środowiska J2SE w wersji 1.4) interfejsu API dla mechanizmu rejestrowania zdarzeń.

Do prostych eksperymentów w zupełności wystarczy stosowanie metody `log()` obiektu `ServletContext()`, jednak w rzeczywistym środowisku niemal na pewno będziesz potrzebował czegoś bardziej zaawansowanego. Szczegółowe omówienie mechanizmów rejestrowania zdarzeń (opartych zarówno na pakiecie Log4j, jak i na innych technikach) można znaleźć w książce *Java Servlet & JSP Cookbook* wydawnictwa O'Reilly.

Rejestrowanie zdarzeń w pliku dziennika nie należy do celów egzaminu, ale jest ważnym elementem aplikacji internetowych. Na szczęście (o czym się z pewnością przekonasz) używanie odpowiednich interfejsów API jest dosyć proste.

Mam dość oczerniania interfejsu ServletContext, ale faktem jest, że przechowywane w jego obiektach parametry muszą być ŁAŃCUCHAMI! To dla mnie poważny problem! Co będzie, kiedy zapragnę zainicjalizować moją aplikację komponentem DataSource bazy danych, który powinien być dostępny dla wszystkich serwletów?



Co powinniśmy zrobić, jeśli będziemy chcieli użyć parametru inicjalizacji aplikacji mającego postać komponentu DataSource bazy danych?

Parametry kontekstu mogą mieć wyłącznie postać łańcuchów (obiektów klasy String). Nie moglibyśmy przecież precyzyjnie zadeklarować obiektu *Pies* w dokumencie XML, jakim jest deskryptor wdrożenia (w praktyce istnieje możliwość reprezentowania w formacie XML serializowanych obiektów, jednak aktualna specyfikacja serwletów nie przewiduje tego rodzaju rozwiązań... być może odpowiednie elementy zostaną uwzględnione w przyszłości).

Co powinniśmy zrobić, jeśli staniemy przed absolutną koniecznością zapewnienia możliwości współużytkowania przez wszystkie składniki aplikacji internetowej wspólnego połączenia z bazą danych? Z pewnością możemy umieścić identyfikator komponentu DataSource w parametrze inicjalizacji kontekstu, co prawdopodobnie stanowi najbardziej popularne zastosowanie tego typu parametrów.

Ale **kto będzie wówczas odpowiedzialny za przekształcanie parametru łańcuchowego w rzeczywistą referencję do komponentu DataSource** udostępnianego wszystkim składnikom danej aplikacji internetowej?

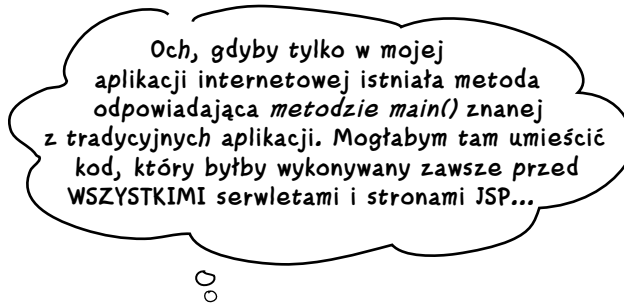
W praktyce nie ma możliwości umieszczenia odpowiedniego kodu w serwlecie, ponieważ w takim przypadku nie da się z góry określić, który serwlet będzie Tym Jedynym, Który Odczytuje Identyfikator DataSource i Umieszcza Go W Atrybucie. Czy *naprawdę* chcesz podjąć próbę zagwarantowania, że jeden serwlet będzie zawsze uruchamiany jako pierwszy? Przemyśl to.



WYTEŻ UMYSŁ

Jak Twoim zdaniem należałoby rozwiązać ten problem?

Jak zainicjalizowałbyś aplikację internetową z jakimś obiektem? Przyjmij, że potrzebujesz łańcuchowego parametru inicjalizacji kontekstu, który wykorzystasz do utworzenia tego obiektu (spróbuj dokonać analizy na przykładzie związanym z bazą danych).



Tym, czego potrzebujemy, jest obiekt nasłuchujący

Cały problem sprowadza się do zastosowania mechanizmu nasłuchującego zdarzeń inicjalizacji kontekstu, który umożliwiłby nam natychmiastowe odczytanie parametrów inicjalizacji kontekstu i **wykonanie pewnego kodu, zanim pozostałe składniki aplikacji będą mogły obsługiwać żądania klienta.**

Potrzebujemy czegoś, co będzie działało w tle i oczekiwało na sygnał o uruchomieniu naszej aplikacji.

Ale która część tej aplikacji mogłaby pełnić tę funkcję? Nie chcemy obarczać odpowiedzialnością za tego typu działania serwletu, ponieważ nasłuchiwanie zdarzeń inicjalizacji kontekstu z pewnością nie należy do jego obowiązków.

Nie byłby to żaden problem w tradycyjnej, autonomicznej aplikacji Javy, ponieważ mamy tam do dyspozycji metodę `main()`! Wciąż nie wiemy jednak, jakie rozwiązanie należałoby zastosować w przypadku serwletów.

Z pewnością potrzebujemy *czegoś innego*. Problemu nie rozwiązuje ani serwlet, ani strona JSP, ale zupełnie inny rodzaj obiektu Javy, którego jedynym celem jest inicjalizowanie aplikacji (być może także jej deinicjalizowanie, zwalnianie zasobów w momencie odkrycia zgonu aplikacji...).

Potrzebujemy interfejsu `ServletContextListener`

Mozemy stworzyć osobną klasę, która nie będzie ani serwletem, ani stroną JSP, i która umożliwi nasłuchiwanie pod kątem dwóch kluczowych zdarzeń w cyklu życia obiektu `ServletContext`: inicjalizacji (tworzenia) i destrukcji. Taka osobna klasa powinna implementować interfejs `javax.servlet.ServletContextListener`.

Potrzebujemy odrębnego obiektu odpowiedzialnego za następujące działania:

- Rejestrowanie momentu inicjalizacji kontekstu (wdrażania aplikacji internetowej).
 - Pobranie parametru inicjalizacji kontekstu z obiektu `ServletContext`.
 - Użycie parametru inicjalizacji z identyfikatorem bazy danych do utworzenia połączenia.
 - Umieszczenie obiektu połączenia z bazą danych w postaci atrybutu, aby pozostałe składniki aplikacji internetowej miały do niej dostęp.
- Rejestrowanie momentu niszczenia kontekstu (usunięcia lub awarii aplikacji).
 - Zamknięcie połączenia z bazą danych.



`javax.servlet.ServletContextListener`

Klasa `ServletContextListener`:

```
import javax.servlet.*;
```

Interfejs `ServletContextListener`
należy do pakietu `javax.servlet`.

Nasłuchiwanie zdarzeń kontekstu jest proste, wystarczy zaimplementować interfejs `ServletContextListener`.

```
public class MojServletContextListener implements ServletContextListener {
```

```
    public void contextInitialized(ServletContextEvent event) {
```

```
        // kod inicjalizujący połączenie z bazą danych i
```

```
        // umieszczający odpowiedni obiekt w atrybucie kontekstu
```

```
    }
```

```
    public void contextDestroyed(ServletContextEvent event) {
```

```
        // kod zamykający połączenie z bazą danych
```

```
    }
```

```
}
```

Te dwie metody są wywoływane w odpowiedzi na określone zdarzenia. Obie metody reagują na zdarzenie `ServletContextEvent`.

Dobrze, mam już klasę
nasłuchującą. Co teraz powinienem
zrobić? Kto stworzy obiekt tej klasy? Jak
zarejestrować obiekt nasłuchujący zdarzeń?
W jaki sposób obiekt nasłuchujący może
ustawić atrybut we właściwym obiekcie
ServletContext?



WYTEŻ UMYŚŁ

Jak sądzisz? W jaki sposób przedstawiony mechanizm może uczynić z klasy nasłuchującej część określonej aplikacji internetowej?

Wskazówka: zastanów się, jak przekazujesz kontenerowi informacje o tym, że pozostałe komponenty należą do aplikacji internetowej. Gdzie kontener może odkryć Twoją klasę nasłuchującą?

Przewodnik: prosty obiekt ServletContextListener

Przeanalizujemy teraz kolejne kroki procesu tworzenia i uruchamiania klasy implementującej interfejs `ServletContextListener`. Będzie to prosta klasa testowa, dzięki której przekonamy się, jak wszystkie elementy naszego rozwiązania mogą ze sobą współpracować; zrezygnowaliśmy z dalszej analizy przykładu połączenia z bazą danych, ponieważ wymagałoby to dodatkowego wysiłku związanego z konfiguracją odpowiedniego środowiska. Opisane poniżej kroki pozostają takie same *niezależnie* od tego, jaki kod umieścimy w metodach zwrótnych klasy nasłuchującej.

W tym przykładzie przekształcimy łańcuchowy parametr inicjalizacji w rzeczywisty obiekt — psa. Pierwszym zadaniem obiektu nasłuchującego jest odczytanie parametru inicjalizacji kontekstu reprezentującego rasę psa (bigiel, pudel itp.), dopiero potem gotowy łańcuch jest wykorzystywany do skonstruowania obiektu psa. Obiekt nasłuchujący upycha następnie obiekt psa w atrybucie obiektu `ServletContext`, aby możliwe było jego uzyskanie z poziomu serwletu.

Kluczowe znaczenie ma dostęp serwletu do wspólnego *obiekту* aplikacji (w tym przypadku reprezentującego psa) bez konieczności odczytywania parametrów kontekstu. To, czy współużytkowany obiekt jest psem, czy połączeniem z bazą danych, nie ma tak naprawdę znaczenia. Najważniejszym elementem decydującym o skuteczności tego rozwiązania jest utworzenie pojedynczego obiektu, który jest następnie udostępniany wszystkim składnikom naszej aplikacji internetowej.



W tym przykładzie umieścimy w obiekcie `ServletContext` naszego psa.

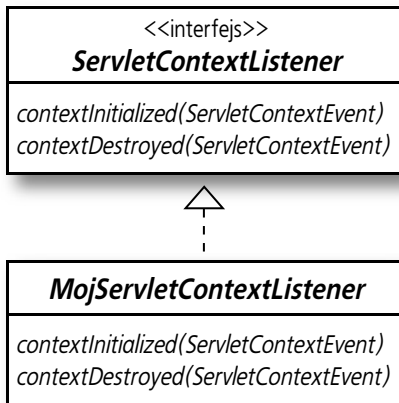
Nasz przykład z psem:

- Obiekt nasłuchujący prosi obiekt `ServletContextEvent` o przekazanie referencji do wykorzystywanego w aplikacji obiektu `ServletContext`.
- Obiekt nasłuchujący wykorzystuje otrzymaną właśnie referencję do obiektu `ServletContext` do uzyskania parametru inicjalizacji kontekstu "rasa", który ma postać łańcucha reprezentującego rasę psa.
- Obiekt nasłuchujący wykorzystuje łańcuch rasy psa do skonstruowania odpowiedniego obiektu psa.
- Obiekt nasłuchujący wykorzystuje referencję do obiektu `ServletContext` do *ustawienia* w tym obiekcie atrybutu reprezentującego psa.
- Serwlet testowy naszej aplikacji internetowej *uzyskuje* obiekt psa z obiektu `ServletContext` i wywołuje należącą do odczytanego obiektu metodę `getRasa()`.

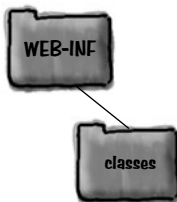
Tworzenie i stosowanie obiektu nasłuchującego zdarzeń kontekstu

Być może wciąż się zastanawiasz, jak to możliwe, że kontener odkrywa i wykorzystuje obiekt nasłuchujący... **Obiekt nasłuchujący konfigurujemy dokładnie w taki sam sposób, w jaki definiujemy na potrzeby kontenera pozostałe składniki aplikacji internetowej, a więc za pośrednictwem deskryptora wdrożenia web.xml!**

1 Tworzymy klasę nasłuchującą



2 Umieszczamy plik klasy w katalogu WEB-INF/classes



(Nie jest to jednak JEDYNE miejsce dla tego pliku... katalog `WEB-INF/classes` jest tylko jednym z wielu miejsc, w którym kontener może szukać plików klas. Pozostałe katalogi omówimy w rozdziale poświęconym wdrażaniu aplikacji.)

3 Umieszczamy element <listener> w deskrytorze wdrożenia web.xml

```

<listener>
  <listener-class>
    com.example.MojServletContextListener
  </listener-class>
</listener>
  
```

Pytanie dla Ciebie: w której części deskryptora wdrożenia należy umieścić element `<listener>`? Czy powinien się znaleźć w elemencie `<servlet>`, czy może w elemencie `<web-app>`? Przemysł to.

Aby nasłuchiwać zdarzeń związanych z obiektem `ServletContext`, musisz napisać klasę implementującą interfejs `ServletContextListener`, umieścić go w katalogu `WEB-INF/classes` oraz poinformować o tym kontener przez umieszczenie elementu `<listener>` w deskrytorze wdrożenia.

Potrzebujemy trzech klas i pojedynczego deskryptora wdrożenia

Aby nasz testowy przykład obiektu nasłuchującego kontekstu był kompletny, musimy nie tylko napisać odpowiednie klasy, ale także przygotować plik *web.xml*.

Aby ułatwić proces testowania aplikacji, umieścimy wszystkie klasy w tym samym pakiecie — *com.example*.

1 Interfejs ServletContextListener

MojServletContextListener.java

Ta klasa implementuje interfejs ServletContextListener, uzyskuje dostęp do parametrów inicjalizacji kontekstu, tworzy obiekt klasy Pies i ustawia atrybut kontekstu reprezentujący ten obiekt.

2 Klasa atrybutu

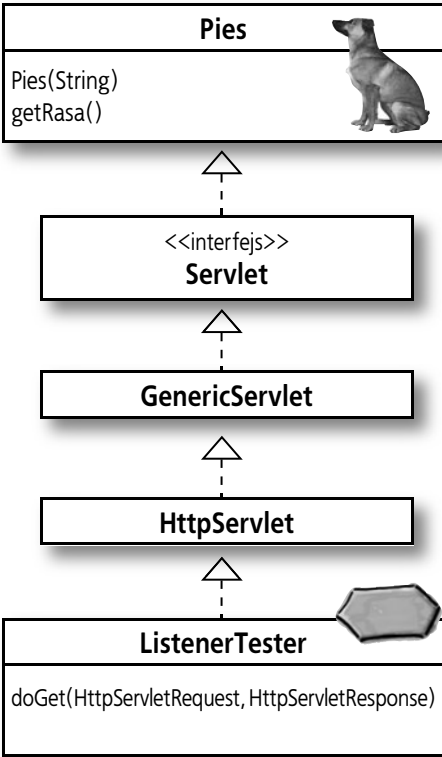
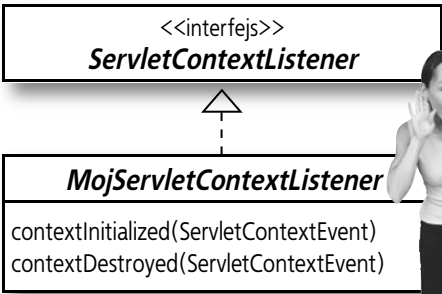
Pies.java

Klasa Pies jest po prostu zwykłą, tradycyjną klasą Javy. Obiekt tej klasy zostanie przekształcony w wartość atrybutu — za tworzenie i ustawianie tego atrybutu w obiekcie ServletContext odpowiada klasa implementująca interfejs ServletContextListener.

3 Serwlet

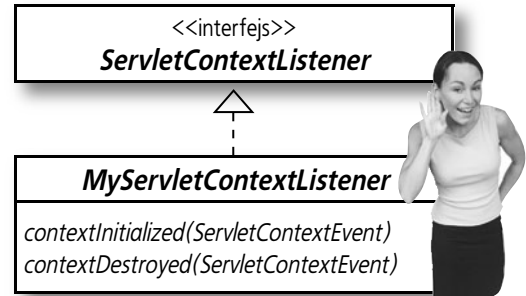
ListenerTester.java

Klasa ListenerTester rozszerza klasę bazową HttpServlet. Zadaniem tej klasy jest sprawdzenie, czy obiekt nasłuchujący zadziałał prawidłowo, a więc określenie, czy odczytał z kontekstu atrybut pies, wywołał metodę getRasa() obiektu Pies oraz zapisał wynik w obiekcie odpowiedzi (będzie wówczas widoczny w oknie przeglądarki).



Pisanie klasy nasłuchującej

Nasza klasa powinna działać podobnie do innych typów klas nasłuchujących, z którymi być może miałeś już do czynienia (np. z klasami obsługującymi zdarzenia graficznego interfejsu użytkownika opartego na komponentach Swing). Pamiętaj, że musimy tylko uzyskać dostęp do parametrów inicjalizacji kontekstu (aby móc określić rasę psa), stworzyć obiekt klasy Pies oraz umieścić nowo utworzony obiekt w postaci atrybutu obiektu kontekstu.



```
package com.example;
```

```
import javax.servlet.*;
```

Implementuje interfejs javax.servlet.
ServletContextListener.

```
public class MojServletContextListener implements ServletContextListener {
```

```
    public void contextInitialized(ServletContextEvent event) {
```

```
        ServletContext sc = event.getServletContext();
```

Prosi obiekt zdarzenia o dostęp do
obiektu ServletContext.

```
        String rasaPsa = sc.getInitParameter("rasa");
```

Wykorzystuje kontekst do uzyskania
parametru inicjalizacji.

```
        Pies p = new Pies(rasaPsa);
```

Tworzy nowy obiekt klasy Pies.

```
        sc.setAttribute("pies", p);
```

Wykorzystuje kontekst do ustawienia atrybutu
(pary nazwa-obiekt) reprezentującego obiekt klasy
Pies. Od tej chwili pozostałe składniki aplikacji
internetowej będą mogły odczytać wartość tego
atrybutu (obiekt klasy Pies).

```
    public void contextDestroyed(ServletContextEvent event) {
```

```
        // w tym miejscu nie musimy podejmować żadnych działań
```

```
    }
```

```
}
```

Nie musimy umieszczać w tym miejscu żadnego
kodu. Obiekt klasy Pies nie musi być usuwany...
zniszczenie obiektu kontekstu jest równoważne
usunięciu z pamięci całej aplikacji, więc
z obiektem reprezentującym psa.

Pisanie klasy atrybutu (Pies)

To oczywiste, potrzebujemy jeszcze klasy Pies — klasy reprezentującej obiekt, który będzie umieszczany w obiekcie ServletContext bezpośrednio po odczytaniu odpowiedniego parametru inicjalizacji kontekstu.

package com.example;

Nie ma tu nic specjalnego,
tylko zwykły kod Javy.


```
public class Pies {  
    private String rasa;
```

```
    public Pies(String rasa) {  
        this.rasa = rasa;  
    }
```

(Użyjemy parametru inicjalizacji kontekstu w postaci argumentu konstruktora klasy Pies.)

```
    public String getRasa() {  
        return rasa;  
    }  
}
```

Nasz serwet uzyska referencję do obiektu Pies z kontekstu aplikacji (obektu, który został zapisany w formie atrybutu przez obiekt nastuchujący), wywoła metodę getRasa() tego obiektu i zapisze rasę psa w obiekcie odpowiedzi, aby została wyświetlona w oknie przeglądarki.

Pies	
Pies(String) getRasa()	

P: Wydaje mi się, że przeczytałem gdzieś, że atrybuty serwletu muszą implementować interfejs Serializable...

U: Interesujące pytanie. Istnieje wiele różnych typów atrybutów i to, czy dany atrybut powinien być serializowalny (ang. *serializable*) ma znaczenie wyłącznie w przypadku atrybutów sesyjnych. Jedynym scenariuszem, w którym serializacja atrybutów jest istotna, jest rozproszenie aplikacji pomiędzy więcej niż jedną wirtualną maszynę Javy. Więcej szczegółowych informacji na ten temat znajdziesz w rozdziale poświęconym sesjom.

Z technicznego punktu widzenia zapewnianie serializacji jakichkolwiek atrybutów (włącznie z atrybutami sesyjnymi) nie jest konieczne, choć można oczywiście rozważyć domyślną implementację serializacji dla wszystkich atrybutów, chyba że masz naprawdę istotne powody, by tego NIE robić.

Przemyśl to — czy naprawdę jesteś pewien, że nikt nigdy nie będzie chciał wykorzystywać obiektów tego typu w postaci argumentów lub zwracać wartości w formie części zdalnego wywołania metody? Czy rzeczywiście możesz zagwarantować, że żaden z przyszłych użytkowników tej klasy (w tym przypadku klasy Pies) nigdy nie będzie jej wykorzystywał w środowisku rozproszonym?

Można więc przyjąć, że choć zapewnienie serializacji atrybutów nie jest wymagane, prawdopodobnie powinniśmy o to dbać zawsze wtedy, gdy jest to możliwe.

Pisanie klasy serwletu

Przedstawiona poniżej klasa ma na celu przetestowanie naszej implementacji interfejsu `ServletContextListener`. Jeśli wszystko pójdzie zgodnie z planem, już w momencie pierwszego wywołania metody `doGet()` niniejszego serwletu obiekt klasy `Pies` będzie miał postać gotowego do odczytania atrybutu obiektu `ServletContext`.

```
package com.example;
```

```
import javax.servlet.*;
```

```
import javax.servlet.http.*;
```

```
import java.io.*;
```

Na razie nie ma w tym kodzie nic szczególnego... tylko zwykły serwlet.

```
public class ListenerTester extends HttpServlet {
```

```
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException {
```

```
        response.setContentType("text/html");
```

```
        PrintWriter out = response.getWriter();
```

```
        out.println("test atrybutów kontekstu ustawianych przez obiekt nasłuchujący<br>");
```

```
        out.println("<br>");
```

```
        Pies pies = (Pies) getServletContext().getAttribute("pies");
```

← Nie zapomnij o rzutowaniu!!

```
        out.println("Rasa psa: " + pies.getRasa());
```

```
    }
```

```
}
```

W tym miejscu odczytujemy obiekt `Pies` z obiektu `ServletContext`. Jeśli obiekt nasłuchujący zadziałał prawidłowo, atrybut `pies` został umieszczony w obiekcie `ServletContext` PRZED pierwszym wywołaniem metody `doGet` tego serwletu.

Gdyby coś poszło nie tak, w TYM miejscu się o tym dowiemy... otrzymamy wielki, tłusty komunikat `NullPointerException` w odpowiedzi na próbę wywołania metody `getRasa()` nieistniejącego obiektu `Pies`.

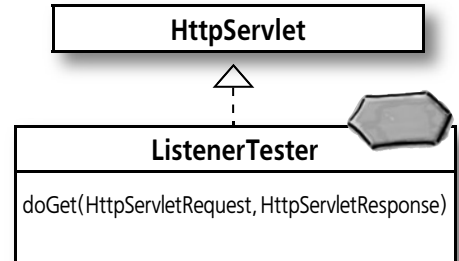


Oglądaj to!

Metoda `getAttribute()` zwraca obiekt typu `Object`! Musimy rzutować zwrócony obiekt!.

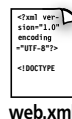
Warto jednak pamiętać, że metoda `getInitParameter()` zwraca łańcuch. Oznacza to, że zawsze musimy rzutować obiekty zwracane przez metodę `getAttribute()`, ale już obiekty zwracane przez metodę `getInitParameter()` możemy przypisywać bezpośrednio do zmiennych łańcuchowych. Nie daj się zmylić podstępnyim twórcom przykładów egzaminacyjnych, którzy celowo pomijają rzutowanie:

```
Pies p = ctx.getAttribute("pies"); ← Żle!!  
(Przyjmujemy, że ctx jest obiektem klasy ServletContext.)
```



Pisanie deskryptora wdrożenia

Musimy teraz przekazać kontenerowi, że istnieje klasa nasłuchująca dla naszej aplikacji internetowej — wykorzystamy do tego celu element `<listener>`. Struktura tego elementu jest bardzo prosta, ponieważ wymaga jedynie nazwy klasy. To wszystko.



To jest plik `web.xml` przechowywany w katalogu `WEB-INF` właściwym dla danej aplikacji internetowej.

```
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
  version="2.4">
```

```
<servlet>
  <servlet-name>ListenerTester</servlet-name>
  <servlet-class>com.example.ListenerTester</servlet-class>
</servlet>
```

```
<servlet-mapping>
  <servlet-name>ListenerTester</servlet-name>
  <url-pattern>/ListenTest.do</url-pattern>
</servlet-mapping>
```

```
<context-param>
  <param-name>rasa</param-name>
  <param-value>Dog niemiecki</param-value>
</context-param>
```

W naszej aplikacji niezbędny jest parametr inicjalizacji kontekstu. Obiekt nasłuchujący będzie potrzebował tego parametru podczas konstruowania obiektu klasy `Pies`.

```
<listener>
  <listener-class>
    com.example.MojServletContextListener
  </listener-class>
</listener>
```

Rejestrujemy tę klasę w roli klasy nasłuchującej. UWAGA: element `<listener>` NIE może się znajdować wewnątrz elementu `<servlet>`. Takie rozwiązanie nie byłoby właściwe, ponieważ obiekt nasłuchujący kontekst obejmuje swoim działaniem zdarzenia dotyczące obiektu `ServletContext` (a więc kontekstu całej aplikacji internetowej). Naszym zasadniczym celem jest zainicjalizowanie aplikacji PRZED inicjalizacją któregośkolwiek z serwetów.

```
</web-app>
```

Nie ma niemądrych pytań

P: Zaczekaj chwilę... w jaki sposób przekazujemy kontenerowi, że dany obiekt nasłuchujący ma reagować na zdarzenia związane z obiektem `ServletContext`? Nigdzie nie użyto elementu XML definiującego typ obiektu nasłuchującego lub typ zdarzeń nasłuchiowanych przez ten obiekt. Zauważyłem jednak, że użyto wyrażenia `ServletContextListener` jako części nazwy klasy — czy samo użycie nazwy interfejsu decyduje o wiedzy kontenera w tym zakresie? Czy jest to efekt stosowania jakiejś konwencji nazewnictwa?

O: Nie. To nie ma związku z żadną konwencją nazewnictwa. Użyliśmy nazwy interfejsu tylko po to, by położyć szczególny nacisk na rodzaj napisanej przez nas klasy. Kontener określa charakter klasy nasłuchującej wyłącznie w oparciu o samą klasę i implementowany przez nią interfejs nasłuchujący (lub interfejsy nasłuchujące; klasa nasłuchująca może przecież implementować więcej niż jeden taki interfejs).

P: Czy to oznacza, że w interfejsie API serwletów istnieją też inne typy obiektów nasłuchujących?

O: Tak, istnieje wiele innych typów interfejsów nasłuchujących, którymi za chwilę się zajmiemy.

Kompilacja i wdrożenie

Spróbujmy teraz to wszystko uruchomić.
Musimy w tym celu wykonać następujące kroki:

❶ Kompilacja trzech klas.

Wszystkie te klasy należą do tego samego pakietu...

❷ Utworzenie nowej aplikacji internetowej w kontenerze Tomcat:

- Utworzenie katalogu nazwanego *listenerTest* i umieszczenie go wewnątrz katalogu *webapps* kontenera Tomcat.
- Utworzenie katalogu nazwanego *WEB-INF* i umieszczenie go wewnątrz katalogu *listenerTest*.
- Umieszczenie pliku *web.xml* w nowoutworzonym katalogu *WEB-INF*.
- Utworzenie podkatalogu *classes* wewnątrz katalogu *WEB-INF*.
- Utworzenie wewnątrz katalogu *classes* całej struktury katalogów odpowiadającej naszej strukturze pakietów (w tym przypadku chodzi o katalog *com* z podkatalogiem *example*).

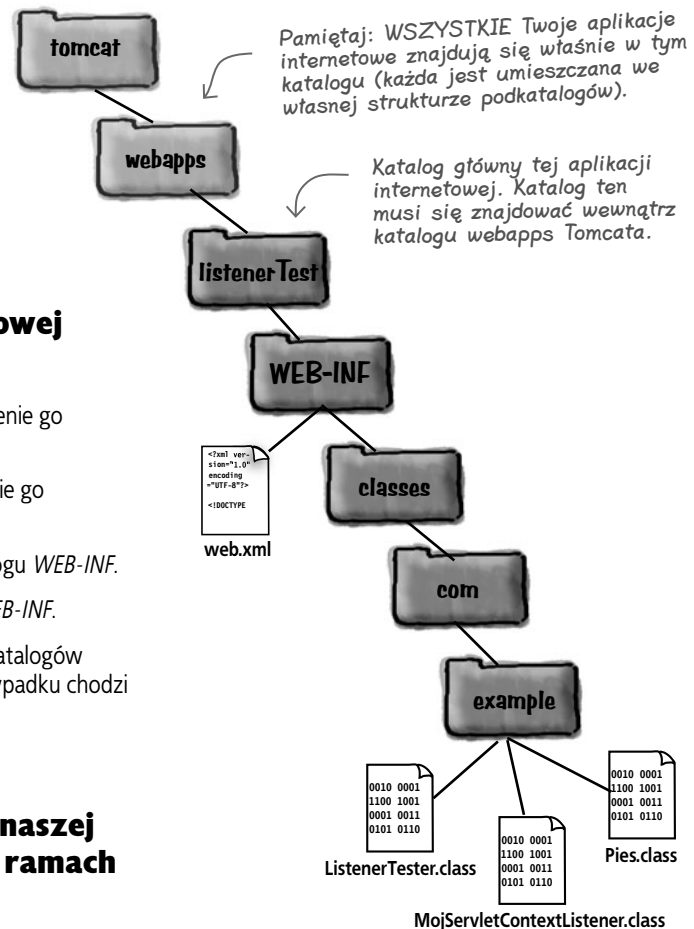
❸ Skopiowanie trzech skompilowanych plików klas do struktury katalogów naszej aplikacji internetowej utworzonej w ramach struktury katalogowej Tomcata:

```
listenerTest/WEB-INF/classes/com/example/Pies.class  
listenerTest/WEB-INF/classes/com/example/ListenerTester.class  
listenerTest/WEB-INF/classes/com/example/MyServletContextListener.class
```

❹ Umieszczenie pliku deskryptora wdrożenia (web.xml) w katalogu WEB-INF naszej aplikacji internetowej:

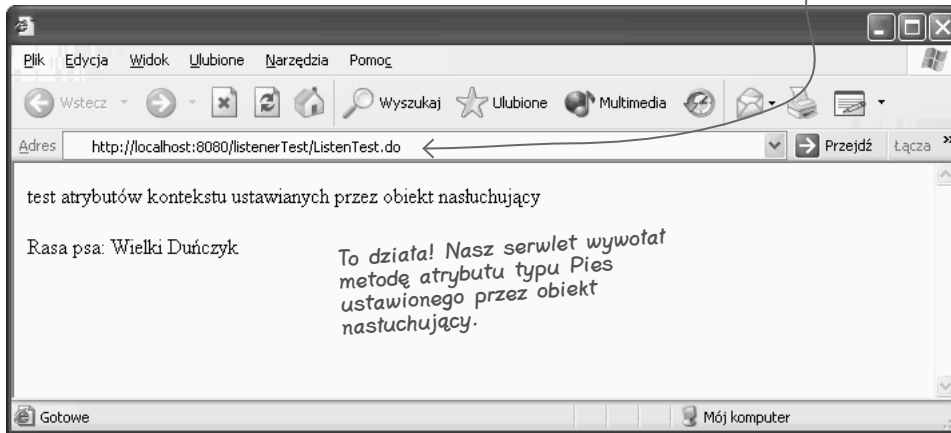
```
listenerTest/WEB-INF/web.xml
```

❺ Wdrożenie aplikacji przez wstrzymanie i ponowne uruchomienie całego kontenera Tomcat.



Wypróbuj aplikację

Uruchom swoją przeglądarkę i wpisz adres wskazujący bezpośrednio na utworzony przed chwilą serwlet. Nie komplikowaliśmy sobie zadania przez dodatkowe tworzenie odpowiedniej strony HTML, zatem będziemy uzyskiwali dostęp do serwletu przez wpisywanie adresu URL odwzorowywanego na serwlet dzięki właściwym elementom deskryptora wdrożenia (`ListenTest.do`).



Rozwiązywanie problemów

Jeśli otrzymasz komunikat o wyjątku `NullPointerException`, będzie to oznaczało, że nie udało Ci się pobrać obiektu klasy `Pies` za pośrednictwem metody `getAttribute()`. W pierwszej kolejności powinieneś wówczas sprawdzić nazwę użytą w wywołaniu metody `setAttribute()` i upewnić się, że zastosowany tam łańcuch odpowiada łańcuchowi użytemu w późniejszym wywołaniu metody `getAttribute()`.

Ponownie sprawdź zawartość pliku `web.xml` i upewnij się, że element `<listener>` został zarejestrowany.

Spróbuj przejrzeć dzienniki serwera i sprawdzić, czy nie ma tam informacji, które umożliwiają określenie, czy obiekt nasłuchujący faktycznie został wywołany.

Aby wprowadzić do tego przykładu jak najwięcej zamieszania i niejasności, celowo nadaliśmy wszystkim składnikom naszej aplikacji bardzo podobne nazwy. Chcieliśmy się upewnić, że przywiązujesz odpowiednio dużą wagę do stosowania tych nazw w praktyce — kiedy przez pomyłkę nadasz dwóm składnikom takie same nazwy, będziesz miał ogromne trudności ze zlokalizowaniem ewentualnych błędów.

Nazwa klasy **serwletu**: **`ListenerTester.class`**

Nazwa katalogu **aplikacji internetowej**: **`listenerTest`**

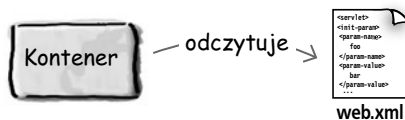
Wzorzec URL odwzorowywany na nasz serwlet: **`ListenTest.do`**

Zachowaj szczególną ostrożność podczas stosowania takich słów jak `Listener` i `Listen` oraz `Tester` i `Test`.

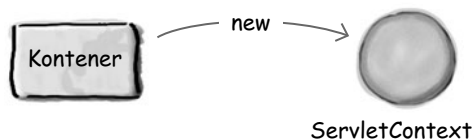
Cała historia...

Poniżej przedstawiono kompletny scenariusz wykorzystania obiektu nasłuchującego od samego początku (inicjalizacji aplikacji) do końca (uruchomienia serwletu). Podczas analizy czynności z kroku 11. zobaczysz, jak udało nam się cały proces inicjalizacji serwletu ująć w jednym zdaniu i na nieskomplikowanym schemacie.

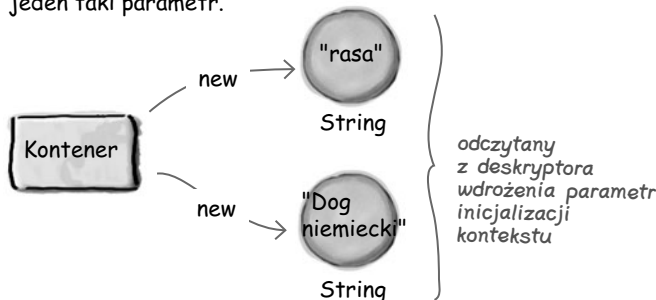
- ❶ Kontener odczytuje deskryptor wdrożenia danej aplikacji internetowej (włącznie z elementami `<listener>` i `<context-param>`).



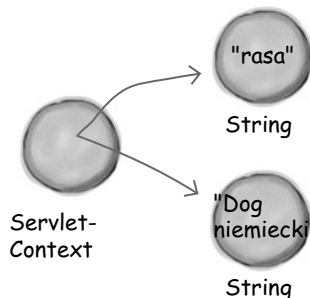
- ❷ Kontener tworzy dla tej aplikacji nowy obiekt `ServletContext`, który będzie współużytkowany przez wszystkie jej składniki.



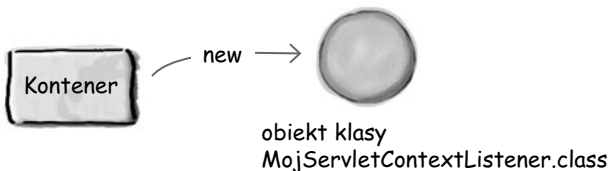
- ❸ Kontener tworzy parę łańcuchów nazwa-wartość dla każdego znalezionego w deskrytorze wdrożenia parametru inicjalizacji kontekstu. Przyjmijmy, że w tym przypadku istnieje tylko jeden taki parametr.



- ❹ Kontener przekazuje do obiektu `ServletContext` referencje do parametrów w postaci nazwa-wartość.



- ❺ Kontener tworzy nowy obiekt klasy `MojServletContextListener`.

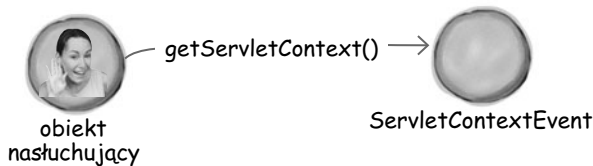


- ❻ Kontener wywołuje metodę `contextInitialized()` obiektu nasłuchującego, przekazując na jej wejściu nowy obiekt `ServletContextEvent`. Obiekt zdarzenia zawiera referencję do odpowiedniego obiektu kontekstu (`ServletContext`), zatem kod obsługujący to zdarzenie może uzyskać dostęp zarówno do samego kontekstu, jak i do parametrów inicjalizacji kontekstu.

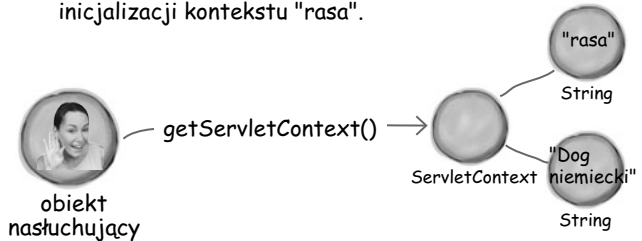


Ciąg dalszy historii...

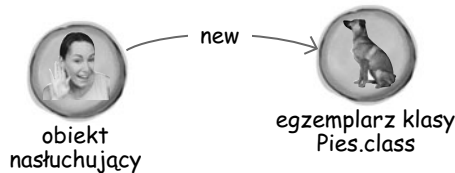
- 7 Obiekt nasłuchujący prosi otrzymany obiekt ServletContextEvent o referencję do obiektu ServletContext.



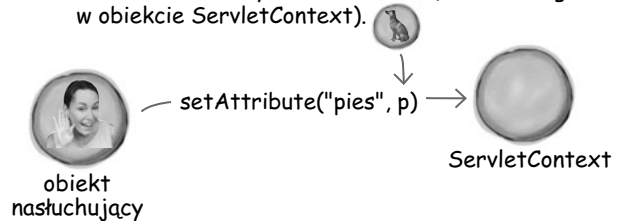
- 8 Obiekt nasłuchujący prosi uzyskany przed chwilą obiekt ServletContext o wartość parametru inicjalizacji kontekstu "rasa".



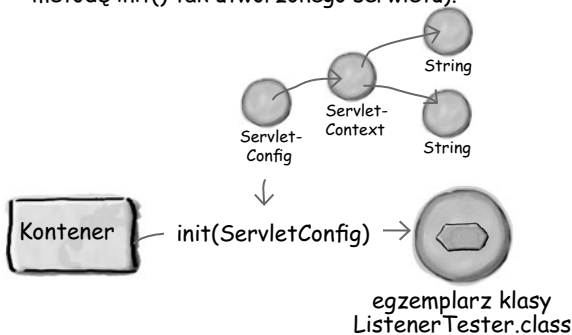
- 9 Obiekt nasłuchujący wykorzystuje parametr inicjalizacji do skonstruowania nowego obiektu klasy Pies.



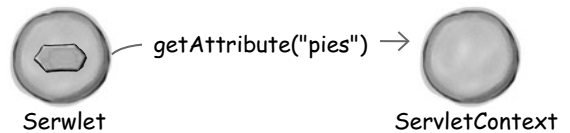
- 10 Obiekt nasłuchujący zapisuje nowy obiekt klasy Pies w formie atrybutu kontekstu (składowanego w obiekcie ServletContext).



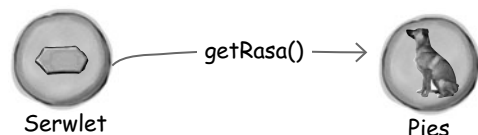
- 11 Kontener tworzy nowy serwlet (tj. tworzy nowy obiekt ServletConfig z parametrami inicjalizacji, przekazuje do tego egzemplarza referencję do istniejącego obiektu ServletContext i wywołuje metodę init() tak utworzonego serwletu).



- 12 Serwlet otrzymuje żądanie i prosi obiekt ServletContext o udostępnienie wartości atrybutu "pies".



- 13 Serwlet wywołuje metodę getRasa() obiektu klasy Pies (i umieszcza otrzymany wynik w obiekcie odpowiedzi HttpServletResponse).



Tak sobie myślę... skoro atrybuty można ustawiać z poziomu aplikacji (w przeciwieństwie do parametrów inicjalizacji), to czy nie można nasłuchiwać zdarzeń związanych z samymi atrybutami? Czy mógłbym na przykład wykrywać, że ktoś dodał lub zastąpił atrybut pies?



Interfejsy nasłuchujące — nie tylko zdarzenia kontekstu...

Tam, gdzie istnieje *moment cyklu życia*, tam występuje zwykle interfejs nasłuchujący, który ten moment wychwytuje. Poza zdarzeniami kontekstu możemy nasłuchiwać także zdarzeń związanych z *atrybutami* kontekstu, żądaniami i atrybutami serwletu oraz sesjami HTTP i atrybutami tych sesji.



Relax Nie musisz znać całego interfejsu API mechanizmów nasłuchujących.

Inaczej niż w przypadku najważniejszych metod interfejsu `ServletContextListener`, metod pozostałych interfejsów nasłuchujących naprawdę nie musisz zapamiętywać. **MUSISZ** jednak wiedzieć, jakiego typu zdarzenia można nasłuchiwać.

Cele egzaminu są jasne: otrzymasz scenariusz (stawiane przed programistą zadanie, które ma być realizowane przez jego aplikację) i będziesz musiał zdecydować, który typ interfejsu nasłuchującego będzie najwłaściwszy, lub określić, czy reagowanie na określone zdarzenia cyklu życia w ogóle jest **MOŻLIWE**.

Uwaga: zagadnieniom związanym z sesjami poświęcimy kolejny rozdział, zatem nie przejmuj się, jeśli stwierdzisz, że brakuje Ci wiedzy o tym, czym jest sesja HTTP lub dlaczego jest dla nas taka istotna...



Ćwiczenie

Wybierz interfejs nasłuchujący

Przyporządkuj właściwy interfejs nasłuchujący (dostępne interfejsy znajdują się na dole strony) do scenariusza po lewej stronie. Żaden interfejs nie może być użyty więcej niż raz. (Tak, WIEMY, że nie omawialiśmy jeszcze zagadnień z tym związanych. Spróbuj jednak wykonać to ćwiczenie najlepiej, jak potrafisz, choćby interpretując nazwy interfejsów. Odpowiedzi zamieszczono na następnej stronie; staraj się oprzeć pokusie zerknięcia tam przed ukończeniem ćwiczenia!)



Scenariusz

Chcemy wiedzieć, czy w obiekcie kontekstu aplikacji internetowej dodano, usunięto lub zastąpiono jakiś atrybut.

Chcemy wiedzieć, ilu użytkowników jest w danej chwili obsługiwanych. Innymi słowy, chcemy śledzić aktywne sesje.

Chcemy być informowani o każdym otrzymywanym żądaniu, aby rejestrować aktywność aplikacji.

Chcemy wiedzieć, kiedy jest dodawany, usuwany lub zastępowany jakiś atrybut żądania.

Używamy klasy atrybutu (klasy dla obiektu, który zostanie umieszczony w atrybucie) i chcemy, aby obiekty tego typu były informowane o przypisywaniu lub usuwaniu z sesji.

Chcemy wiedzieć, kiedy jest dodawany, usuwany lub zastępowany jakiś atrybut sesji.

Interfejs nasłuchujący

Wybierz poniższe interfejsy nasłuchujące.
Każdego interfejsu powinienś użyć dokładnie raz.

- HttpSessionAttributeListener

ServletRequestListener
- HttpSessionBindingListener

HttpSessionListener
- ServletContextAttributeListener

ServletRequestAttributeListener

Osiem interfejsów nasłuchujących

Scenariusz	Interfejs nasłuchujący	Typ zdarzenia
Chcemy wiedzieć, czy w obiekcie kontekstu aplikacji internetowej dodano, usunięto lub zastąpiono jakiś atrybut.	<code>javax.servlet.ServletContextAttributeListener</code> <i>attributeAdded</i> <i>attributeRemoved</i> <i>attributeReplaced</i>	<code>ServletContextAttributeEvent</code>
Chcemy wiedzieć, ilu użytkowników jest w danej chwili obsługiwanych. Innymi słowy, chcemy śledzić aktywne sesje. (Zagadnienia związane z sesjami szczegółowo omówimy w następnym rozdziale.)	<code>javax.servlet.http.HttpSessionListener</code> <i>sessionCreated</i> <i>sessionDestroyed</i>	<code>HttpSessionEvent</code>
Chcemy być informowani o każdym otrzymywanym żądaniu, aby rejestrować aktywność aplikacji.	<code>javax.servlet.ServletRequestListener</code> <i>requestInitialized</i> <i>requestDestroyed</i>	<code>ServletRequestEvent</code>
Chcemy wiedzieć, kiedy jest dodawany, usuwany lub zastępowany jakiś atrybut żądania.	<code>javax.servlet.ServletRequestAttributeListener</code> <i>attributeAdded</i> <i>attributeRemoved</i> <i>attributeReplaced</i>	<code>ServletRequestAttributeEvent</code>
Używamy klasy atrybutu (klasy dla obiektu, który zostanie umieszczony w atrybucie) i chcemy, aby obiekty tego typu były informowane o przypisywaniu lub usuwaniu z sesji.	<code>javax.servlet.http.HttpSessionBindingListener</code> <i>valueBound</i> <i>valueUnbound</i>	<code>HttpSessionBindingEvent</code>
Chcemy wiedzieć, kiedy jest dodawany, usuwany lub zastępowany jakiś atrybut sesji.	<code>javax.servlet.HttpSessionAttributeListener</code> <i>attributeAdded</i> <i>attributeRemoved</i> <i>attributeReplaced</i>	<code>HttpSessionAttributeEvent</code> <small>Zwróć uwagę na pewną niekonsekwencję w schemacie nazewnictwa! Zdarzeniem nasłuchiwanym przez interfejs <code>HttpSessionAttributeListener</code> NIE jest (jak można by się spodziewać) <code>HttpSessionAttributeEvent</code>.</small>
Chcemy wiedzieć, czy został utworzony lub zniszczony jakiś obiekt kontekstu.	<code>javax.servlet.ServletContextListener</code> <i>contextInitialized</i> <i>contextDestroyed</i>	<code>ServletContextEvent</code>
Mamy klasę atrybutu i chcemy, aby obiekty tego typu były informowane o zdarzeniach przekazywania powiązanych z nimi sesji pomiędzy wirtualnymi maszynami Javy w środowisku rozproszonym.	<code>javax.servlet.http.HttpSessionActivationListener</code> <i>sessionDidActivate</i> <i>sessionWillPassivate</i>	<code>HttpSessionEvent</code> <small>NIE <code>HttpSessionActivationEvent</code>!</small>

Interfejs HttpSessionBindingListener

Być może nie jest dla Ciebie jasna różnica pomiędzy interfejsem `HttpSessionBindingListener` a interfejsem `HttpSessionAttributeListener` (No dobra, nie chodzi o Ciebie, ale kogoś z Twoich współpracowników).

`HttpSessionAttributeListener` jest tradycyjnym interfejsem nasłuchującym zdarzeń polegających na dodawaniu, usuwaniu lub zastępowaniu dowolnych rodzajów atrybutów w ramach sesji. Z drugiej strony, interfejs `HttpSessionBindingListener` zaprojektowano z myślą o wykrywaniu zdarzeń swojego dodawania lub usuwania z sesji przez *sam* atrybut.

```
package com.example;

import javax.servlet.http.*;

public class Pies implements HttpSessionBindingListener {
    private String rasa;

    public Pies(String rasa) {
        this.rasa = rasa;
    }

    public String getRasa() {
        return rasa;
    }

    public void valueBound(HttpSessionBindingEvent event) {
        // kod wykonywany w sytuacji, gdy wiem, że jestem w sesji
    }

    public void valueUnbound(HttpSessionBindingEvent event) {
        // kod wykonywany w sytuacji, gdy wiem, że nie jestem już częścią sesji
    }
}
```

Tym razem atrybut Pies jest DODATKOWO obiektem nasłuchującym... nasłuchuje zdarzeń polegających na dodaniu lub usunięciu danego obiektu z sesji. (Uwaga: obiekty nasłuchujące zdarzeń wiązania NIE są rejestrowane w deskrytorze wdrożenia... wszystko dzieje się automatycznie).

W nazwach metod słowa „bound” i „unbound” występują odpowiednio w znaczeniu „dodany” i „usunięty”.



P: Dobrze. Wiem już, jak to działa. Rozumiem, że Pies (atrybut, który zostanie dodany do sesji) chce wiedzieć, kiedy jest dodawany do sesji i kiedy jest z tej sesji usuwany. Nie rozumiem tylko, PO CO to wszystko?

U: Jeśli wiesz cokolwiek na temat komponentów encyjnych (ang. *Entity Bean*), być może potrafisz spojrzeć na ten model jak na swoisty „komponent biedaka”. Jeśli nie masz wystarczającej wiedzy o komponentach encyjnych, powinieneś pójść do najbliższej księgarni i kupić dwa egzemplarze książki *Head First EJB*¹ (jeden dla siebie, drugi dla osoby sobie najbliższej, z którą będziesz mógł spędzić niezapomniane chwile pograżony w dyskusji nad prezentowanymi tam zagadnieniami).

W międzyczasie możesz się zapoznać z nieco innym wyjaśnieniem tego rozwiązania — wyobraź sobie, że klasa `Pies` tak naprawdę jest

klasą Klient, a każdy jej obiekt reprezentuje informacje na temat pojedynczego klienta (włącznie z nazwiskiem, adresem, danymi zamówień etc.). Właściwe dane są przechowywane w działającej w tle bazie danych. Wykorzystujemy informacje zapisane w bazie danych do wypełniania pól obiektu Klient, problemem jest więc to, *jak i kiedy należy synchronizować informacje przechowywane w obiekcie Klient z odpowiednim rekordem bazy danych*. Wiesz, że kiedy obiekt Klient jest dodawany do sesji, należy odświeżyć pola tego obiektu i zaktualizować je w oparciu o zawartość odpowiedniego rekordu w bazie danych. Znaczenie metody `valueBound()` można więc wyjaśnić następująco: „Wczytaj mnie ze świeżymi informacjami z bazy danych... tak na wszelki wypadek, gdyby dane te uległy zmianie od ostatniego razu, kiedy byłem wykorzystywany”. W takim przypadku można przyjąć, że metoda `valueUnbound()` mówi: „Zaktualizuj odpowiedni rekord bazy danych z wykorzystaniem wartości pól obiektu Klient”.

¹ Polskie wydanie: *Head First EJB. Edycja polska*, Helion, 2005 — przyp. tłum.



Zapamiętywanie interfejsów nasłuchujących

Postaraj się wypełnić puste pola poniższej tabeli najlepiej jak potrafisz. Pamiętaj, że interfejsy nasłuchujące i odpowiednie metody (najczęściej) są nazywane zgodnie z przejrzystymi schematami nazewnictwa. Odpowiedzi znajdziesz na końcu tego rozdziału.

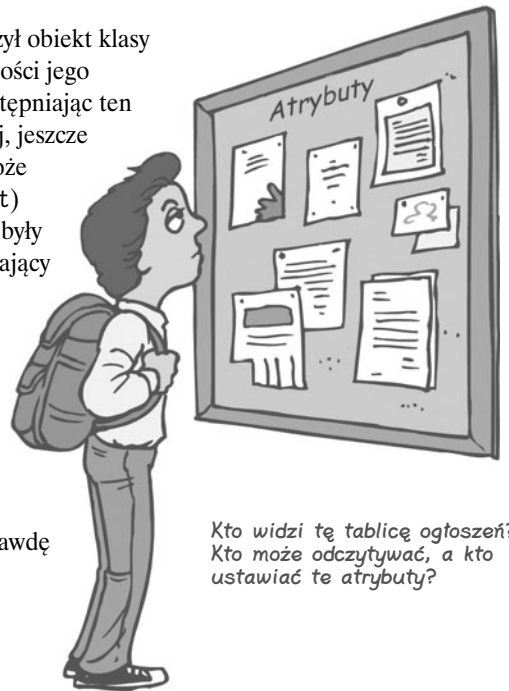


Interfejsy nasłuchujące dla atrybutów	
Interfejsy nasłuchujące pozostałych zdarzeń cyklu życia	
Metody we wszystkich interfejsach nasłuchujących dla atrybutów (z wyjątkiem tych nasłuchujących zdarzeń wiązania wartości)	
Zdarzenia cyklu życia związane z sesjami (z wyłączeniem zdarzeń związanych z atrybutami)	
Zdarzenia cyklu życia związane z żądaniami (z wyłączeniem zdarzeń związanych z atrybutami)	
Zdarzenia cyklu życia związane z kontekstem serwletu (z wyłączeniem zdarzeń związanych z atrybutami)	

Czym dokładnie jest atrybut?

Widzieliśmy już, jak obiekt nasłuchujący zdarzeń kontekstu serwletu tworzył obiekt klasy `Pies` (po uzyskaniu parametru inicjalizacji kontekstu) i korzystał z możliwości jego ustawiania w postaci atrybutu obiektu `ServletContext` (tym samym udostępniając ten obiekt pozostałym składnikom tej samej aplikacji internetowej). Wcześniej, jeszcze podczas omawiania aplikacji dla piwoszy, przekonaliśmy się, że serwlet może zapisywać w obiekcie żądania (zwykle obiekcie klasy `HttpServletRequest`) wyniki uzyskane przez wywołanie modelu — także w tym przypadku dane były zapisywanie w postaci odpowiedniego atrybutu (a więc w sposób umożliwiający odczytywanie tych wartości przez strony JSP i inne elementy widoku).

Atrybut jest obiektem ustawionym (*związanym*) wewnątrz jednego z trzech innych obiektów interfejsu API serwletu — `ServletContext`, `HttpServletRequest` (lub `ServletRequest`) bądź `HttpSession`. Atrybut można traktować jak prostą parę w postaci nazwa-wartość (gdzie nazwa jest łańcuchem typu `String`, a wartość jest obiektem typu `Object`) w klasowej zmiennej odwzorowania. W praktyce nie wiemy i nie interesuje nas, jak tego typu mechanizmy są implementowane — tak naprawdę zasadnicze znaczenie ma dla nas *zasięg* dostępności tych atrybutów. Innymi słowy, musimy wiedzieć, *kto* będzie widział atrybut i *jak długo* ten atrybut będzie istniał.



Atrybut jest jak kartka
przypięta do tablicy ogłoszeń.
Ktoś umieszcza ją na tablicy,
aby inni mogli się z nią
zapoznać.

Zasadnicze pytanie brzmi:
kto ma dostęp do tej tablicy
informacyjnej i jak długo
informacja będzie na tej tablicy
dostępna? Innymi słowy, jaki
jest zasięg tego atrybutu?

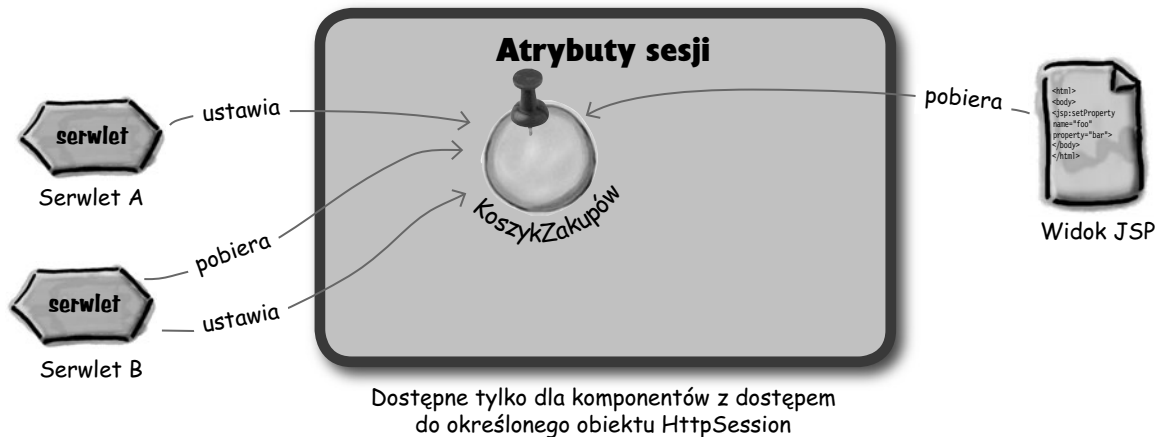
Atrybuty to nie parametry

Jeśli nie miałeś wcześniej do czynienia z serwletami, być może będziesz musiał poświęcić trochę czasu na analizę różnic dzielących *atrybuty* od *parametrów*. Pozostali mogą być pewni, że podczas tworzenia egzaminu celowo staraliśmy gmatwać pytania dotyczące atrybutów i parametrów w sposób gwarantujący jak najwięcej pomyłek ze strony zdających.*

	Atrybuty	Parametry
Typy	<div>Aplikacja (kontekst)</div> <div>Żądanie</div> <div>Sesja</div> <div>Nie ma atrybutu specyficznego dla serwletu (należy w tej roli wykorzystywać zwykłe zmienne egzemplarzy).</div>	<div>Parametry inicjalizacji aplikacji (kontekstu)</div> <div>Parametry żądania</div> <div>Parametry inicjalizacji <u>serwletu</u></div> <div>Nie istnieje coś takiego jak parametry sesji!</div>
Metoda ustawiająca	setAttribute(String name, Object value)	<div>NIE możesz ustawiać ani parametrów inicjalizacji aplikacji, ani parametrów inicjalizacji serwletu — tego typu parametry są ustawiane w deskrytorze wdrożenia, pamiętasz?</div> <div>(W przypadku parametrów żądania można co prawda dostosowywać łańcuch zapytania, ale to zupełnie inna historia.)</div>
Zwracany typ	Object	String ← To duża różnica!
Metoda zwracająca	<div>getAttribute(String name)</div> <div>Nie zapominaj, że atrybuty muszą być rzutowane, ponieważ zwracanym typem jest Object.</div>	getInitParameter(String name)

* Tak, to prawda. Gdyby przygotowany przez nas egzamin był prosty, łatwy i przyjemny, jego zdanie z dobrym wynikiem nie byłoby żadnym osiągnięciem ani tym bardziej powodem do dumy. Pamiętaj jednak, że nigdy, PRZENIGDY naszym celem nie było stworzenie egzaminu, który wymagałby od Ciebie dokupywania innych podręczników. *Poważnie*, przygotowując pytania egzaminacyjne, dokładnie przeanalizowaliśmy *Twoje* możliwości.

Trzy rodzaje zasięgu: kontekstu, żądania i sesji





Ćwiczenie

Zasięg atrybutów

Postaraj się wypełnić puste pola poniższej tabeli najlepiej, jak potrafisz. Zanim przystąpisz do egzaminu (i opracowywania rzeczywistych rozwiązań), MUSISZ dobrze rozumieć zagadnienia związane z zasięgiem atrybutów — znajomość tej problematyki jest niezbędna do podejmowania właściwych decyzji w konkretnych scenariuszach. Odpowiedzi dla niniejszego ćwiczenia znajdziesz kilka stron dalej, staraj się jednak tam nie zaglądać przed wykonaniem ćwiczenia! Jeśli planujesz przystąpić do egzaminu, zaufaj nam... będziesz się musiał bardzo poważnie zastanowić, zanim samodzielnie i właściwie wypełnisz niniejszą tabelę.

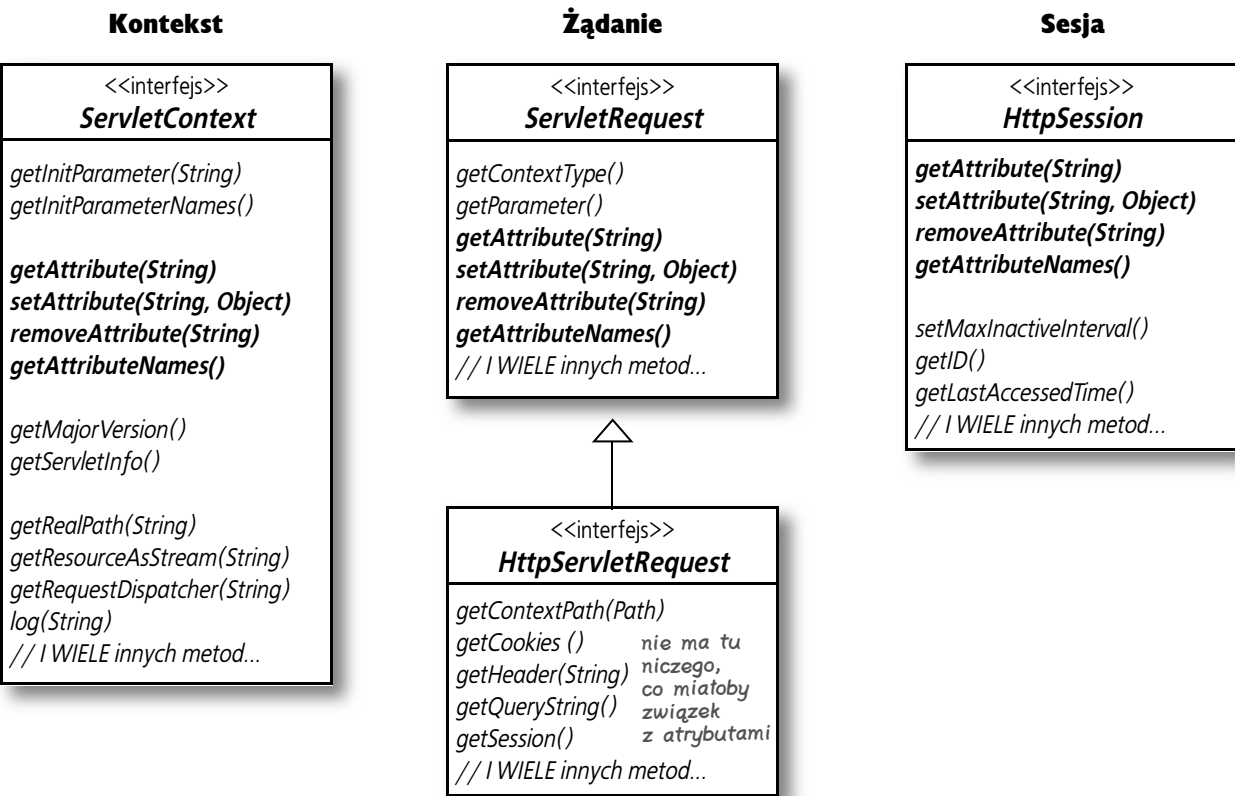
	Dostępność (kto widzi atrybut)	Zasięg (jak długo ten atrybut żyje)	Do czego służy?
Kontekst			
Sesja HTTP			
Żądanie			

(Uwaga: analizując zasięg atrybutów, powinieneś mieć na uwadze jego wpływ na funkcjonowanie mechanizmu zwalniania pamięci... Niektóre atrybuty nie zostaną usunięte z pamięci aż do usunięcia lub przerwania pracy całej aplikacji. Egzamin nie obejmuje co prawda żadnych zagadnień związanych z zarządzaniem pamięcią, ale warto mieć tego typu problemy na uwadze podczas projektowania aplikacji).

Interfejs API dla atrybutów

Wspominane już trzy zasięgi atrybutów — kontekstu, żądania i sesji — są obsługiwane odpowiednio przez interfejsy `ServletContext`, `ServletRequest` i `HttpSession`. Okazuje się, że w każdym z tych interfejsów istnieją dokładnie takie same metody obsługujące atrybuty.

- Object** `getAttribute(String name)`
- void** `setAttribute(String name, Object value)`
- void** `removeAttribute(String name)`
- Enumeration** `getAttributeNames()`



Ciemna strona atrybutów...

Kim zdecydował się przetestować nasze atrybuty. Ustawił atrybut w obiekcie kontekstu, by zaraz potem odczytać i zapisać jego wartość w obiekcie odpowiedzi. Metoda `doGet()` Kima wygląda następująco:

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException {

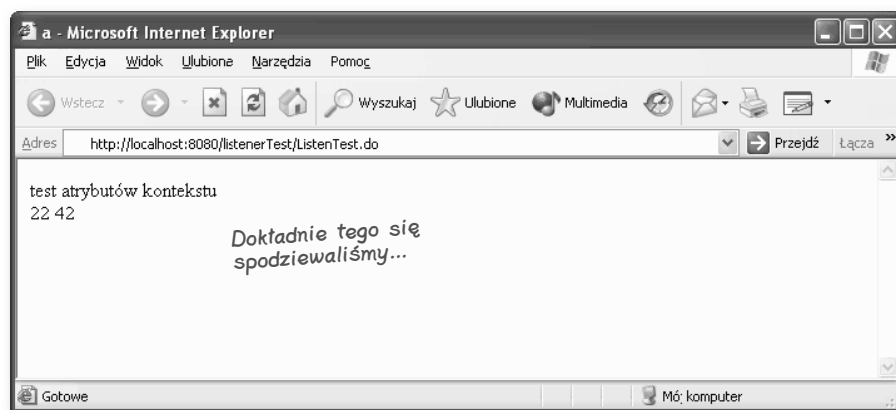
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    out.println("test atrybutów kontekstu<br>");

    getServletContext().setAttribute("foo", "22");
    getServletContext().setAttribute("bar", "42");

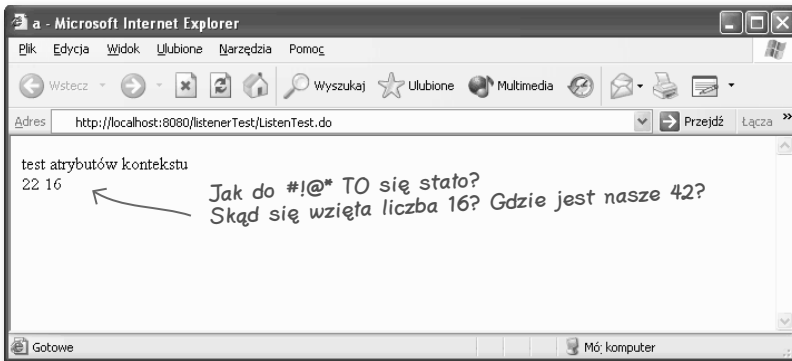
    out.println(getServletContext().getAttribute("foo"));
    out.println(getServletContext().getAttribute("bar"));
}
```

Oto, jak wygląda wynik odesłany do przeglądarki przez serwet Kima. Wyświetlony rezultat dokładnie odpowiada naszym oczekiwaniom.



Coś jednak poszło zupełnie nie tak...

Kiedy Kim drugi raz wywołał swój serwlet, ze zgrozą stwierdził, że otrzymał inny wynik:



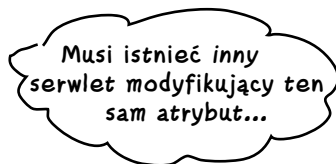
WYTEŻ UMYŚŁ

Przeanalizuj raz jeszcze kod metody `doGet()` i dokładnie przemyśl jej działanie. Czy już widzisz źródło problemu?

Być może nie dysponujesz jeszcze wystarczającą ilością informacji, aby rozwiązać tę zagadkę; być może przyda Ci się jeszcze jedna wskazówka: Kim umieścić swój kod w serwlecie testowym, który jest częścią większej aplikacji internetowej. Innymi słowy, serwlet zawierający przedstawioną metodę `doGet()` został wdrożony i działa jako część większej aplikacji.

Co to oznacza?

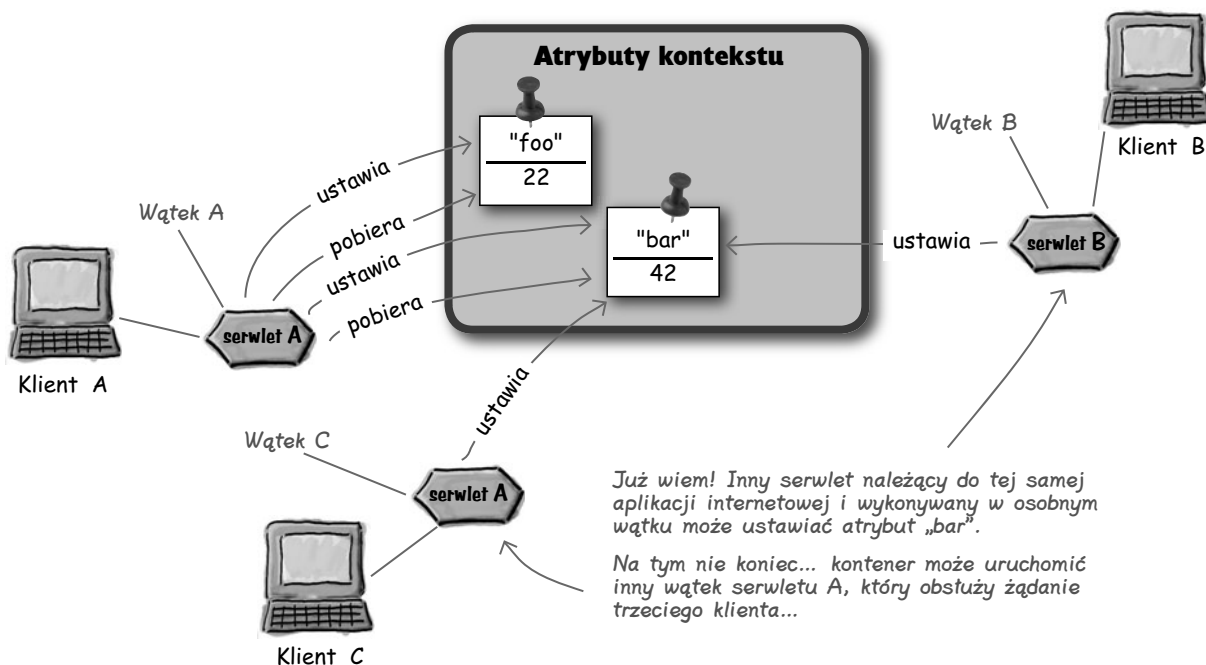
Masz jakiś pomysł, jak można ten problem rozwiązać?



Zasięg kontekstu nie zapewnia bezpieczeństwa wątków!

Na tym właśnie polega nasz problem.

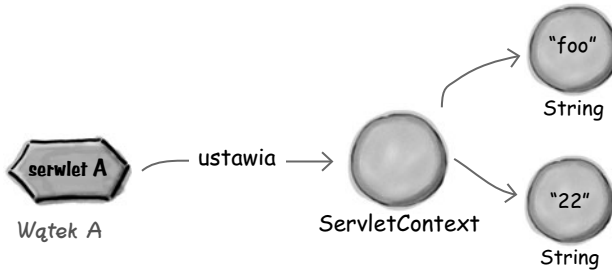
Pamiętaj, że wszystkie składniki danej aplikacji internetowej mają dostęp do atrybutów kontekstu, zatem ich wartości mogą być modyfikowane przez wiele innych serwletów. Istnienie **wielu serwletów oznacza, że możemy mieć do czynienia z jeszcze większą liczbą wątków**, ponieważ żądania klientów są obsługiwane współbieżnie (każde przez osobny wątek). Takie rozwiązanie jest stosowane niezależnie od tego, czy przychodzące żądania są kierowane do jednego, czy wielu różnych serwletów.



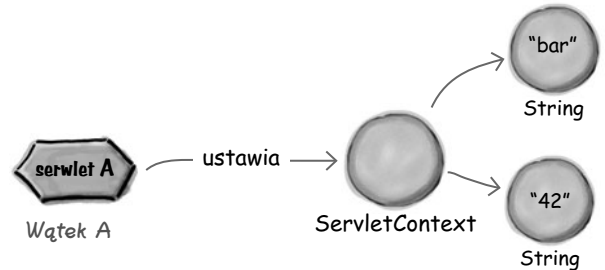
Omówienie problemu w zwolnionym tempie...

Poniżej omawiamy po kolei, co dzieje się z serwletem Kima.

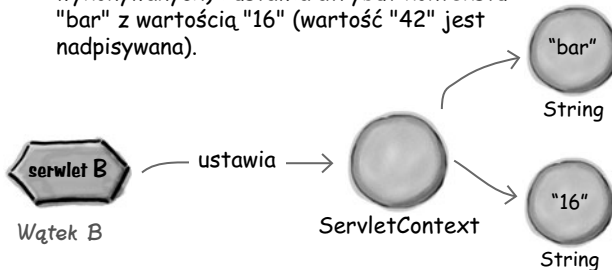
- ❶ Serwlet A ustawia atrybut kontekstu "foo" z wartością "22".



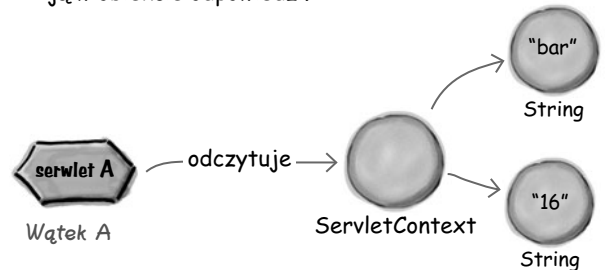
- ❷ Serwlet A ustawia atrybut kontekstu "bar" z wartością "42".



- ❸ Wątek B staje się aktywnym wątkiem (wątek A wraca do puli wątków wykonywalnych-ale-nie-wykonywanych) i ustawia atrybut kontekstu "bar" z wartością "16" (wartość "42" jest nadpisywana).



- ❹ Wątek A ponownie staje się aktywnym wątkiem, odczytuje wartość atrybutu "bar" i zapisuje ją w obiekcie odpowiedzi.



```
getServletContext().setAttribute("foo", "22");
getServletContext().setAttribute("bar", "42");
```

```
out.println(getServletContext().getAttribute("foo"));
out.println(getServletContext().getAttribute("bar"));
```

W czasie pomiędzy ustawieniem a odczytaniem wartości atrybutu „bar” przez serwlet A uruchomiony w tajemnicy wątek innego serwletu ustawia inną wartość w atrybucie „bar”.

Oznacza to, że w momencie, w którym serwlet A ponownie odczytuje i przekazuje do obiektu odpowiedzi wartość atrybutu „bar”, w rzeczywistości operuje na wartości zmienionej przez inny serwlet (w tym przypadku „16”).

Jak można zapewnić bezpieczeństwo wątków podczas przetwarzania atrybutów kontekstu?

Posłuchajmy teraz, co inni programiści mają do powiedzenia na ten temat...

Myślę, że mogłabym synchronizować metodę `doGet()`, ale nie jestem pewna, czy to rozwiązanie jest właściwe. Nic innego nie przychodzi mi jednak do głowy.

Synchronizacja wywołań metod `doGet()` w praktyce oznacza koniec współbieżności. Jeśli zsynchronizujesz metodę `doGet()`, Twój serwet będzie mógł jednocześnie obsługiwać tylko **JEDNEGO** klienta!

Dlaczego twórcy specyfikacji serwletu po prostu nie przewidzieli konieczności synchronizowania metod atrybutowych na poziomie obiektu `ServletContext`, aby zapewnić bezpieczeństwo wątków podczas przetwarzania atrybutów kontekstu?

Specyfikacja mówi, że za ochronę atrybutów odpowiada sam programista. Po co ktokolwiek miałby nas zmuszać do synchronizacji wątków, skoro nie zawsze jest to konieczne? Niektóre kontenery WWW oczywiście implementują synchronizację mimo braku takiego wymagania, ponieważ jednak nie możemy być pewni dostępności takiej implementacji, powinniśmy zachować ostrożność

Synchronizacja metody obsługującej żądania jest wyjątkowo ZŁYM pomysłem

No dobrze, wiemy już, że synchronizacja metody obsługującej żądania będzie końcem współbieżności, ale w ten sposób możemy przynajmniej chronić nasze wątki, prawda? Przyjrzyj się przedstawionemu poniżej zmodyfikowanemu fragmentowi kodu i zdecyduj, czy w ten sposób można rzeczywiście wyeliminować napotkany przez Kima problem modyfikowania atrybutu kontekstu przez inny serwet...

```
public synchronized void doGet(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException {

    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    out.println("test atrybutów kontekstu<br>");

    getServletContext().setAttribute("foo", "22");
    getServletContext().setAttribute("bar", "42");

    out.println(getServletContext().getAttribute("foo"));
    out.println(getServletContext().getAttribute("bar"));
}
```

To nie może działać!
Cóż, zmodyfikowany serwet jest
prawidłowy, ale nie mam pojęcia,
w jaki sposób mógłby rozwiązać
nasz problem...



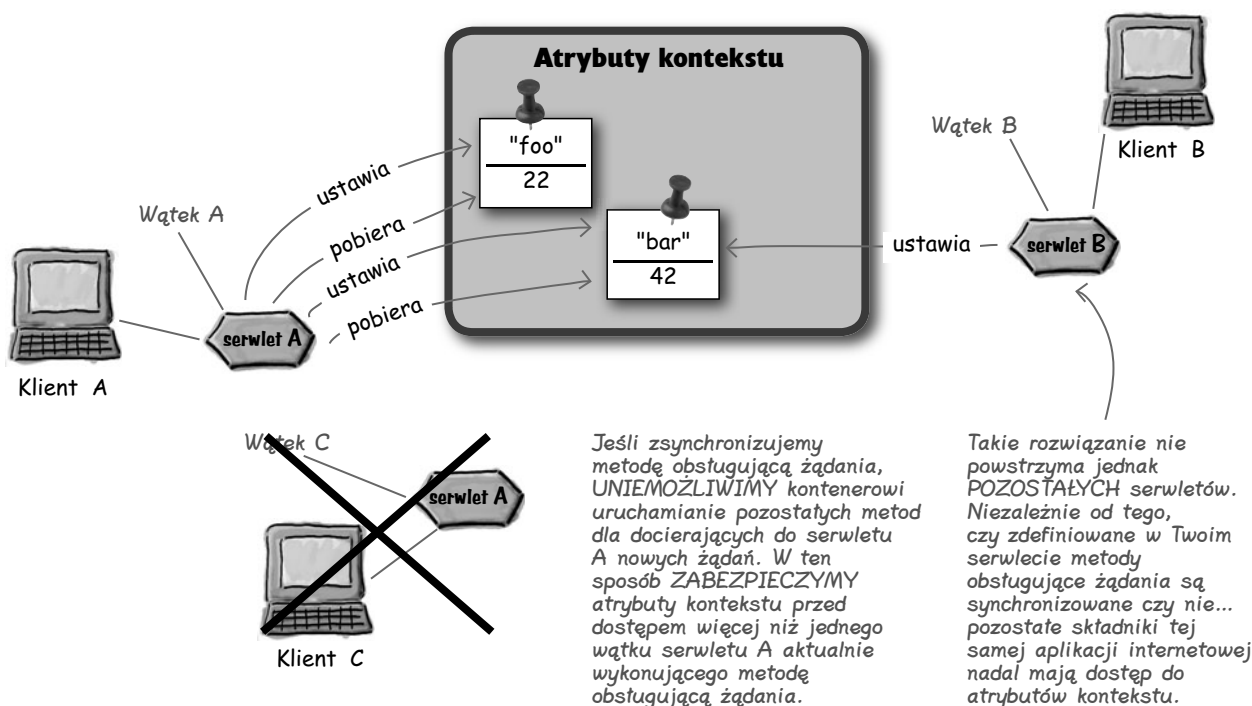
Jak myślisz? Czy w ten sposób można
rozwiązać problem Kima? Jeśli nie
jesteś pewien, rzuć okiem na powyższy
kod i na zaprezentowany wcześniej
diagram.

Nie synchronizuj metod obsługujących żądania

Synchronizacja metody obsługującej żądania nie ochroni atrybutu kontekstu!

Synchronizacja metody obsługującej żądania oznacza, że jednocześnie będzie można wykonywać tylko jeden wątek serwletu... ale wcale nie zapobiegnie modyfikacjom interesującego nas atrybutu przez *inne* serwlety lub strony JSP!

Synchronizacja metody obsługującej żądania uniemożliwi dostęp do atrybutów kontekstu wyłącznie pozostałym wątkom tego samego serwletu, ale nie będzie miała najmniejszego wpływu na możliwości, jakie mają w tym zakresie *inne* serwlety tej samej aplikacji internetowej.



Blokowanie atrybutów na poziomie serwletu jest bezcelowe... musimy zablokować atrybuty na poziomie kontekstu!

Typowym sposobem zabezpieczania atrybutów kontekstu jest synchronizacja DOSTĘPU do samego obiektu kontekstu. Gdyby każdy składnik danej aplikacji internetowej musiał przed uzyskaniem dostępu do atrybutów kontekstu założyć blokadę na obiekcie kontekstu, mielibyśmy gwarancję, że możliwość odczytywania i ustawiania atrybutów kontekstu będzie miał jednocześnie tylko jeden wątek. Nadal jednak używamy słowa *jeśli*. Takie rozwiązanie spełni swoją rolę, *jeśli wszystkie pozostałe fragmenty kodu operującego na atrybutach tego samego kontekstu TAKŻE będą synchronizowały dostęp do obiektu ServletContext*. Jeśli któryś z tych fragmentów nie zażąda blokady tego obiektu, będzie mógł bez żadnych ograniczeń przetwarzać atrybuty kontekstu. Jeśli jednak sam projektujesz taką aplikację internetową, możesz oczywiście zdecydować o stosowaniu we wszystkich składnikach aplikacji mechanizmu blokad przed uzyskiwaniem dostępu do atrybutów.



ServletContext

W przypadku atrybutów kontekstu synchronizacja metod na poziomie pojedynczych serwletów nie jest dobrym rozwiązaniem, ponieważ pozostałe składniki tej samej aplikacji internetowej nadal mają dostęp do kontekstu!

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException {
```

```
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
```

```
    out.println("test atrybutów kontekstu<br>");
```

```
    synchronized(getServletContext()) {
        getServletContext().setAttribute("foo", "22");
        getServletContext().setAttribute("bar", "42");

        out.println(getServletContext().getAttribute("foo"));
        out.println(getServletContext().getAttribute("bar"));
    }
}
```

Teraz zakładamy blokadę na samym obiekcie kontekstu! W ten sposób zabezpieczamy aktualny stan atrybutów kontekstu (nie należy stosować polecenia `synchronized(this)`).



Oglądaj to!

Możesz się spodziewać mnóstwa przykładów kodu źródłowego związanych z problematyką bezpieczeństwa wątków.

Na egzaminie będziesz miał do czynienia z wieloma fragmentami kodu reprezentującymi rozmaite strategie zapewniania bezpieczeństwa przetwarzania wielowątkowego podczas operacji na atrybutach. Będziesz musiał decydować, czy poszczególne rozwiązania sprawdzają się w praktyce przy określonych założeniach odnośnie celów aplikacji. To, że kod jest poprawny z punktu widzenia kompilatora (został skompilowany i uruchomiony bez zastrzeżeń), wcale nie musi oznaczać, że stanowi prawidłowe rozwiązanie problemu.

Ponieważ w tym przypadku zastosowaliśmy blokadę na poziomie całego kontekstu, zakładamy, że kiedy już uda nam się wejść do synchronizowanego bloku kodu, atrybuty kontekstu będą zabezpieczone przed ewentualnym dostępem innych wątków aż do momentu, w którym opuścimy ten blok kodu... lub coś w tym rodzaju. Bezpieczeństwo oznacza, że atrybuty są „zabezpieczone przed sytuacją, w której inny fragment kodu TAKŻE zsynchronizuje swój dostęp do obiektu ServletContext”.

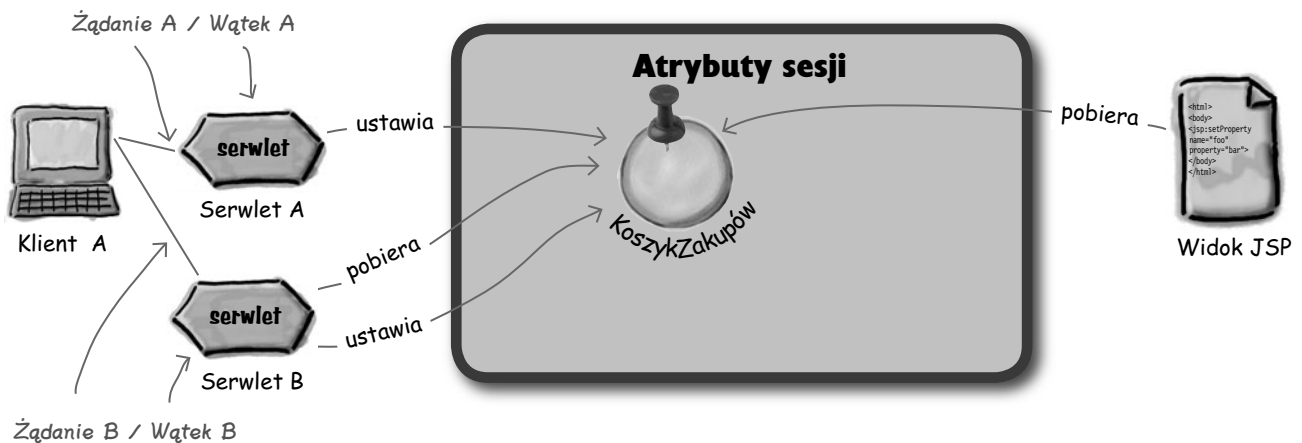
Przedstawione rozwiązanie jest najlepszym z istniejących sposobów zapewnienia bezpieczeństwa wątków podczas przetwarzania atrybutów kontekstu.

Czy atrybuty sesji gwarantują bezpieczeństwo przetwarzania wielowątkowego?

Przemyśl to.

Co prawda nie omawialiśmy jeszcze szczegółów związanych z sesjami protokołu HTTP (które będziemy analizowali w rozdziale poświęconym wyłącznie temu zagadnieniu), ale wiemy już, że sesja ma postać obiektu wykorzystywanego do utrzymywania stanu konwersacji z klientem. Sesja zwykle stanowi *część wspólną wielu żądań pochodzących od tego samego klienta*. Nadal jednak mówimy tylko o jednym kliencie aplikacji internetowej.

A skoro istnieje tylko jeden klient, który w dodatku nie może generować w tym samym czasie więcej niż jednego żądania, czyż nie oznacza to automatycznie, że sesje gwarantują bezpieczeństwo przetwarzania wielowątkowego? Innymi słowy, nawet jeśli aplikacja internetowa zawiera wiele serwletów, w dowolnym momencie istnieje tylko jedno żądanie od konkretnego klienta... zatem jednocześnie możemy mieć do czynienia tylko z jednym wątkiem operującym na danej sesji, prawda?



Oba serwlety mogą co prawda uzyskiwać dostęp do atrybutów sesji w osobnych wątkach, ale każdy wątek stanowi osobne żądanie. Można więc przyjąć, że taka konfiguracja jest całkowicie bezpieczna.

Chociaż...

Czy jesteś w stanie wymyślić scenariusz, w którym *może* istnieć więcej niż jedno żądanie realizowane w *tem samym czasie i pochodzące od tego samego klienta*?

Co o tym sądzisz? Czy w tym scenariuszu atrybuty sesji gwarantują bezpieczeństwo przetwarzania wielowątkowego?

Jakie są RZECZYWISTE zależności pomiędzy atrybutami a bezpieczeństwem wątków?



Posłuchaj, jak nasi karatecy dyskutują o zagadnieniach związanych z ochroną stanu atrybutów przed problemami przetwarzania wielowątkowego.

Wiemy, że atrybuty kontekstu z natury rzeczy NIE są bezpieczne, ponieważ wszystkie składniki tej samej aplikacji internetowej mogą uzyskiwać do nich dostęp (niezależnie od obsługiwanego żądania, a więc z poziomu dowolnego wątku).

Świetnie. A co z atrybutami **sesji**? Czy są bezpieczne?

Musisz się jeszcze *sporo* nauczyć, dzieciaku. Nie myśl, że znasz całą prawdę o atrybutach sesji. Zanim ponownie zabierzesz głos w tej sprawie, oddaj się medytacji.

Spróbuj przez chwilę abstrahować od kontenera. Oznacz kolorem zewnętrzne granice systemu i *użyj swoich nożyczek*.

Tak! Kontener może potraktować żądanie przychodzące z drugiego okna przeglądarki jak element tej samej sesji.

Jak wobec tego można zabezpieczyć atrybuty sesji przed zamieszaniami związanym z przetwarzaniem wielowątkowym?

Dobrze, to prawda, ale synchronizować dostęp *do czego*?

Tak, Mistrzu. Wiem też, że synchronizacja metody obsługującej żądania nie rozwiązuje problemu, ponieważ samo powstrzymanie tego samego serwletu przed jednoczesną obsługą więcej niż jednego żądania NIE zapobiegnie dostępowi *pozostałych* serwletów i stron JSP do obiektu kontekstu.

Tak, Mistrzu. Są bezpieczne, ponieważ są przypisane do *jednego* klienta, a prawa fizyki nie przewidują możliwości generowania przez pojedynczego klienta więcej niż jednego żądania w tym samym czasie.

Ale mistrzu, już nad tym medytowałem i wciąż nie wiem, w jaki sposób pojedynczy klient może w tym samym czasie generować więcej niż jedno żądanie...

To bardzo mądra rada, Mistrzu! **Przecież klient może stworzyć nowe okno przeglądarki!** Czy to oznacza, że kontener może dla danego klienta stosować tę samą sesję, mimo że otrzymywane żądania pochodzą z innego egzemplarza przeglądarki?

Zatem atrybuty sesji *nie* gwarantują bezpieczeństwa wątków i — podobnie jak atrybuty kontekstu — muszą być odpowiednio chronione. Cóż, jeszcze nad tym pomedytuję...

Już wiem... muszę synchronizować tę część mojego kodu, która uzyskuje dostęp do atrybutów sesji. Należy więc zastosować bardzo podobne rozwiązanie do tego, którego użyliśmy dla atrybutów kontekstu.

Muszę synchronizować dostęp do obiektu HttpSession!

Chroń atrybuty sesji przez synchronizowanie dostępu do obiektu HttpSession

Przyjrzyj się raz jeszcze technice, którą zastosowaliśmy do ochrony atrybutów kontekstu. Co wówczas zrobiliśmy?

Możemy ten sam zabieg w prosty sposób przenieść na grunt atrybutów sesji — wystarczy wprowadzić mechanizm synchronizacji dostępu do obiektu HttpSession!

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException {

    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    out.println("test atrybutów sesji<br>");
    HttpSession session = request.getSession();

    synchronized(session) { ← Tym razem synchronizujemy dostęp
        session.setAttribute("foo", "22");          do obiektu HttpSession, aby chronić
        session.setAttribute("bar", "42");          atrybuty sesji.

        out.println(session.getAttribute("foo"));
        out.println(session.getAttribute("bar"));
    }
}
```

Nie ma
niemądrych pytań

P: Czy to nie przesada? Czy to rzeczywiście możliwe, że klient otworzy drugie okno przeglądarki?

O: Oczywiście, że to możliwe. Z pewnością sam to już wielokrotnie robiłeś (choć być może nieświadomie) — otwierałeś drugie okno przeglądarki, ponieważ byłeś już zmęczony oczekiwaniem na odpowiedź z drugiej strony, minimalizowałeś jedno okno i bezmyślnie otwierałeś kolejne lub po prostu nie zauważyłeś, że dana aplikacja internetowa jest już otwarta w jednym z egzemplarzy przeglądarki. Rzecz w tym, że nigdy nie powinieneś mieć takiej możliwości w przypadku aplikacji, która musi gwarantować bezpieczeństwo wątków dla Twoich zmiennych sesyjnych. Musisz pamiętać, że w przypadku atrybutów o zasięgu sesyjnym dosyć często mamy do czynienia z sytuacją, w której są one jednocześnie wykorzystywane przez więcej niż jeden wątek.

228 Rozdział 5.

P: Czy synchronizowanie kodu rzeczywiście jest dobrym rozwiązaniem, skoro wymusza tak wiele dodatkowych nakładów pracy i ogranicza współbieżność?

O: ZAWSZE powinieneś podchodzić ostrożnie do synchronizowania jakiegokolwiek kodu, ponieważ — jak słusznie zauważyłeś — wymaga to dodatkowych nakładów związanych ze sprawdzaniem, przydzielaniem i zwalnianiem blokad. Jeśli potrzebujesz ochrony, powinieneś stosować mechanizm synchronizacji, ale musisz mieć na uwadze standardową regułę dotyczącą wszelkich form blokowania zasobów: blokady należy utrzymywać przez możliwie krótki czas, absolutnie niezbędny do zrealizowania założonych celów! Innymi słowy, nie należy synchronizować kodu, który nie zawiera operacji dostępu do chronionego stanu. Staraj się, by Twoje synchronizowane bloki były jak najmniejsze. Załóż blokadę, wejdź do środka, zrealizuj swoje zadania i jak najszybciej zdejmij blokadę i zwolnij zasoby, aby pozostałe wątki mogły wykonać swój kod.

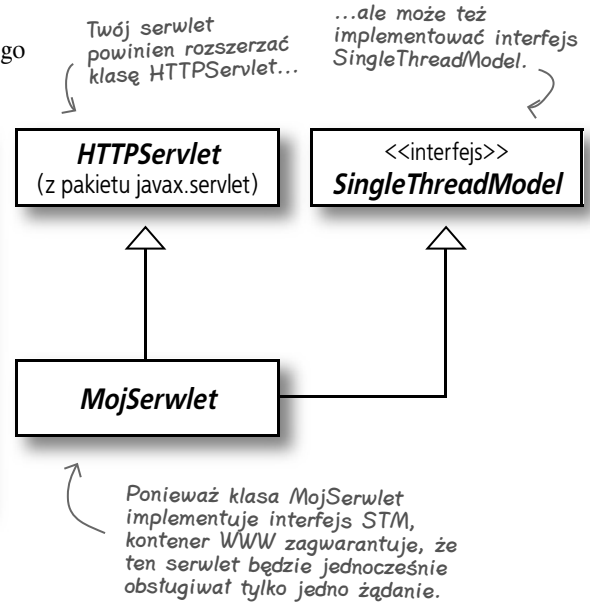
Model jednowątkowy zaprojektowano z myślą o ochronie zmiennych egzemplarzu

Oto, co specyfikacja serwletów mówi o interfejsie modelu jednowątkowego (**SingleThreadModel**, STM):

To jest kluczowy fragment...

Model STM gwarantuje, że serwlety jednocześnie obsługują tylko po jednym żądaniu.

Interfejs `SingleThreadModel` nie zawiera żadnych metod. Jeśli nasz serwlet implementuje ten interfejs, możemy być pewni, że nigdy dwa wątki nie będą jednocześnie wykonywały metody obsługującej tego serwletu. Kontener zarządzający serwletem może to zagwarantować, synchronizując dostęp do jedynego obiektu serwletu lub utrzymując pulę wielu obiektów serwletu i przydzielając kolejne żądania wolnym serwletom.

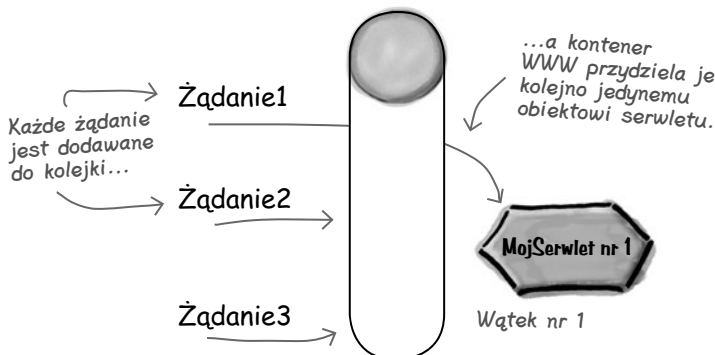


Ale jak kontener może zagwarantować, że jeden serwlet będzie jednocześnie obsługiwał tylko jedno żądanie?

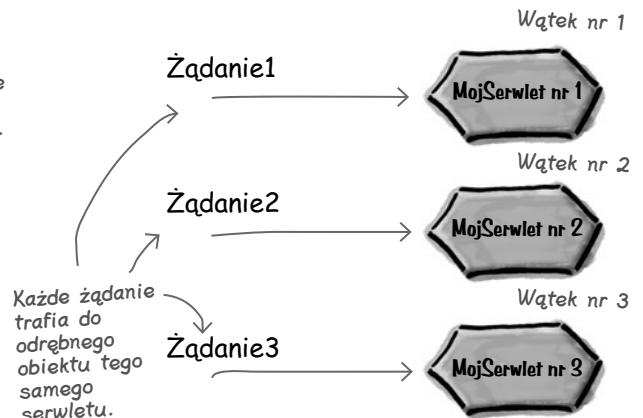
Producent kontenera WWW ma do wyboru dwa rozwiązania — może albo utrzymywać pojedynczy serwlet (takie podejście wymaga kolejowania żądań i oczekiwania z przetwarzaniem kolejnego żądania do chwili zakończenia poprzedniego), albo utworzyć pulę obiektów serwletu (i przetwarzać żądania współbieżnie, po jednym na obiekt serwletu w puli).

Która strategia modelu jednowątkowego wydaje Ci się lepsza?

Kolejkowanie wszystkich żądań



Przekazywanie żądań do obiektów z puli



Która implementacja interfejsu STM jest lepsza?

Raz jeszcze musimy się skonsultować z naszymi dzielnymi karatekami. Powinni wskazać, która implementacja jest lepsza. Przyjrzyjmy się ich walce.



Kolejkowanie wszystkich żądań

Kolejkowanie żądań kierowanych do pojedynczego serwletu jest logiczne. Właśnie taką implementację mieli na myśli twórcy specyfikacji.

Tak, ale to jedyny sposób skutecznej ochrony zmiennych egzemplarza serwletu.

Ach, chyba zbyt mocno wzięłeś sobie do serca przepowiednię znalezioną w ciastku — pamiętaj, mój uczniu, że fortuna bywa bardzo przewrotna...

Specyfikacja serwletów jasno określa, że jedna deklaracja serwletu z deskryptora wdrożenia staje się pojedynczym obiektem serwletu w czasie wykonywania, jednak stosowanie interfejsu STM przekreśla tę definicję. Potrafisz sobie wyobrazić scenariusz, w którym stosowanie wielu obiektów serwletu prowadzi do błędów?

TAK! Właśnie zrozumiałeś, jaką pułapką może być pula obiektów serwletu. W ten sposób rezygnujemy z zasady „jedynego obiektu serwletu”, a sam serwlet traci kontakt z rzeczywistością.

Przekazywanie żądań do obiektów z puli

Ale Mistrzu, czy to nie będzie miało fatalnego wpływu na wydajność? Czyż kolejkowanie żądań nie uniemożliwia dostępu do tego samego serwletu wielu użytkownikom?

Ale Mistrzu, kontener może przecież utworzyć pulę obiektów serwletu. Jedno żądanie może wówczas trafiać do jednego obiektu, drugie do innego itd. Każde żądanie będzie więc obsługiwane równoległe z pozostałymi.

Jesteś zagadkowy, Mistrzu. Co może się nie udać w strategii z pulą obiektów?

Hm, co będzie, jeśli jedna ze zmiennych egzemplarza będzie rejestrowała liczbę przetwarzanych żądań? Taka zmienna mogłaby reprezentować w każdym obiekcie inną wartość, a żaden z tych liczników nie oddawałby faktycznej liczby obsługiwanych żądań... poprawna byłaby tylko suma wartości wszystkich tych zmiennych.

Nie ma niemądrych pytań

P: Co jest grane? Dlaczego specyfikacja serwletów jest tak niezyciowa?

U: Twórcy tej specyfikacji chcieli zapewnić producentom kontenerów jak największą swobodę w kwestii doboru strategii decydujących o wydajności i elastyczności tworzonych rozwiązań.

P: Skąd mogę wiedzieć, którą strategię wybrać producent mojego kontenera?

U: Cóż, przy odrobinie szczęścia odpowiednie informacje powinieneś odnaleźć gdzieś w dokumentacji kontenera; jeśli nie, warto się skontaktować bezpośrednio z producentem.

P: Jak strategia STM wpływa na sposób, w jaki powinienem pisać moje serwlety?

U: Jeśli kontener stosuje strategię kolejkowania żądań, semantyka „jednego obiektu serwletu” jest zachowana, zatem nie musisz wprowadzać żadnych zmian w swoim kodzie. Jeśli jednak kontener stosuje strategię wielu obiektów serwletu w puli, może się zmienić semantyka niektórych zmiennych egzemplarzy. Jeśli na przykład dysponujemy zmienną egzemplarza reprezentującą „licznik żądań”, trudno traktować jej wartość poważnie w sytuacji, gdy stosowana pula zawiera wiele egzemplarzy naszego serwletu. W takim przypadku warto rozważyć przekształcenie zmiennej licznika w zmienną klasową.

P: Ale czy zmienne klasowe gwarantują bezpieczeństwo przetwarzania wielowątkowego?

U: Nie, nie gwarantują, a mechanizm STM nie oferuje w tym obszarze żadnych ułatwień. Interfejs STM co prawda chroni zmienne egzemplarzy przed dostępem współbieżnym, jednak umieszczanie wielu egzemplarzy (obiektów) serwletu w puli istotnie zmienia semantykę serwletu. Co więcej, interfejs STM nie pomaga nam także w obsłudze pozostałych zmiennych ani zasięgów atrybutów — we wszystkich tych kwestiach musimy liczyć na siebie...

P: Czy w takim razie stosowanie interfejsu `SingleThreadModel` w ogóle znajduje uzasadnienie?

U: Nie. Naprawdę. Właśnie dlatego interfejs STM został uznany za przestarzały i niezalecany element interfejsu API serwletów.



Relax

Znajomość tych strategii implementowania interfejsu STM przez kontenery nie jest wymagana na egzaminie. Musisz tylko wiedzieć, że interfejs STM próbuje chronić zmienne egzemplarza serwletu.



Zaostrz ołówki

Zaznacz pola obok struktur, które **NIE** gwarantują bezpieczeństwa przetwarzania wielowątkowego (zaznaczyliśmy już pierwszą pozycję).

- ☒ Atrybuty zasięgu kontekstu
- ☐ Atrybuty zasięgu sesji
- ☐ Atrybuty zasięgu żądania
- ☐ Zmienne klasowe w serwlecie
- ☐ Zmienne lokalne w metodach obsługujących żądania
- ☐ Zmienne statyczne w serwlecie

Ale wiedza o tym interfejsie nadal jest wymagana na egzaminie.

Tylko atrybuty żądania i zmienne lokalne gwarantują bezpieczeństwo przetwarzania wielowątkowego!

To wszystko (kiedy mówimy o *zmiennych lokalnych*, mamy na myśli także parametry metod). Wszystkie inne struktury mogą być przedmiotem manipulacji ze strony wielu wątków, chyba że zrobisz coś, co tym manipulacjom zapobiegnie.

Nie ma niemądrych pytań

P: Zatem zmienne klasowe nie gwarantują bezpieczeństwa przetwarzania wielowątkowego?

U: To prawda. Istnienie wielu klientów przysyłających żądania do danego serwletu musiałoby oczywiście oznaczać współbieżne wykonywanie wielu wątków realizujących kod tego serwletu, a ponieważ wszystkie wątki mają dostęp do zmiennych klasowych serwletu, zmienne te nie są bezpieczne z punktu widzenia przetwarzania wielu wątków.

P: Ale **BYŁYBY** bezpieczne, gdybyśmy zaimplementowali interfejs `SingleThreadModel`, prawda?

U: Tak, ponieważ w takim przypadku nigdy nie mógłby istnieć więcej niż jeden wątek danego serwletu, zatem wszelkie zmienne klasowe byłyby całkowicie bezpieczne z tego punktu widzenia. Musisz jednak pamiętać, że stosowanie tego typu rozwiązań na zawsze zamknęłoby przed Tobą drzwi do klubu programistów serwletów.

P: Moje pytania miały wyłącznie charakter hipotetyczny. To tak, jakbym powiedział: „Co by się stało, gdyby ktoś **BYŁ** na tyle głupi, by implementować interfejs `SingleThreadModel`...”. W życiu nie zrobiłbym czegoś takiego. Jednak pozostając w świecie hipotetycznych rozważań... gdybym miał przyjaciela, który — powiedzmy — synchronizuje metodę obsługującą żądania, to czy wówczas zmienne klasowe **TAKŻE** nie gwarantowałyby bezpieczeństwa przetwarzania wielowątkowego?

U: Tak. Ale Twój przyjaciel byłby kompletnym idiotą. Efekt implementowania interfejsu `SingleThreadModel` jest mniej więcej taki sam jak opisywany już wynik synchronizowania metody obsługującej żądania. Oba rozwiązania kompletnie przeczą idei serwletów i *wcale nie gwarantują ochrony stanu sesji czy atrybutów*.

P: Ale skoro nie należy implementować interfejsu `SingleThreadModel` ani synchronizować metod obsługujących żądania, jak **MOŻEMY** zapewnić bezpieczeństwo przetwarzania wielowątkowego podczas operacji na zmiennych klasowych?

U: To niemożliwe. Jeśli dokładnie przeanalizujesz kod dobrze napisanego serwletu, raczej nie znajdziesz tam żadnych zmiennych klasowych. A już na pewno nie uda Ci się znaleźć zmiennych klasowych, które nie zostały zadeklarowane z modyfikatorem `final` (ponieważ jesteś programistą Javy, zapewne wiesz, że nawet taka zmienna może być modyfikowana, chyba że jest strukturą niezmienną).

Jeśli więc zależy Ci na bezpieczeństwie wątków, nie powinienes w ogóle używać zmiennych klasowych, ponieważ wszystkie wątki tego samego serwletu będą mogły tego typu zmienne modyfikować bez żadnych ograniczeń.

P: Co wobec tego **POWINNIŚMY** zrobić, jeśli potrzebujemy czegoś, co będzie współużytkowane przez wiele egzemplarzy naszego serwletu?

U: Natychmiast przestań! Powiedziałeś „wiele egzemplarzy naszego serwletu”. Chyba się przesłyszałem, przecież jednocześnie może istnieć tylko **JEDEN** egzemplarz serwletu. *Jeden* egzemplarz, *wiele* wątków.

Jeśli chcesz, aby wszystkie wątki miały dostęp do tej samej wartości, musisz zdecydować, który stan atrybutu będzie najwłaściwszy, i zapisać odpowiednią wartość w tym atrybucie. Być może uda Ci się rozwiązać ten problem na jeden z dwóch sposobów:

1. Zadeklaruj odpowiednią zmienną w postaci zmiennej lokalnej wewnątrz metody obsługującej żądania, zamiast — jak do tej pory — w postaci zmiennej klasowej.

LUB

2. Użyj atrybutu w zasięgu, który najbardziej pasuje do danej sytuacji.

Atrybuty żądania i przydzielanie żądań

Stosowanie atrybutów żądania jest uzasadnione w sytuacjach, w których chcemy, aby pozostałe komponenty naszej aplikacji internetowej przejmowały (w części lub w całości) żądania docierające do naszego serwletu. Typowym i stosunkowo prostym przykładem takiego rozwiązania jest aplikacja MVC, która rozpoczyna przetwarzanie od serwletu *kontrolera*, ale która kończy pracę na poziomie strony JSP *widoku*. Kontroler komunikuje się z modelem i odczytuje dane wynikowe, które są niezbędne do skonstruowania odpowiedzi przez widok. Nie ma najmniejszego powodu, by przechowywać te dane w postaci atrybutu kontekstu lub atrybutu sesji, ponieważ mają one zastosowanie *wyłącznie* dla bieżącego żądania, zatem w naturalny sposób pasują do zasięgu żądania.

Warto się teraz zastanowić, jak można sprawić, by inny fragment naszej aplikacji przejął odpowiednio przygotowane żądanie. Użyjemy w tym celu klasy **RequestDispatcher**.

```
// kod w metodzie doGet()
EkspertPiwny be = new EkspertPiwny();
ArrayList wynik = be.getMarki(c);

request.setAttribute("styles", wynik);

RequestDispatcher view =
    request.getRequestDispatcher("wynik.jsp");

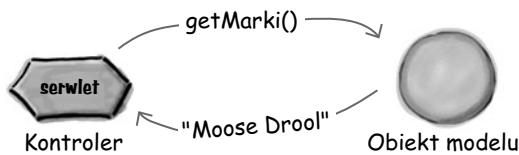
view.forward(request, response);
```

Umieszcza dane modelu w zasięgu żądania.

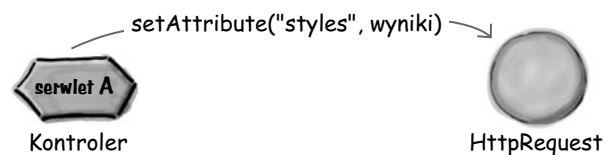
Zwraca obiekt RequestDispatcher dla strony JSP widoku.

Nakazuje stronie JSP przejęcie żądania i, jak nietrudno się domyślić, przekazuje obiekty żądania i odpowiedzi.

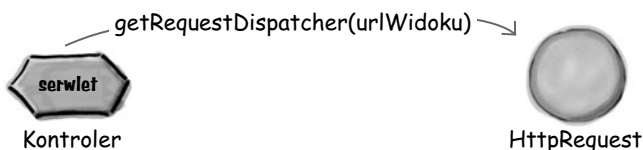
- 1 Serwlet Piwo wywołuje należącą do modelu metodę `getMarki()`, która zwraca pewne dane wymagane przez widok.



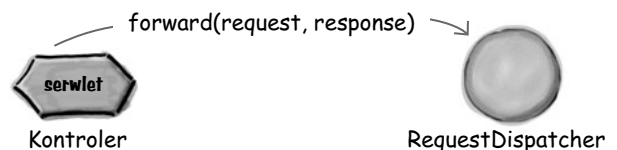
- 2 Serwlet ustawia atrybut żądania nazwany "styles" (wcześniej umieszcza łańcuch "Moose Drool" w strukturze typu ArrayList).



- 3 Serwlet prosi obiekt `HttpServletRequest` o obiekt `RequestDispatcher`, przekazując w wywołaniu odpowiedniej metody względną ścieżkę do strony JSP widoku.



- 4 Serwlet wywołuje metodę `forward()` obiektu `RequestDispatcher`, aby nakazać stronie JSP przejęcie obsługi żądania (nie pokazaliśmy, jak strona JSP odczytuje przekazane żądanie i uzyskuje dostęp do atrybutu "styles" o zasięgu żądania).



Odkrywamy klasę `RequestDispatcher`

Interfejs `RequestDispatcher` udostępnia tylko dwie metody: `forward()` i `include()`. Obie otrzymują w formie argumentów obiekty reprezentujące żądanie i odpowiedź (niezbędne do zrealizowania założonych zadań w komponencie, do którego przekazujemy żądanie). Ze wspomnianej pary metod zdecydowanie bardziej popularna jest `forward()`. Jest bardzo mało prawdopodobne, abyś kiedykolwiek wywoływał metodę `include()` z poziomu serwletu kontrolera; okazuje się jednak, że metoda ta często jest wykorzystywana w tle, w kodzie strony JSP, w standardowej akcji `<jsp:include>` (którą szczegółowo omówimy w rozdziale 8.). Dostęp do obiektu `RequestDispatcher` można uzyskać na dwa sposoby: albo za pośrednictwem obiektu żądania, albo za pośrednictwem obiektu kontekstu aplikacji. Niezależnie od tego, skąd mamy ten obiekt, musimy wskazać komponent aplikacji internetowej, do którego prześlemy żądanie. Innymi słowy, już w metodzie zwracającej obiekt `RequestDispatcher` należy określić docelowy serwlet lub stronę JSP.

<<interfejs>> <i>RequestDispatcher</i>
<i>forward(ServletRequest, ServletResponse)</i> <i>include(ServletRequest, ServletResponse)</i>

`javax.servlet.RequestDispatcher`

Uzyskiwanie obiektu `RequestDispatcher` z obiektu `ServletRequest`

```
RequestDispatcher view = request.getRequestDispatcher("wynik.jsp");
```

Należąca do klasy `ServletRequest` metoda `getRequestDispatcher()` otrzymuje w formie argumentu łańcuch ścieżki do zasobu, do którego zostanie przekazane żądanie. Jeśli przekazana w ten sposób ścieżka rozpoczyna się od znaku prawego ukośnika, kontener przyjmuje, że „ścieżka rozpoczyna się od katalogu głównego danej aplikacji internetowej”. Jeśli natomiast na początku podanej ścieżki NIE ma znaku prawego ukośnika, kontener uznaje, że ma do czynienia ze ścieżką *względną* wobec oryginalnego żądania. W żadnym przypadku nie możesz jednak zmusić kontenera do przeszukiwania katalogów poza strukturą bieżącej aplikacji internetowej. Innymi słowy, użycie w przekazanej ścieżce wyrażenia `".././../"` spowoduje błąd, jeśli okaże się, że ścieżka wskazuje na zasób *poza* katalogiem głównym bieżącej aplikacji.

To jest ścieżka względna (ponieważ nie użyliśmy początkowego znaku prawego ukośnika), zatem w tym przypadku kontener będzie szukał pliku `"wynik.jsp"` w tej samej logicznej lokalizacji, w której „znajduje się” żądanie (szczegóły związane ze ścieżkami względnymi i logicznymi lokalizacjami omówimy w rozdziale poświęconym wdrażaniu aplikacji internetowych).

Uzyskiwanie obiektu `RequestDispatcher` z obiektu `ServletContext`

```
RequestDispatcher view = getServletContext().getRequestDispatcher("/wynik.jsp");
```

Podobnie jak w przypadku odpowiedniej metody klasy `ServletRequest`, także udostępniana w obiekcie kontekstu metoda `getRequestDispatcher()` otrzymuje w formie argumentu łańcuch ścieżki do zasobu, do którego zostanie przekazane żądanie — zasadnicza RÓŻNICA polega na *braku* możliwości określania ścieżek względnych wobec bieżącego zasobu (a więc tego, który otrzymał żądanie). Oznacza to, że ***musisz rozpocząć łańcuch ścieżki od znaku prawego ukośnika!***

W metodzie `getRequestDispatcher()` klasy `ServletContext` zawsze **MUSISZ** używać znaku prawego ukośnika.

Wywoływanie metody `forward()` obiektu `RequestDispatcher`

```
view.forward(request, response);
```

To proste. Otrzymany z kontekstu serwletu lub bieżącego żądania obiekt `RequestDispatcher` wie, gdzie znajduje się zasób, do którego przekazujesz żądanie — jest to zasób (serwlet lub strona JSP), który wskazałeś w postaci argumentu wywołanej wcześniej metody `getRequestDispatcher()`. Zatem w powyższym wywołaniu metody `forward()` mówimy: „Hej, `RequestDispatcher`! Przekaż to żądanie *tam*, gdzie Ci mówiłem (w tym przypadku do odpowiedniej strony JSP), kiedy po raz pierwszy uzyskiwałem do Ciebie dostęp. A oto potrzebne obiekty żądania i odpowiedzi, ponieważ komponent docelowy będzie ich potrzebował do obsługi tego żądania”.

Gdzie popełniliśmy błąd?

Jak myślisz? Czy poniższy kod wykorzystujący obiekt `RequestDispatcher` będzie działał zgodnie z naszymi oczekiwaniami?

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException {
    response.setContentType("application/jar");
    ServletContext ctx = getServletContext();
    InputStream is = ctx.getResourceAsStream("bookCode.jar");
    int read = 0;
    byte[] bytes = new byte[1024];
    OutputStream os = response.getOutputStream();
    while ((read = is.read(bytes)) != -1) {
        os.write(bytes, 0, read);
    }
    os.flush();
    RequestDispatcher view = request.getRequestDispatcher("wynik.jsp");
    view.forward(request, response);
    os.close()
}
```

Przyjmij, że ten fragment kodu działa prawidłowo

Dostaniemy wielki, tłusty wyjątek `IllegalStateException`!



Oglądaj to!

Nie możesz przekazywać żądania, jeśli sam zatwierdziłeś już odpowiedź!

Przez „zatwierdzenie odpowiedzi” rozumiemy „odesłanie odpowiedzi do klienta”. Przeanalizuj powyższy kod raz jeszcze. Zasadniczym problemem jest w tym przypadku wywołanie:

`os.flush();`

Właśnie ten wiersz powoduje, że odpowiedź jest natychmiast wysyłana do klienta, zatem bezpośrednio po jego wykonaniu odpowiedź jest GOTOWA. ZATWIERDZONA. ZAKOŃCZONA. Od tego momentu prawdopodobnie nie będziemy mogli przekazać żądania dalej, ponieważ nasze żądanie przeszło już do historii! Już na nie odpowiedziliśmy, a przecież nie można na jedno żądanie odpowiadać więcej niż raz. Nie daj się więc zwieść, jeśli znajdziesz na egzaminie pytania, w których żądanie zostało przekazane dalej JUŻ PO odesłaniu odpowiedzi do klienta. Kontener wygeneruje wówczas wyjątek `IllegalStateException`.

P: Jak to się stało, że w ogóle nie wspomniałeś o metodzie `include()` interfejsu `RequestDispatcher`?

O: Po pierwsze, metoda ta z pewnością nie pojawi się na egzaminie. Po drugie, wspominaliśmy już, że metoda `include()` jest bardzo rzadko wykorzystywana w praktyce. Aby jednak zaspokoić Twoją ciekawość, wyjaśniamy, że metoda `include()` przekazuje żądanie do przetworzenia przez inny składnik aplikacji (zwykle inny serwet), **który następnie zwraca to żądanie do komponentu, który wywołał metodę `include()`**! Innymi słowy, metoda `include()` oznacza prośbę o pomoc w obsłudze żądania, ale nie wymusza na jego odbiorcy przejęcia całkowitej odpowiedzialności za realizację żądania. Przekazanie żądania ma raczej charakter *tymczasowy*, nie jest więc *trwałym* oddaniem kontroli do pomocniczego składnika aplikacji. Wywołując metodę `forward()`, mówimy: „Oto wspomniane żądanie, nie będę już w żaden sposób przetwarzał ani żądania, ani odpowiedzi”. Warto jednak pamiętać, że wywołując metodę `include()`, mówimy coś zupełnie innego: „Chcę, żeby ktoś inny wykonał pewne działania na tym żądaniu i (lub) odpowiedzi, ale kiedy określone zadania zostaną zrealizowane, sam chcę zakończyć obsługę żądania i odpowiedzi (choćby mogę jeszcze zdecydować o przekazaniu tego żądanie w kolejne miejsce...)”.



Cwiczenie

Zapamiętywanie interfejsów nasłuchujących

ODPOWIEDZI

Interfejsy nasłuchujące dla atrybutów	<div>ServletRequestAttributeListener</div> <div>ServletContextAttributeListener</div> <div>HttpSessionAttributeListener</div>
Interfejsy nasłuchujące pozostałych zdarzeń cyklu życia	<div>ServletRequestListener</div> <div>ServletContextListener</div> <div>HttpSessionListener</div> <div>HttpSessionBindingListener</div> <div>HttpSessionActivationListener</div> <div>(Pamiętaj, że jedyną różnicą pomiędzy tymi interfejsami a interfejsami nasłuchującymi dla atrybutów jest słowo „Attribute” wykorzystywane w nazwach interfejsów).</div>
Metody we wszystkich interfejsach nasłuchujących dla atrybutów (z wyjątkiem tych nasłuchujących zdarzeń wiązania wartości)	<div>attributeAdded()</div> <div>attributeRemoved()</div> <div>attributeReplaced()</div>
Zdarzenia cyklu życia związane z sesjami (z wyłączeniem zdarzeń związanych z atrybutami)	<div>kiedy sesja jest tworzona i niszczona</div> <div>sessionCreated()</div> <div>sessionDestroyed()</div> <div>(Uwaga: istnieją także inne zdarzenia, które omówimy w rozdziale poświęconym sesjom).</div>
Zdarzenia cyklu życia związane z żądaniami (z wyłączeniem zdarzeń związanych z atrybutami)	<div>kiedy żądanie jest inicjalizowane lub niszczone</div> <div>requestInitialized()</div> <div>requestDestroyed()</div> <div>(Zwróć uwagę na różnicę pomiędzy zdarzeniami <u>sesji</u> a zdarzeniami <u>żądania</u> — w przypadku sesji mamy do czynienia ze zdarzeniem sessionCreated(), natomiast w przypadku żądań odpowiednie zdarzenie nosi nazwę request<u>Initialized</u>()).</div>
Zdarzenia cyklu życia związane z kontekstem serwletu (z wyłączeniem zdarzeń związanych z atrybutami)	<div>kiedy kontekst jest inicjalizowany lub niszczony</div> <div>contextInitialized()</div> <div>contextDestroyed()</div>



Cwiczenie

Zasięg atrybutów

ODPOWIEDZI

	Dostępność (kto widzi atrybut)	Zasięg (jak długo ten atrybut żyje)	Do czego służy?
Kontekst (NIE gwarantuje bezpieczeństwa przetwarzania wielowątkowego)	Dowolny składnik danej aplikacji internetowej, włącznie z serwetami, stronami JSP, obiektami ServletContextListener oraz obiektami ServletContextAttribute-Listener.	Czas życia obiektu ServletContext, czyli czas istnienia wdrożonej i uruchomionej aplikacji. Jeśli działanie serwera lub aplikacji zostanie przerwane, kontekst (wraz ze swoimi atrybutami) zostanie automatycznie zniszczony.	Zasoby, które mają być współużytkowane przez całą aplikację, a więc takie struktury jak połączenia z bazami danych, identyfikatory JNDI, adresy poczty elektronicznej itp.
Sesja HTTP (NIE gwarantuje bezpieczeństwa przetwarzania wielowątkowego)	Dowolne serwlety i strony JSP, które mają dostęp do określonej sesji. Pamiętaj, że sesja wykracza poza pojedyncze żądanie klienta i obejmuje wiele żądań nadsyłanych przez tego samego klienta (które mogą być kierowane do różnych serwetów).	Życie sesji. Sesja może być zniszczona albo programowo, albo po upływie czasu określonego z góry czasu oczekiwania (szczegóły na ten temat znajdziesz w rozdziale poświęconym zarządzaniu sesjami).	Dane i zasoby związane z sesją klienta, nie tylko z jednym żądaniem. Coś, co wymaga dłuższej konwersacji z klientem (obejmującej wiele żądań). Typowym przykładem jest koszyk z zakupami w sklepie internetowym.
Żądanie (zapewnia bezpieczeństwo przetwarzania wielowątkowego)	Dowolny składnik danej aplikacji internetowej, który ma bezpośredni dostęp do obiektu żądania. W większości przypadków dotyczy to wyłącznie serwetów i stron JSP, do których żądanie jest przekazywane za pośrednictwem obiektu RequestDispatcher. Dostęp do atrybutów żądania mają także związane z tym żądaniem obiekty nasłuchujące.	Życie żądania, a więc cały czas wykonywania metody service() serwetu. Innymi słowy, czas życia wątku (stosu) obsługującego dane żądanie.	Przekazywanie informacji modelu z kontrolera do widoku lub dowolnych innych danych właściwych dla pojedynczego żądania klienta.



Ćwiczenie

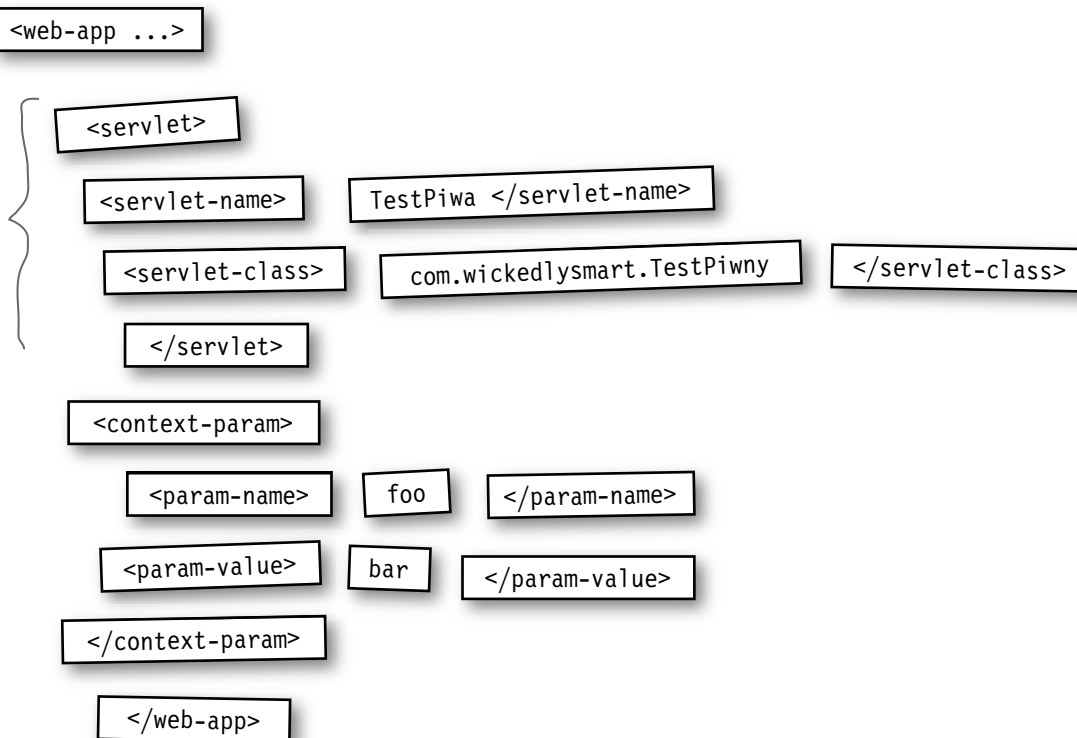


Magnesiki z kodem

ODPOWIEDZI

(konfiguracja parametrów kontekstu w deskrytorze wdrożenia)

Ta część NIE jest wymagana.



Niewykorzystane:

Element `<init-param>` jest wykorzystywany dla parametrów inicjalizacji serwletu, nie dla parametrów inicjalizacji kontekstu. Element ten stosuje się WYŁĄCZNIE w ramach elementu `<servlet>`.

Nie istnieje coś takiego jak element `<servlet-param>`.

`</init-param>`

`</servlet-param>`

`<init-param>`

`<servlet-param>`



**BAR
KAWOWY**

Examin próbny

1 Która metoda może doprowadzić do wystąpienia wyjątku **IllegalStateException** podczas stosowania obiektu **RequestDispatcher**? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. **read**
- ☐ B. **flush**
- ☐ C. **write**
- ☐ D. **getOutputStream**
- ☐ E. **getRescueAsStream**

2 Które zdania o parametrach inicjalizacji obiektu kontekstu **ServletContext** są prawdziwe? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. Powinny być stosowane dla danych, które rzadko podlegają zmianom.
- ☐ B. Powinny być stosowane dla często modyfikowanych danych.
- ☐ C. Dostęp do nich można uzyskiwać za pomocą metody **ServletContext.getParameter()**.
- ☐ D. Dostęp do nich można uzyskiwać za pomocą metody **ServletContext.getInitParameter()**.
- ☐ E. Powinny być stosowane dla danych właściwych dla konkretnego serwletu.
- ☐ F. Powinny być wykorzystywane dla danych mających zastosowanie w całej aplikacji internetowej.

-
- 2 Które z wymienionych typów definiują metody **getAttribute()** i **setAttribute()**? (Zaznacz wszystkie prawidłowe odpowiedzi).
- ☐ A. **HttpSession**
 - ☐ B. **ServletRequest**
 - ☐ C. **ServletResponse**
 - ☐ D. **ServletContext**
 - ☐ E. **ServletConfig**
 - ☐ F. **SessionConfig**
-
- 4 Jeśli serwet zostanie wywołany za pomocą metody **forward()** lub **include()** obiektu **RequestDispatcher**, które metody obiektu żądania serwletu będą miały dostęp do atrybutów tego żądania ustawionych przez kontener? (Zaznacz wszystkie prawidłowe odpowiedzi).
- ☐ A. **getCookies()**
 - ☐ B. **getAttribute()**
 - ☐ C. **getRequestPath()**
 - ☐ D. **getRequestAttribute()**
 - ☐ E. **getRequestDispatcher()**
-
- 5 Które wywołania zwracają informacje na temat parametrów inicjalizacji mające zastosowanie we wszystkich składnikach bieżącej aplikacji internetowej? (Zaznacz wszystkie prawidłowe odpowiedzi).
- ☐ A. **ServletConfig.getInitParameters()**
 - ☐ B. **ServletContext.getInitParameters()**
 - ☐ C. **ServletConfig.getInitParameterNames()**
 - ☐ D. **ServletContext.getInitParameterNames()**
 - ☐ E. **ServletConfig.getInitParameter(String)**
 - ☐ F. **ServletContext.getInitParameter(String)**

- 6 Które zdania o interfejsach nasłuchujących są prawdziwe? (Zaznacz wszystkie prawidłowe odpowiedzi).
- ☐ A. Interfejs **ServletResponseListener** można wykorzystać do podejmowania pewnych działań już po odesłaniu odpowiedzi.
 - ☐ B. Interfejs **ServletSessionListener** można wykorzystać do podejmowania pewnych działań już po wyczerpaniu czasu sesji **HttpSession**.
 - ☐ C. Interfejs **ServletContextListener** można wykorzystać do podejmowania pewnych działań w czasie, gdy kontekst serwletu ma zostać zniszczony.
 - ☐ D. Interfejs **ServletRequestAttributeListener** można wykorzystać do podejmowania pewnych działań w momencie, gdy jakiś atrybut zostaje usunięty z obiektu **ServletRequest**.
 - ☐ E. Interfejs **ServletContextAttributeListener** można wykorzystać do podejmowania pewnych działań bezpośrednio po utworzeniu kontekstu serwletu i udostępnieniu go metodzie obsługującej pierwsze żądanie.
-
- 7 Która z wymienionych poniżej struktur w największym stopniu nadaje się do składowania w postaci atrybutu zasięgu sesji?
- ☐ A. Kopia wpisanego przez użytkownika parametru zapytania.
 - ☐ B. Wynik zapytania wykonanego w bazie danych, który bezzwłocznie ma zostać zwrócony do użytkownika.
 - ☐ C. Wykorzystywany przez wszystkie komponenty danego systemu obiekt połączenia z bazą danych.
 - ☐ D. Obiekt reprezentujący użytkownika, który właśnie się zalogował w systemie.
 - ☐ E. Uzyskana za pośrednictwem obiektu **ServletContext** kopia parametru inicjalizacji.

- 8 Przeanalizuj poniższy (składniad prawidłowy) fragment kodu klasy rozszerzającej klasę bazową **HttpServletRequest**, która dodatkowo została zarejestrowana jako implementacja interfejsu **ServletRequestAttributeListener**:

```
10. public void doGet(HttpServletRequest req, HttpServletResponse res)
11.             throws IOException, ServletException {
12.     req.setAttribute("a", "b");
13.     req.setAttribute("a", "c");
14.     req.removeAttribute("a");
15. }
16. public void attributeAdded(ServletRequestAttributeEvent ev) {
17.     System.out.print("A:" + ev.getName() + "->" + ev.getValue());
18. }
19. public void attributeRemoved(ServletRequestAttributeEvent ev) {
20.     System.out.print("M:" + ev.getName() + "->" + ev.getValue());
21. }
22. public void attributeReplaced(ServletRequestAttributeEvent ev) {
23.     System.out.print("P:" + ev.getName() + "->" + ev.getValue());
24. }
```

Jakie zdarzenia zostaną wygenerowane i zarejestrowane w pliku dziennika?

- ☐ A. A:a->b P:a->b
- ☐ B. A:a->b M:a->c
- ☐ C. A:a->b P:a->b M:a->c
- ☐ D. A:a->b P:a->b P:a->null
- ☐ E. A:a->b M:a->b A:a->c M:a->c
- ☐ F. A:a->b M:a->b A:a->c P:a->null

- 9 Które z wymienionych poniżej podelementów elementu **<listener>** są wymagane podczas deklarowania obiektu nasłuchującego w deskrytorze wdrożenia? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. **<description>**
- ☐ B. **<listener-name>**
- ☐ C. **<listener-type>**
- ☐ D. **<listener-class>**
- ☐ E. **<servlet-mapping>**

10 Które z wymienionych poniżej typów obiektów mogą przechowywać atrybuty? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. **ServletConfig**
- ☐ B. **ServletResponse**
- ☐ C. **RequestDispatcher**
- ☐ D. **HttpServletRequest**
- ☐ E. **HttpSessionContext**

11 Które zdanie jest prawdziwe? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. Kiedy aplikacja internetowa przygotowuje się do zakończenia działania, kolejność sygnalizowania zdarzeń obiektom nasłuchującym nie jest w żaden sposób gwarantowana.
- ☐ B. Kiedy występują zdarzenia właściwe dla obiektów nasłuchujących, kolejności wywoływania ich metod nie można przewidzieć.
- ☐ C. Kontener rejestruje obiekty nasłuchujące na podstawie deklaracji zawartych w deskrytorze wdrożenia.
- ☐ D. Tylko kontener może unieważnić sesję.

12 Która z przedstawionych poniżej opinii na temat interfejsu **RequestDispatcher** jest prawdziwa (tam, gdzie takie podejście jest uzasadnione, przyjmij, że obiekt **RequestDispatcher** nie został uzyskany za pośrednictwem wywołania metody **getNamedDispatcher()**)? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. Obiekt **RequestDispatcher** można wykorzystywać do przekazywania żądań do innych serwletów.
- ☐ B. Jediną metodą interfejsu **RequestDispatcher** jest **forward()**.
- ☐ C. Wykorzystywane podczas tworzenia obiektu **RequestDispatcher** parametry z łańcucha zapytania nie są już przekazywane dalej przez metodę **forward()**.
- ☐ D. Serwlet, do którego dociera przekazywane dalej żądanie, może uzyskać dostęp do oryginalnego łańcucha zapytania przez wywołanie metody **getQueryString()** otrzymanego obiektu **HttpServletRequest**.
- ☐ E. Serwlet, do którego dociera przekazywane dalej żądanie, może uzyskać dostęp do oryginalnego łańcucha zapytania przez wywołanie metody **getAttribute("javax.servlet.query_string")** otrzymanego obiektu **ServletRequest**.

- 13
- Która technika zapewniania bezpieczeństwa przetwarzania wielowątkowego w aplikacjach opartych na serwetach jest zalecana?
- ☐

A. Napisanie kodu serwletu rozszerzającego klasę bazową **ThreadSafeServlet**.
- ☐

B. Zaimplementowanie w klasie serwletu interfejsu **SingleThreadModel**.
- ☐

C. Rejestrowanie wszystkich wywołań metody serwletu.
- ☐

D. Stosowanie wyłącznie zmiennych lokalnych, a w razie konieczności użycia zmiennych klasowych, synchronizacja dostępu do składowanych wartości.

- 14
- Dysponując następującymi metodami:
- **getCookies**

– **getContextPath**

– **getAttributes**
- dopasuj je do wymienionych poniżej klas i interfejsów. Pamiętaj, że każda z tych metod może występować więcej niż raz.
- | | | | |
|---------------------------|-------|-------|-------|
| HttpSession | | | |
| ServletContext | | | |
| HttpServletRequest | | | |
- 15

Które zdanie o interfejsie **RequestDispatcher** jest prawdziwe?
(Zaznacz wszystkie prawidłowe odpowiedzi).

☐

A. Z dwóch metod tego interfejsu metoda **forward()** jest stosowana częściej.

☐

B. Metody tego interfejsu otrzymują na wejściu następujące argumenty: zasób, żądanie i odpowiedź.

☐

C. W zależności od klasy, której metodę wykorzystano do utworzenia obiektu **RequestDispatcher**, ścieżka do zasobu docelowego może być zmieniana.

☐

D. Niezależnie od klasy, której metodę wykorzystano do utworzenia obiektu **RequestDispatcher**, ścieżka do zasobu docelowego NIE jest zmieniana.

☐

E. Serwlet wywołujący metodę **RequestDispatcher.forward** może uprzednio wysłać klientowi własną odpowiedź, ale nigdy po tym wywołaniu.
- 244 Rozdział 5.
- ebookpoint kopia dla: Pawel Domanski pawel.domanski@outlook.com G0629171329



BAR KAWOWY

Egzamin próbny — odpowiedzi

1 Która metoda może doprowadzić do wystąpienia wyjątku **IllegalStateException** podczas stosowania obiektu **RequestDispatcher**? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. **read**
- ☒ B. **flush**
- ☐ C. **write**
- ☐ D. **getOutputStream**
- ☐ E. **getRescueAsStream**

(Specyfikacja serwletów 2.4, str. 167).

— Wyjątek **IllegalStateException** ma miejsce w sytuacji, gdy z jednej strony odpowiedź została już „zatwierdzona” i odesłana do klienta (przez wywołanie metody **flush()**), z drugiej strony próbujemy przekazać żądanie dalej.

2 Które zdania o parametrach inicjalizacji obiektu kontekstu **ServletContext** są prawdziwe? (Zaznacz wszystkie prawidłowe odpowiedzi).

(Specyfikacja serwletów 2.4, str. 31).

- ☒ A. Powinny być stosowane dla danych, które rzadko podlegają zmianom.
- ☐ B. Powinny być stosowane dla często modyfikowanych danych.
- ☐ C. Dostęp do nich można uzyskiwać za pomocą metody **ServletContext.getParameter()**.
- ☒ D. Dostęp do nich można uzyskiwać za pomocą metody **ServletContext.getInitParameter()**.
- ☐ E. Powinny być stosowane dla danych właściwych dla konkretnego serwletu.
- ☒ F. Powinny być wykorzystywane dla danych mających zastosowanie w całej aplikacji internetowej.

— Odpowiedź B jest nieprawidłowa, ponieważ parametry inicjalizacji obiektu **ServletContext** są odczytywane tylko w momencie uruchamiania kontenera.

— Odpowiedź C jest nieprawidłowa, ponieważ taka metoda w ogóle nie istnieje.

— Odpowiedź E jest nieprawidłowa, ponieważ dla każdej aplikacji internetowej może istnieć tylko jeden obiekt **ServletContext**.

- 2 Które z wymienionych typów definiują metody **getAttribute()** i **setAttribute()**?
(Zaznacz wszystkie prawidłowe odpowiedzi).

(Specyfikacja serwletów 2.4,
str. 32, 36, 59).

- ☒ A. **HttpSession**
- ☒ B. **ServletRequest**
- ☐ C. **ServletResponse**
- ☒ D. **ServletContext**
- ☐ E. **ServletConfig**
- ☐ F. **SessionConfig**

- 4 Jeśli serwlet zostanie wywołany za pomocą metody **forward()** lub **include()** obiektu **RequestDispatcher**, które metody obiektu żądania serwletu będą miały dostęp do atrybutów tego żądania ustawionych przez kontener? (Zaznacz wszystkie prawidłowe odpowiedzi).

(Specyfikacja serwletów 2.4,
str. 65 – 66).

- ☐ A. **getCookies()**
- ☒ B. **getAttribute()**
— Odpowiedź B zawiera właściwą metodę. Za jej pomocą możemy uzyskać dostęp do ustawianych przez kontener atrybutów `javax.servlet.forward.Xxx` oraz `javax.servlet.include.Xxxx`.
- ☐ C. **getRequestPath()**
- ☐ D. **getRequestAttribute()**
— Odpowiedzi C i D odwołują się do metod, które w ogóle nie istnieją.
- ☐ E. **getRequestDispatcher()**

- 5 Które wywołania zwracają informacje na temat parametrów inicjalizacji mające zastosowanie we wszystkich składnikach bieżącej aplikacji internetowej? (Zaznacz wszystkie prawidłowe odpowiedzi).

(Specyfikacja serwletów 2.4, str. 32).

- ☐ A. **ServletConfig.getInitParameters()**
- ☐ B. **ServletContext.getInitParameters()**
- ☐ C. **ServletConfig.getInitParameterNames()**
- ☒ D. **ServletContext.getInitParameterNames()**
- ☐ E. **ServletConfig.getInitParameter(String)**
- ☒ F. **ServletContext.getInitParameter(String)**

— Odpowiedzi A i B są nieprawidłowe, ponieważ wymienione tam metody w ogóle nie istnieją.

— Odpowiedzi C i E są nieprawidłowe, ponieważ wymienione tam metody oferują dostęp wyłącznie do parametrów inicjalizacji serwletu.

- 6 Które zdania o interfejsach nasłuchujących są prawdziwe? (Zaznacz wszystkie prawidłowe odpowiedzi). (Specyfikacja serwletów 2.4, str. 80).
- ☐ A. Interfejs **ServletResponseListener** można wykorzystać do podejmowania pewnych działań już po odesłaniu odpowiedzi.
 - ☒ B. Interfejs **ServletSessionListener** można wykorzystać do podejmowania pewnych działań już po wyczerpaniu czasu sesji **HttpSession**.
 - ☒ C. Interfejs **ServletContextListener** można wykorzystać do podejmowania pewnych działań w czasie, gdy kontekst serwletu ma zostać zniszczony.
 - ☒ D. Interfejs **ServletRequestAttributeListener** można wykorzystać do podejmowania pewnych działań w momencie, gdy jakiś atrybut zostaje usunięty z obiektu **ServletRequest**.
 - ☐ E. Interfejs **ServletContextAttributeListener** można wykorzystać do podejmowania pewnych działań bezpośrednio po utworzeniu kontekstu serwletu i udostępnieniu go metodzie obsługującej pierwsze żądanie.
- Odpowiedź A jest niepoprawna, ponieważ interfejs **ServletResponseListener** nie istnieje.
- Odpowiedź E jest niepoprawna, ponieważ do tego celu wykorzystuje się interfejs **ServletContextListener**.

- 7 Która z wymienionych poniżej struktur w największym stopniu nadaje się do składowania w postaci atrybutu zasięgu sesji? (Specyfikacja serwletów 2.4, str. 58).
- ☐ A. Kopia wpisanego przez użytkownika parametru zapytania.
 - ☐ B. Wynik zapytania wykonanego w bazie danych, który bezzwłocznie ma zostać zwrócony do użytkownika.
 - ☐ C. Wykorzystywany przez wszystkie komponenty danego systemu obiekt połączenia z bazą danych.
 - ☒ D. Obiekt reprezentujący użytkownika, który właśnie się zalogował w systemie.
 - ☐ E. Uzyskana za pośrednictwem obiektu **ServletContext** kopia parametru inicjalizacji.
- Odpowiedź A jest niepoprawna, ponieważ parametr zapytania jest zazwyczaj wykorzystywany bezpośrednio do wykonywania pewnych operacji.
- Odpowiedź B jest niepoprawna, ponieważ zazwyczaj takie dane są albo natychmiast zwracane, albo umieszczane w zasięgu żądania.
- Odpowiedź C jest niepoprawna, ponieważ tego typu obiekty (które nie są przecież związane z określoną sesją) powinny być składowane w zasięgu kontekstu.
- Odpowiedź E jest niepoprawna, ponieważ parametry kontekstu serwletu powinny pozostawać w obiekcie **ServletContext**.

- 8 Przeanalizuj poniższy (składniad prawidłowy) fragment kodu klasy rozszerzającej klasę bazową **HttpServlet**, która dodatkowo została zarejestrowana jako implementacja interfejsu **ServletRequestAttributeListener**:

(Specyfikacja serwletów 2.4, str. 199 – 200).

```
10. public void doGet(HttpServletRequest req, HttpServletResponse res)
11.             throws IOException, ServletException {
12.     req.setAttribute("a", "b");
13.     req.setAttribute("a", "c");
14.     req.removeAttribute("a");
15. }
16. public void attributeAdded(ServletRequestAttributeEvent ev) {
17.     System.out.print("A:" + ev.getName() + "->" + ev.getValue());
18. }
19. public void attributeRemoved(ServletRequestAttributeEvent ev) {
20.     System.out.print("M:" + ev.getName() + "->" + ev.getValue());
21. }
22. public void attributeReplaced(ServletRequestAttributeEvent ev) {
23.     System.out.print("P:" + ev.getName() + "->" + ev.getValue());
24. }
```

Jakie zdarzenia zostaną wygenerowane i zarejestrowane w pliku dziennika?

- ☐ A. A:a->b P:a->b
- ☐ B. A:a->b M:a->c
- ☒ C. A:a->b P:a->b M:a->c
- ☐ D. A:a->b P:a->b P:a->>null
- ☐ E. A:a->b M:a->b A:a->c M:a->c
- ☐ F. A:a->b M:a->b A:a->c P:a->>null

— Uwaga! Jeśli atrybut został zastąpiony, metoda `getValue()` zwraca STARĄ wartość tego atrybutu.

- 9 Które z wymienionych poniżej podelementów elementu **<listener>** są wymagane podczas deklarowania obiektu nasłuchującego w deskrypcji wdrożenia? (Zaznacz wszystkie prawidłowe odpowiedzi).

(Specyfikacja serwletów 2.4, podrozdział 10.4, §13.4.9).

- ☐ A. **<description>**
- ☐ B. **<listener-name>**
- ☐ C. **<listener-type>**
- ☒ D. **<listener-class>**
- ☐ E. **<servlet-mapping>**

— **<listener-class>** jest JEDYNYM wymaganym podelementem elementu **<listener>**.

10 Które z wymienionych poniżej typów obiektów mogą przechowywać atrybuty? (API)
(Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. **ServletConfig** — Odpowiedzi A, B i C są niepoprawne, ponieważ te typy nie zawierają atrybutów.
- ☐ B. **ServletResponse**
- ☐ C. **RequestDispatcher**
- ☒ D. **HttpServletRequest**
- ☐ E. **HttpSessionContext** — Odpowiedź E jest niepoprawna, ponieważ taki typ w ogóle nie istnieje.

Uwaga: Do pozostałych typów związanych z serwetami, które mogą przechowywać atrybuty, należą klasy `HttpSession` i `ServletContext`.

11 Które zdanie jest prawdziwe? (Zaznacz wszystkie prawidłowe odpowiedzi). (Specyfikacja serwetów 2.4, str. 81 – 84).

- ☐ A. Kiedy aplikacja internetowa przygotowuje się do zakończenia działania, kolejność sygnalizowania zdarzeń obiektom nasłuchującym nie jest w żaden sposób gwarantowana.
- ☐ B. Kiedy występują zdarzenia właściwe dla obiektów nasłuchujących, kolejności wywoływania ich metod nie można przewidzieć.
- ☒ C. Kontener rejestruje obiekty nasłuchujące na podstawie deklaracji zawartych w deskrypcorze wdrożenia.
- ☐ D. Tylko kontener może unieważnić sesję.
- Odpowiedzi A i B są niepoprawne, ponieważ kontener wykorzystuje deskryptor wdrożenia do określania kolejności informowania zarejestrowanych obiektów nasłuchujących o wykrywanych zdarzeniach.
- Odpowiedź D jest niepoprawna, ponieważ także serwet może unieważnić sesję za pomocą metody `HttpSession.invalidate()`.

12 Która z przedstawionych poniżej opinii na temat interfejsu **RequestDispatcher** jest prawdziwa (tam, gdzie takie podejście jest uzasadnione, przyjmij, że obiekt **RequestDispatcher** nie został uzyskany za pośrednictwem wywołania metody **getNamedDispatcher()**)? (Zaznacz wszystkie prawidłowe odpowiedzi). (Specyfikacja serwetów 2.4, str. 65).

- ☒ A. Obiekt **RequestDispatcher** można wykorzystywać do przekazywania żądań do innych serwetów.
- ☐ B. Jediną metodą interfejsu **RequestDispatcher** jest **forward()**.
- ☐ C. Wykorzystywane podczas tworzenia obiektu **RequestDispatcher** parametry z łańcucha zapytania nie są już przekazywane dalej przez metodę **forward()**.
- ☐ D. Serwet, do którego dociera przekazywane dalej żądanie, może uzyskać dostęp do oryginalnego łańcucha zapytania przez wywołanie metody **getQueryString()** otrzymanego obiektu **HttpServletRequest**.
- ☒ E. Serwet, do którego dociera przekazywane dalej żądanie, może uzyskać dostęp do oryginalnego łańcucha zapytania przez wywołanie metody **getAttribute("javax.servlet.query_string")** otrzymanego obiektu **ServletRequest**.
- Odpowiedź B jest niepoprawna, ponieważ interfejs `RequestDispatcher` zawiera także metodę `include()`.
- Odpowiedź C jest niepoprawna, ponieważ parametry łańcucha są w takim przypadku przekazywane dalej.
- Odpowiedź D jest niepoprawna, ponieważ metoda `getQueryString()` zwraca łańcuch zapytania dla adresu URL uzyskanego z obiektu `RequestDispatcher`.

- 13

Która technika zapewniania bezpieczeństwa przetwarzania wielowątkowego w aplikacjach opartych na serwetach jest zalecana?

(Specyfikacja serwetów 2.4, str. 27).

— Odpowiedzi A i B są niepoprawne, ponieważ interfejs Servlet API w ogóle nie definiuje klasy ThreadSafeServlet, a interfejs SingleThreadModel począwszy od wersji 2.4 uznano za przestarzały i jako taki jest niezalecany.

☐

A.

Napisanie kodu serwletu rozszerzającego klasę bazową **ThreadSafeServlet**.

☐

B.

Zaimplementowanie w klasie serwletu interfejsu **SingleThreadModel**.

☐

C.

Rejestrowanie wszystkich wywołań metody serwletu.

☒

D.

Stosowanie wyłącznie zmiennych lokalnych, a w razie konieczności użycia zmiennych klasowych, synchronizacja dostępu do składowanych wartości.

- 14

Dysponując następującymi metodami:

(API)

– **getCookies**

– **getContextPath**

– **getAttributes**

dopasuj je do wymienionych poniżej klas i interfejsów. Pamiętaj, że każda z tych metod może występować więcej niż raz.

HttpSession	<code>getAttribute</code>
ServletContext	<code>getAttribute</code>	<code>getContextPath</code>
HttpServletRequest	<code>getCookies</code>	<code>getAttribute</code>	<code>getContextPath</code>

Na tym etapie nie musisz się tego uczyć na pamięć — wystarczy, jeśli zapamiętasz, które metody mają sens w poszczególnych zasięgach.
- 15

Które zdanie o interfejsie **RequestDispatcher** jest prawdziwe? (Zaznacz wszystkie prawidłowe odpowiedzi).

(API)

☒

A.

Z dwóch metod tego interfejsu metoda **forward()** jest stosowana częściej.

☐

B.

Metody tego interfejsu otrzymują na wejściu następujące argumenty: zasób, żądanie i odpowiedź.

— Odpowiedź B: zasób jest określany już w momencie tworzenia obiektu.

☒

C.

W zależności od klasy, której metodę wykorzystano do utworzenia obiektu **RequestDispatcher**, ścieżka do zasobu docelowego może być zmieniana.

☐

D.

Niezależnie od klasy, której metodę wykorzystano do utworzenia obiektu **RequestDispatcher**, ścieżka do zasobu docelowego NIE jest zmieniana.

☐

E.

Serwlet wywołujący metodę **RequestDispatcher.forward** może uprzednio wysłać klientowi własną odpowiedź, ale nigdy po tym wywołaniu.

— Odpowiedź E: jeśli Twój serwlet korzysta z obiektu RequestDispatcher, nigdy nie może wysyłać własnych odpowiedzi.
- 250 Rozdział 5.
- ebookpoint kopia dla: Pawel Domanski pawel.domanski@outlook.com G0629171329

6. Zarządzanie sesjami

Stan konwersacyjny



Zaraz po odesłaniu do nas odpowiedzi serwery WWW zapominają, kim jesteśmy. Kiedy wysyłamy kolejne żądanie, docelowy serwer WWW już nas nie rozpoznaje. Innymi słowy, serwery WWW nie pamiętają ani tego, czego żądaliśmy w przeszłości, ani tego, co otrzymaliśmy wówczas w ramach odpowiedzi. Serwery po prostu o wszystkim zapominają. W wielu przypadkach takie zachowanie jest zupełnie prawidłowe, niekiedy jednak konieczne jest utrzymywanie pomiędzy klientem a serwerem stanu konwersacyjnego *obejmującego wiele żądań*. Koszyk z zakupami nie będzie funkcjonował prawidłowo, jeśli klient będzie musiał wybierać interesujące go artykuły i dokonywać przelewu w *pojedynczym żądaniu*. **Okazuje się, że zadziwiająco proste rozwiązanie tego problemu jest oferowane w ramach interfejsu API serwletów.**



Zarządzanie sesjami

- 4.1.** Napisz kod serwletu, który będzie zapisywał i odczytywał rozmaite obiekty z obiektu sesji.
- 4.2.** Mając dany określony scenariusz, opisz interfejsy API wykorzystywane do uzyskiwania dostępu do obiektu sesji; wyjaśnij, kiedy obiekt sesji jest tworzony, oraz opisz mechanizmy stosowane do niszczenia obiektów sesji (napisz także, kiedy tego typu obiekty są niszczone).
- 4.3.** Wykorzystując interfejsy nasłuchujące zdarzeń związanych z sesją, napisz kod, który będzie odpowiadał na zdarzenie dodania obiektu do sesji; przygotuj także kod reagujący na zdarzenie migracji obiektu sesji z jednej wirtualnej maszyny Javy do innej.
- 4.4.** Mając dany określony scenariusz, opisz, który mechanizm zarządzania sesjami może być stosowany w kontenerach WWW, w jaki sposób znaczniki kontekstu klienta (tzw. ciasteczka, ang. *cookies*) mogą być wykorzystywane podczas zarządzania sesjami i jak w zarządzaniu sesjami może pomóc technika przepisywania adresów URL; napisz kod serwletu stosującego technikę przepisywania adresów URL.

Wdrażanie aplikacji internetowych

Wszystkie cztery cele egzaminu związane z zarządzaniem sesjami zostaną dogłębnie omówione w tym rozdziale (choć o niektórych z nich wspominaliśmy w poprzednim rozdziale). Zapoznanie się z treścią tego rozdziału jest Twoją jedyną szansą na poznanie i zapamiętanie prezentowanych zagadnień, a zatem przygotuj się na jego długotrwałe studiowanie.

Chcę, żeby moja aplikacja
piwna zwracała klientowi odpowiedź
i wymuszała konwersację... czyż nie byłoby
wspaniale, gdyby użytkownik odpowiadał na
pytanie, po czym otrzymywał odpowiedź
aplikacji z nowym pytaniem wygenerowanym
na podstawie poprzednich
odpowiedzi?



Kim chce poprzez wiele żądań utrzymywać stan specyficzny dla danego klienta

Logika biznesowa istniejącego obecnie modelu sprowadza się do prostego sprawdzania parametru przekazanego wraz z żądaniem i odesłania odpowiedzi (*porady*). Nikt w aplikacji nie pamięta *żadnych* zdarzeń, które wiązałyby się z komunikacją pomiędzy serwerem a danym klientem sprzed bieżącego żądania.

Kim ma **TERAZ**:

```
public class EkspertPiwny {
    public ArrayList getMarki(String kolor) {
        ArrayList marki = new ArrayList();
        if (kolor.equals("bursztynowy")) {
            marki.add("Jack Amber");
            marki.add("Red Moose");
        } else {
            marki.add("Jail Pale Ale");
            marki.add("Gout Stout");
        }
        return marki;
    }
}
```

Sprawdzamy jedyny
przychodzący parametr
(wybrany kolor piwa)
i odsyłamy ostateczną
odповідź (tablicę
z markami piw, które
pasują do wybranego
koloru). Taki system
porad nie jest
szczególnie przydatny...

Kim **CHCE** mieć:

```
public class EkspertPiwny {
    public NastepnaOdpowiedz getPorada(String answer) {
        // Przetwarza odpowiedź klienta przez analizę
        // nie tylko WSZYSTKICH wcześniejszych odpowiedzi
        // danego klienta, ale także bieżącego żądania.
        // Jeśli zebrane informacje są wystarczające,
        // metoda zwraca ostateczną poradę; w przeciwnym
        // przypadku metoda wysyła do klienta kolejne
        // pytanie.
    }
}
```

Przyjmij, że klasa *NastepnaOdpowiedz* obejmuje
nie tylko zawartość kolejnej strony wyświetlanej
w przeglądarce użytkownika, ale także kod
określający, czy należy wygenerować ostateczną
poradę czy może następne pytanie.

*Model (logika biznesowa)
musi określić, czy
dysponuje już wszystkimi
informacjami potrzebnymi
do wygenerowania porady
(a więc odesłania klientowi
ostatecznej odpowiedzi);
jeśli nie, niezbędne jest
odesłanie do klienta
kolejnego pytania, na
które użytkownik musi
odpowiedzieć.*

Należy zapewnić środowisko maksymalnie zbliżone do RZECZYWISTEJ konwersacji...

Potrzebujemy na imprezę jakichś lepszych napojów. Zadzwoń do Kima...



Cześć Stary, jestem na plażowej imprezie u Joego i trzymam właśnie w dłoni jakiś mętny czerwony drink z parasolką... może masz tam u siebie jakieś piwko?



Drinki z parasolkami? Fuuuuuj, to musi być naprawdę PASKUDNE. Dobrze, że zadzwoniłeś... pozwól jednak, że zadam Ci kilka pytań – po pierwsze, chcesz coś ciemnego, bursztynowego czy może brązowego?



No cóż, sam lubię ciemne... ale mam tutaj mnóstwo rozweselonych gości, więc chyba bezpieczniej będzie podać bursztynowe.



Hmmm... mam mnóstwo różnych gatunków piwa bursztynowego... masz jakieś preferencje odnośnie ceny?



Stary... czy pracowałbym jako model fotograficzny do książki informatycznej, gdybym nie potrzebował pieniędzy? OCZYWIŚCIE, że interesuje mnie cena piwa.



Nie ma sprawy... mam kilka przygotowanych skrzynek, które mogę Ci zaraz wystać.



Jak można śledzić odpowiedzi klienta?

Projekt Kima nie będzie prawidłowo działał, dopóki nie skonstruuje odpowiedniego mechanizmu śledzenia *wszystkiego*, co klient zdążył powiedzieć w czasie konwersacji (a więc nie tylko jego odpowiedzi zawartej w *bieżącym* żądaniu). Serwlet Kima musi mieć nie tylko dostęp do parametrów żądania reprezentujących opcje wybierane przez klienta, ale także możliwość ich zapisywania w swoich strukturach danych. Za każdym razem, gdy klient odpowiada na pytanie, moduł generowania porad wykorzystuje *wszystkie* wcześniejsze odpowiedzi klienta albo do wygenerowania *kolejnego* pytania, albo do stworzenia ostatecznej rekomendacji.

Jakie rozwiązania mamy do dyspozycji?

Użycie stanowego, sesyjnego komponentu Enterprise JavaBeans



Jasne, zawsze można to zrobić. Kim może przekształcić swój serwlet w klienta stanowego komponentu sesyjnego, a za każdym razem, gdy do tak zmodyfikowanego serwletu dotrze żądanie, można lokalizować stany komponent właściwy dla danego klienta. W takim przypadku Kim będzie co prawda musiał stawić czoła wielu bardziej lub mniej skomplikowanym utrudnieniom, ale z pewnością zastosowanie stanowego komponentu sesyjnego do przechowywania stanu konwersacyjnego jest rozwiązaniem wartym rozważenia.

Ten sposób wiąże się jednak ze *znacznym* nakładem sił, szczególnie w przypadku tak prostej aplikacji! Poza tym usługodawca obsługujący witrynę internetową Kima nie dysponuje serwerem, który byłby w pełni zgodny ze standardem J2EE i udostępniałby niezbędny w tym przypadku kontener EJB. Firma współpracująca z Kimem ma jedynie Tomcata (kontener WWW).

Użycie bazy danych



Także to rozwiązanie może się sprawdzić w praktyce. Jak się okazało, firma obsługująca witrynę Kima *udostępnia* serwer bazy danych MySQL, zatem jej wykorzystanie do składowania stanu konwersacyjnego jest możliwe. Serwer może zapisywać dane klienta w bazie danych... jednak takie rozwiązanie będzie miało równie negatywny (lub nawet gorszy) wpływ na wydajność aplikacji internetowej jak stosowanie komponentów EJB. W przypadku prostej aplikacji Kima używanie bazy danych jest przesadą.

Użycie obiektu HttpSession



Z pewnością od początku domyślałeś się, że rozwiązanie oparte na obiekcie HttpSession jest najwłaściwsze. Można w prosty sposób użyć tego obiektu do składowania stanu konwersacyjnego obejmującego wiele żądań. Innymi słowy, obiekt HttpSession może służyć do przechowywania całej *sesji* danego klienta.

(W praktyce Kim i tak musiałby używać obiektu HttpSession, nawet gdyby wybrał inną opcję, np. opartą na bazie danych lub komponencie sesyjnym, ponieważ w sytuacji, gdy funkcję klienta pełni przeglądarka internetowa, musielibyśmy jakoś przyporządkowywać określonego klienta do odpowiedniego klucza w bazie danych lub identyfikatora komponentu sesyjnego — jak się niedługo przekonasz, obiekt HttpSession obsługuje taką identyfikację).

Obiekt HttpSession może przechowywać stan konwersacyjny obejmujący wiele żądań pochodzących od tego samego klienta.

Innymi słowy, taki obiekt istnieje dla całej sesji z określonym klientem.

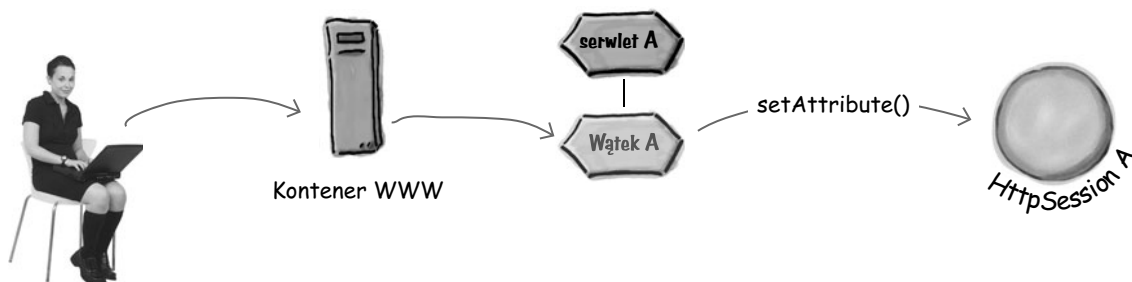
Można używać tego obiektu do przechowywania *wszystkiego*, co otrzymaliśmy od klienta we *wszystkich* żądaniach, które ten klient wygenerował w czasie trwania sesji.

Jak działa sesja

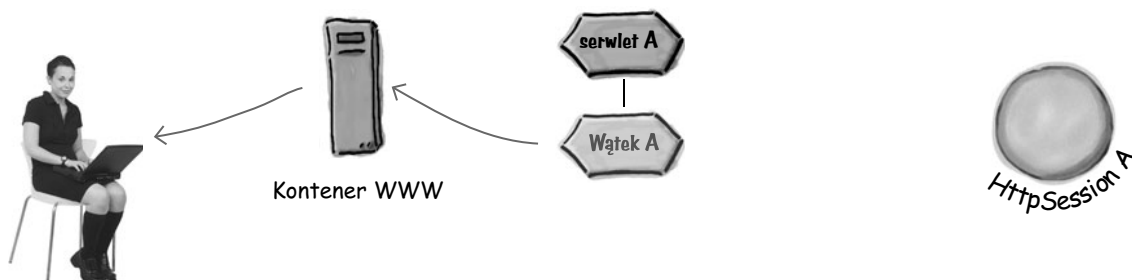
- ❶ Diane wybiera "Ciemne" i klika przycisk Prześlij kwerendę.

Kontener wysyła żądanie do nowego wątku serwletu AplikacjaPiwna.

Wątek serwletu AplikacjaPiwna znajduje sesję powiązaną z Diane i zapisuje jej wybór ("Ciemne") w postaci atrybutu sesji.



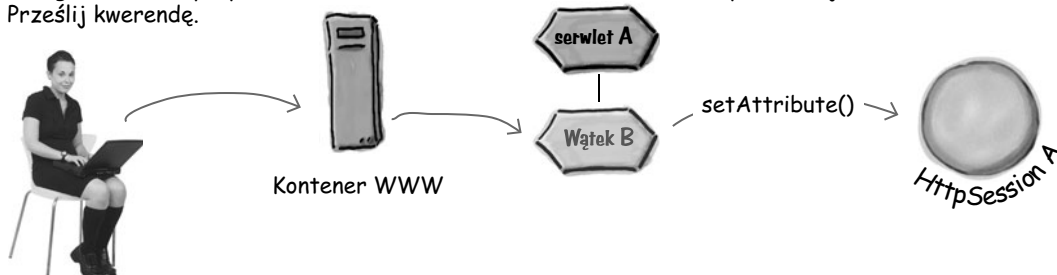
- ❷ Serwlet wykonuje swoją logikę biznesową (włącznie z wywołaniami odpowiedniej funkcjonalności modelu) i zwraca odpowiedź... w tym przypadku kolejne pytanie o dopuszczalny zakres cenowy.



- ❸ Diane zastanawia się chwilę nad odpowiedzią, by ostatecznie wybrać opcję "Drogie" i kliknąć przycisk Prześlij kwerendę.

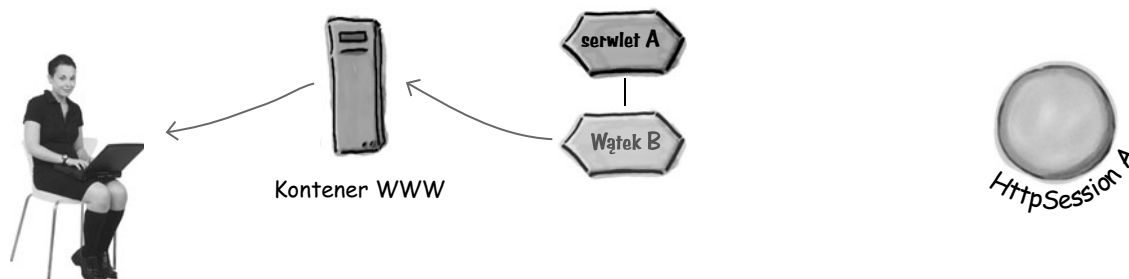
Kontener przekazuje żądanie nowemu wątkowi serwletu AplikacjaPiwna.

Wątek serwletu AplikacjaPiwna odnajduje sesję powiązaną z Diane i zapisuje jej najnowszy wybór (opcję "Drogie") w postaci atrybutu sesji.



Ten sam klient
Ten sam serwlet
Inne żądanie
Inny wątek
Ta sama sesja

- 4 Serwlet wykonuje swoją logikę biznesową (włącznie z wywołaniami funkcjonalności modelu) i zwraca odpowiedź... w tym przypadku kolejne pytanie.

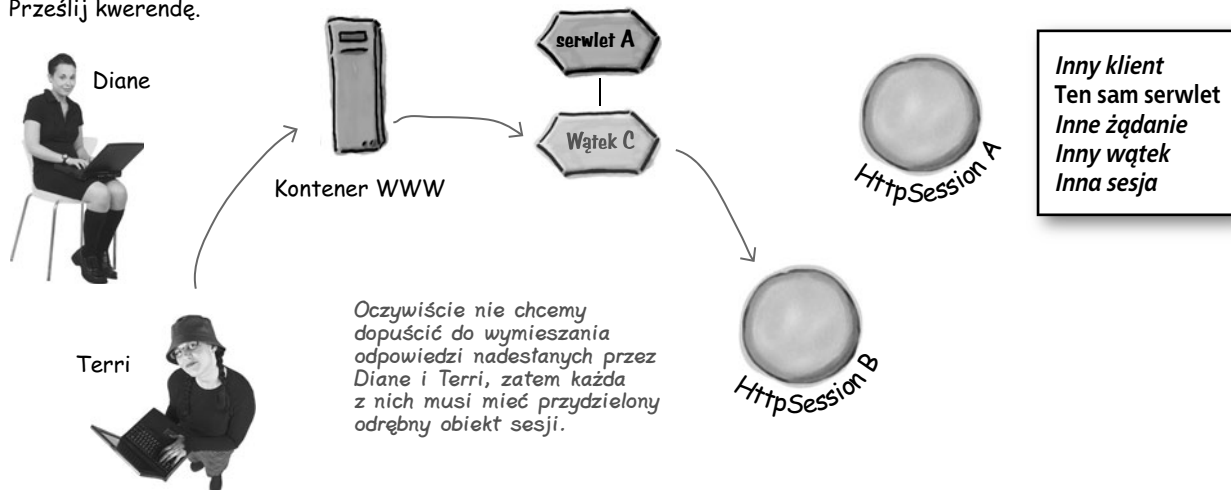


Wyobraź sobie, że w międzyczasie INNY klient wchodzi na witrynę porad piwnych...

- 5 Sesja Diane wciąż jest aktywna, ale w międzyczasie Terri wybiera opcję "Brązowe" i klika przycisk Prześlij kwerendę.

Kontener przekazuje żądanie Terri nowemu wątkowi serwletu AplikacjaPiwna.

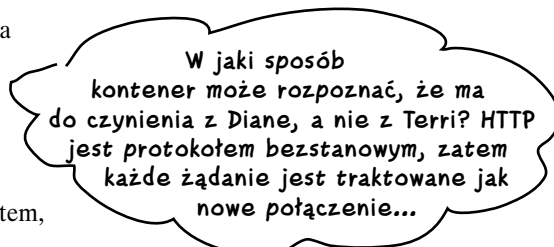
Wątek serwletu AplikacjaPiwna tworzy nową sesję dla Terri i wywołuje metodę `setAttribute()`, aby umieścić w nowym obiekcie sesji wybór Terri (opcję "Brązowe").



Kolejny problem... skąd kontener wie, kim jest dany klient?

Protokół HTTP wykorzystuje tzw. połączenia *bezstanowe*. Przeglądarka klienta tworzy połączenie z serwerem, przesyła żądanie, otrzymuje odpowiedź i zamyka połączenie. Innymi słowy, połączenie protokołu HTTP istnieje tylko dla *pojedynczej* pary żądanie-odpowiedź.

Ponieważ połączenia są nietrwałe, kontener nie ma możliwości stwierdzenia, że dany klient przysyłający żądanie jest tym samym klientem, który przysłał poprzednie żądanie. W efekcie na poziomie kontenera **każde żądanie jest traktowane tak, jakby pochodziło od nowego klienta.**



Nie ma
niemądrych pytań

P: Dlaczego kontener nie może po prostu użyć adresu IP klienta? Przecież adres ten jest częścią żądania, prawda?

U: To fakt, kontener może uzyskać adres IP klienta, który wygenerował żądanie, ale czy na pewno można w ten sposób jednoznacznie identyfikować klientów? Jeśli pracujesz w lokalnej sieci IP, dysponujesz co prawda unikatowym adresem IP, jednak wcale nie musi to oznaczać, że adres ten jest widoczny dla komputerów działających poza Twoją siecią. Z perspektywy serwera Twoim adresem IP jest adres routera, zatem w praktyce masz identyczny adres jak wszystkie pozostałe komputery w Twojej sieci! Oznacza to, że takie posunięcie nie rozwiązuje naszego problemu. Wciąż będzie możliwa sytuacja, w której artykuły umieszczone przez Jima w *jego* koszyku z zakupami znajdą się w koszyku kogoś zupełnie innego i vice versa. Zatem odpowiedź brzmi: nie, adres IP nie jest rozwiązaniem zapewniającym jednoznaczne identyfikowanie określonego klienta w internecie.

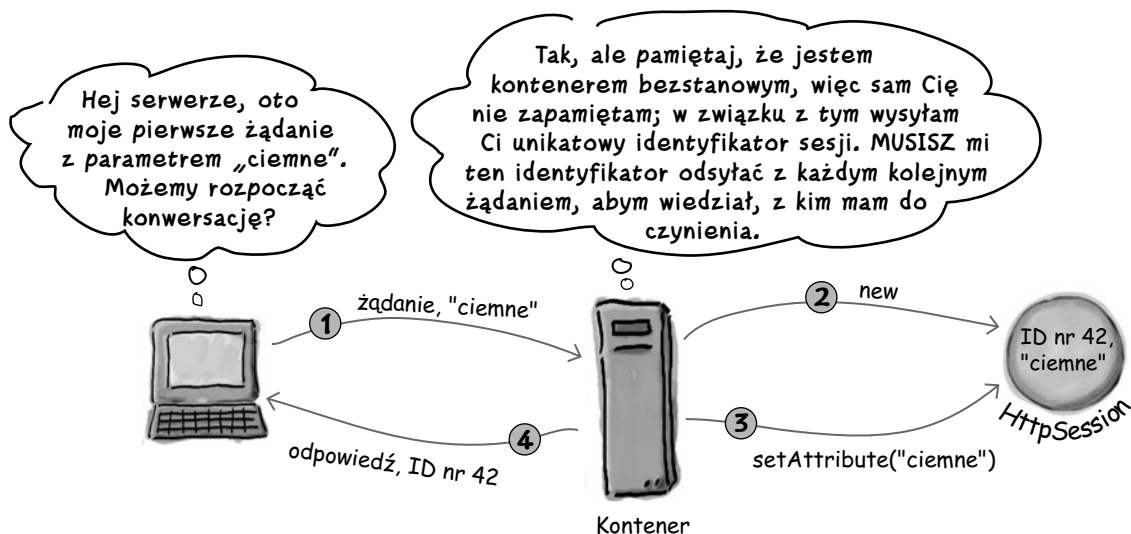
P: A co dzieje się z informacjami decydującymi o bezpieczeństwie aplikacji? Jeśli użytkownik jest zalogowany i stosuje się bezpieczne połączenie (HTTPS), kontener powinien **DOKŁADNIE** wiedzieć, kto jest klientem, prawda?

U: Tak, jeśli użytkownik jest zalogowany i korzysta z bezpiecznego połączenia, kontener może identyfikować właściwie klienta i wiązać go z odpowiednią sesją. To bardzo **ważny warunek**. Zalecenia odnośnie projektowania dobrych witryn internetowych często wspominają, że „nie należy zmuszać użytkownika do logowania, a sama aplikacja nie powinna się przełączać na komunikację za pomocą bezpiecznego protokołu HTTPS aż do momentu, w którym jest to absolutnie konieczne”. Jeśli użytkownicy Twojej aplikacji jedynie przeglądają ogólnodostępne strony internetowe, nawet jeśli dodają oferowane towary do koszyka z zakupami, prawdopodobnie nie ma jeszcze powodu, aby tracić czas (Twój i użytkownika) na zmuszanie ich do uwierzytelniania w systemie — przynajmniej do momentu, w którym użytkownicy podejmą decyzję o dokonaniu przelewu. Oznacza to, że potrzebujemy takiego mechanizmu wiążącego klienta z sesją, który nie będzie wymagał bezpiecznego uwierzytelniania (szczegóły związane z bezpieczeństwem aplikacji internetowych... cierpliwości... omówimy w rozdziale poświęconym bezpieczeństwu).

Klient potrzebuje unikatowego identyfikatora sesji

Idea jest prosta: po otrzymaniu pierwszego żądania danego klienta kontener generuje unikatowy identyfikator sesji i zwraca go klientowi wraz z odpowiedzią. **Klient odsyła otrzymany identyfikator sesji z każdym kolejnym żądaniem.**

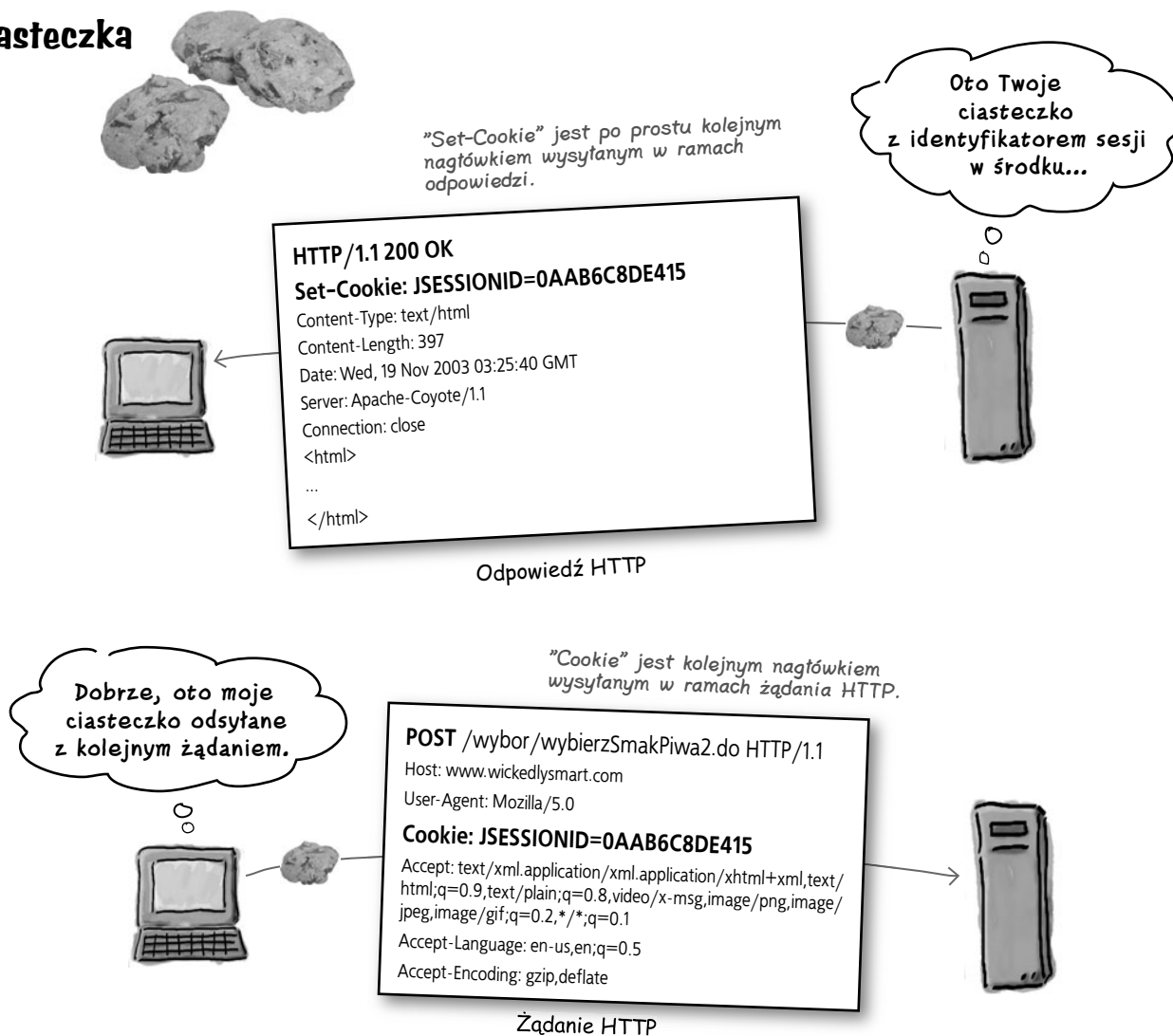
Kontener, widząc taki identyfikator w żądaniu, odnajduje odpowiednią sesję i wiąże ją z tym żądaniem.



Jak w praktyce klient i kontener wymieniają między sobą informacje o identyfikatorze sesji?

Kontener musi w jakiś sposób przekazać klientowi identyfikator sesji w postaci odpowiedniego fragmentu odpowiedzi, natomiast klient musi ten identyfikator odsyłać w postaci fragmentów swoich kolejnych żądań. Najprostszym i najbardziej popularnym sposobem wymiany tego rodzaju identyfikacji są tzw. *ciasteczka* (nazywane też znacznikami kontekstu klienta).

Ciasteczka



Najważniejsza zaleta tego rozwiązania: kontener realizuje niemal wszystkie zadania związane z obsługą ciasteczek!

Musisz zasygnalizować kontenerowi, że chcesz stworzyć lub wykorzystać sesję, ale już za generowanie identyfikatora tej sesji, tworzenie obiektu nowego znacznika kontekstu klienta (ciasteczka), umieszczanie w tym ciasteczku identyfikatora sesji i przesłanie go w ramach odpowiedzi będzie odpowiadał kontener. W przypadku kolejnych żądań pochodzących od tego samego klienta kontener odczytuje identyfikator sesji z dołączonego ciasteczka, dopasowuje ten identyfikator do istniejącej sesji i wiąże tę sesję z bieżącym żądaniem.

Przesyłanie ciasteczka klienta z identyfikatorem sesji w ramach ODPOWIEDZI:

```
HttpSession session = request.getSession();
```

To wszystko. Gdzieś w Twojej metodzie obsługującej żądania prosisz o sesję, a wszystkie pozostałe zadania z tym związane są realizowane *automatycznie*.

Nie tworzysz samodzielnie obiektu HttpSession.

Nie generujesz unikatowego identyfikatora sesji.

Nie tworzysz nowego obiektu ciasteczka.

Nie wiążesz identyfikatora sesji z ciasteczkiem klienta.

Nie dołączasz obiektu ciasteczka do odpowiedzi (w ramach nagłówka Set-Cookie).

Wszystkie działania związane z obsługą ciasteczek klienta są wykonywane w tle.

Odczytywanie identyfikatora sesji z ŻĄDANIA:

```
HttpSession session = request.getSession();
```

Wygląda znajomo, prawda? Tak, to dokładnie takie samo wywołanie jak to, którego użyliśmy do generowania identyfikatora sesji i ciasteczka na potrzeby generowanej odpowiedzi!

IF (żądanie zawiera ciasteczko klienta z identyfikatorem sesji)

znajdź sesję odpowiadającą temu identyfikatorowi;

ELSE IF (nie istnieje ciasteczko klienta z identyfikatorem sesji LUB nie istnieje bieżąca sesja pasująca do otrzymanego identyfikatora)

stwórz nową sesję.

Wszystkie działania związane z obsługą ciasteczek są wykonywane w tle.

Kiedy poprosisz obiekt reprezentujący żądanie o sesję, kontener automatycznie zrealizuje wszystkie niezbędne działania. Nie musisz już wykonywać żadnych dodatkowych kroków!

(Użyta metoda nie tylko tworzy sesję, ale też — kiedy wywołasz ją po raz PIERWSZY — powoduje odesłanie klientowi ciasteczka wraz z odpowiedzią na bieżące żądanie. Nie mamy co prawda pewności, czy klient to ciasteczko ZAAKCEPTUJE... ale przynajmniej mamy świadomość należytego wykonania swoich zadań na odcinku serwera).

Nie do wiary! Metoda ODCZYTUJĄCA znacznik kontekstu (ciasteczko) klienta z identyfikatorem sesji (i dopasowująca ten identyfikator do istniejącej sesji) jest taka sama jak metoda WYSYŁAJĄCA ciasteczko do klienta. W praktyce nie będziesz nawet WIDZIAŁ tego identyfikatora (choć możesz poprosić obiekt sesji o jego zwrócenie).

Co należy zrobić, jeśli chcemy się dowiedzieć, czy dana sesja istniała wcześniej, czy też została właśnie utworzona?

Dobre pytanie. Bezargumentowa metoda `getSession()` obiektu żądania zwraca sesję *niezależnie od tego, czy odpowiednia sesja już istniała*. Ponieważ metoda `getSession()` zawsze zwraca obiekt typu `HttpSession`, jedynym sposobem sprawdzenia, czy dana sesja jest nowa, jest **odwołanie się do samej sesji**.

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException {
```

```
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("test atrybutów sesji<br>");
```

Metoda `getSession()` zwraca sesję niezależnie od tego, czy... nie możesz jednak określić, czy jest to nowa sesja, dopóki nie zapytasz o to otrzymanego obiektu sesji.

```
    HttpSession session = request.getSession();
```

```
    if (session.isNew()) {
        out.println("To jest nowa sesja");
    } else {
        out.println("Witamy ponownie!");
    }
}
```

Metoda `isNew()` zwraca wartość `true`, jeśli klient nie odesłał jeszcze kolejnego żądania z danym identyfikatorem sesji.

P: Obiekt sesji otrzymujemy przez wywołanie metody `request.getSession()`, ale czy nie ma innego sposobu uzyskania sesji? Czy nie możemy do tego celu użyć np. obiektu `ServletContext`?

U: Obiekt sesji otrzymujemy z obiektu żądania, ponieważ — dobrze się nad tym zastanów — sesja jest identyfikowana właśnie przez żądanie. Kiedy wywołujemy metodę `getSession()`, w praktyce zwracamy się do kontenera z następującą prośbą: „Interesuje nas sesja dla TEGO klienta... może to być albo sesja już istniejąca, pasująca do przysłanego przez klienta identyfikatora, albo zupełnie nowa sesja. W obu przypadkach *sesja jest jednak związana z klientem, który przysłał to żądanie*”.

Istnieje także inny sposób uzyskiwania obiektu sesji... za pośrednictwem obiektu zdarzenia związanego z sesją. Zapewne pamiętasz, że klasa nasłuchująca nie jest ani serwiletem, ani tym bardziej stroną JSP — jest to zwykła klasa, która chce być informowana o pewnych zdarzeniach. Na przykład klasa nasłuchująca dla atrybutów może reagować na takie zdarzenia jak dodanie atrybutu (lub obiektu reprezentującego atrybut) do sesji lub usunięcie go z niej.

Metody obsługi zdarzeń zadeklarowane w interfejsach nasłuchujących związanych z sesjami otrzymują w formie argumentu obiekt klasy `HttpSessionEvent` lub jej podklasy `HttpSessionBindingEvent`, a metoda `getSession()` jest udostępniana właśnie przez klasę `HttpSessionEvent`.

Jeśli więc zaimplementujemy którykolwiek z czterech interfejsów nasłuchujących związanych z sesjami (omówimy je w dalszej części tego rozdziału), będziemy mogli uzyskiwać dostęp do sesji za pośrednictwem metod zwrotnych, których zadaniem jest obsługa odpowiednich zdarzeń. Przykładowo, niniejszy fragment kodu pochodzi z klasy implementującej interfejs `HttpSessionListener`:

```
public void sessionCreated(HttpSessionEvent event) {
    HttpSession session = event.getSession();
    // kod obsługujący zdarzenie
}
```

Co należy zrobić, aby uzyskać dostęp TYLKO do już istniejącej sesji?

Możesz się znaleźć w sytuacji, w której serwlet będzie chciał korzystać wyłącznie z obiektu wcześniej utworzonej sesji. W serwlecie odpowiedzialnym na przykład za dokonywanie przelewów tworzenie *nowej* sesji może nie mieć uzasadnienia (lub może nawet budzić pewne podejrzenia).

Z myślą o tego typu sytuacjach przygotowano przeciążoną metodę `getSession(boolean)`. Jeśli nie chcesz tworzyć nowej sesji, powinieneś wywołać metodę `getSession(false)`, która zwróci albo `null`, albo już istniejący obiekt `HttpSession`.

W poniższym fragmencie kodu wywołujemy metodę `getSession(false)`, po czym sprawdzamy, czy zwrócona wartość jest równa `null`. Jeśli *tak*, przekazujemy na standardowe wyjście odpowiedni komunikat i tworzymy nową sesję.

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException {
```

```
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("test sesji<br>");

    HttpSession session = request.getSession(false);

    if (session==null) {
        out.println("Żadna sesja nie była dostępna");
        out.println("tworzenie sesji...");
        session = request.getSession();
    } else {
        out.println("Odpowiednia sesja już istniała!");
    }
}
```

Przekazanie wartości `false` oznacza, że metoda `getSession()` zwróci albo już istniejącą sesję, albo wartość `null` (jeśli jeszcze nie istnieje sesja powiązana z danym klientem).

Możemy teraz sprawdzić, czy taka sesja już istniała (bezargumentowa wersja metody `getSession()` NIGDY nie zwraca wartości `null`).

Teraz już WIEMY, że tworzona jest nowa sesja.

P: Czy powyższy kod nie jest bezsensownym, nieefektywnym sposobem realizacji tego samego zadania, które z powodzeniem wykonywaliśmy na poprzedniej stronie? Przecież w końcu i tak tworzona jest nowa sesja.

U: Masz rację. Powyższy fragment kodu ma na celu jedynie testowanie działania dwóch różnych wersji metody `getSession()`. W praktyce wywołanie metody `getSession(false)` stosuje się wyłącznie w sytuacjach, gdy NIE chcemy tworzyć nowej sesji. Jeśli naszym celem jest stworzenie nowej sesji, ale nadal chcemy reagować w inny sposób na wygenerowanie nowej sesji (w odróżnieniu od uzyskania dostępu do sesji już istniejącej), powinniśmy użyć bezargumentowej metody `getSession()` i dopiero potem wywołać metodę `isNew()` otrzymanego obiektu `HttpSession`, aby sprawdzić, czy mamy do czynienia z nową, czy już istniejącą sesją.

P: Wygląda więc na to, że metoda `getSession(true)` niczym się nie różni od metody `getSession()`...

U: Znowu masz rację. Bezargumentowa wersja tej metody została opracowana wyłącznie z myślą o wygodzie programistów, którzy w każdych okolicznościach chcą otrzymywać obiekty sesji, nowe lub istniejące. Wersja, która otrzymuje na wejściu wartość logiczną, jest przydatna tylko wtedy, gdy wiemy, że nie interesuje nas nowa sesja, lub kiedy decyzja odnośnie tworzenia nowej sesji zapada w czasie wykonywania aplikacji (wówczas do metody `getSession(jakaśWartośćLogiczna)` przekazujemy zmienną logiczną).

Znakomicie... brzmi wspaniale, ale... INFORMACJA Z OSTATNIEJ CHWILI: każdy, kto ma głowę na karku, wyłącza obsługę ciasteczek w swojej przeglądarce. Jak w takim razie stosować sesje bez ciasteczek?



Możesz stosować sesje nawet wtedy, gdy klient nie akceptuje ciasteczek, ale takie rozwiązanie wymaga trochę więcej zachodu...

Nie możemy się zgodzić z tezą, że każdy świadomy użytkownik wyłącza obsługę ciasteczek. W praktyce większość przeglądarek *akceptuje* te znaczniki i wszystko działa znakomicie. **Nie mamy jednak takiej gwarancji.**

Jeśli prawidłowe funkcjonowanie Twojej aplikacji *zależy* od właściwej obsługi sesji, musisz znaleźć inny sposób wymiany informacji o identyfikatorze sesji pomiędzy klientem a kontenerem. Masz szczęście — kontener może się komunikować z klientem, który nie obsługuje ciasteczek, ale obejście tego ograniczenia wymaga od Ciebie nieco większego zaangażowania.

Jeśli wykorzystamy przedstawiony na poprzedniej stronie kod obsługujący sesję (a więc użyte tam wywołanie metody `getSession()` obiektu żądania), kontener spróbuje zastosować właśnie znaczniki kontekstu klienta (ciasteczka). Jeśli okaże się, że obsługa tych znaczników po stronie klienta jest wyłączona, będzie jasne, że klient nigdy nie dołączy do istniejącej sesji. Innymi słowy, **metoda `isNew()` tej sesji zawsze będzie zwracała wartość `true`.**



Oglądaj to!

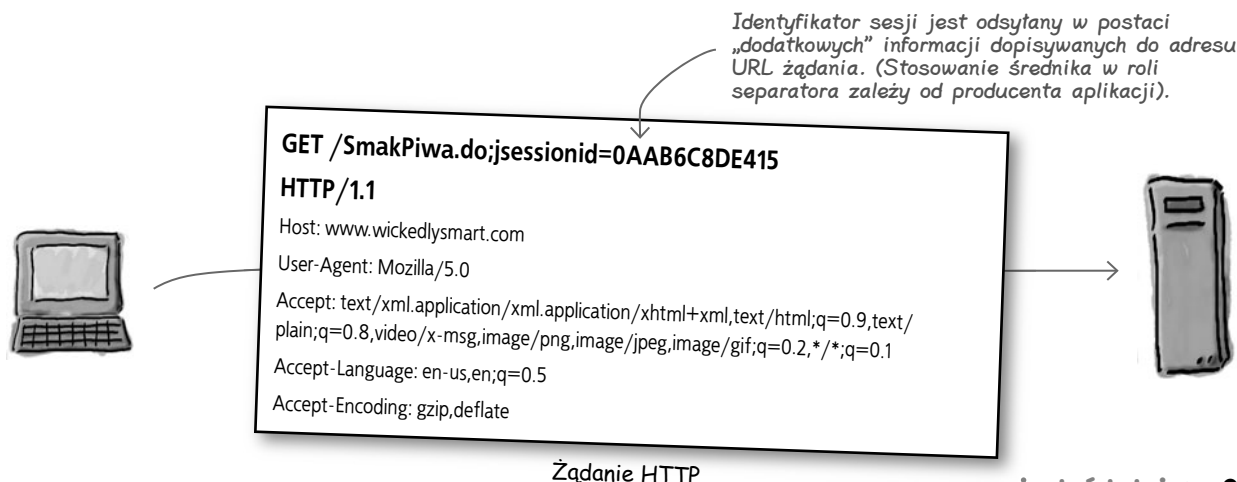
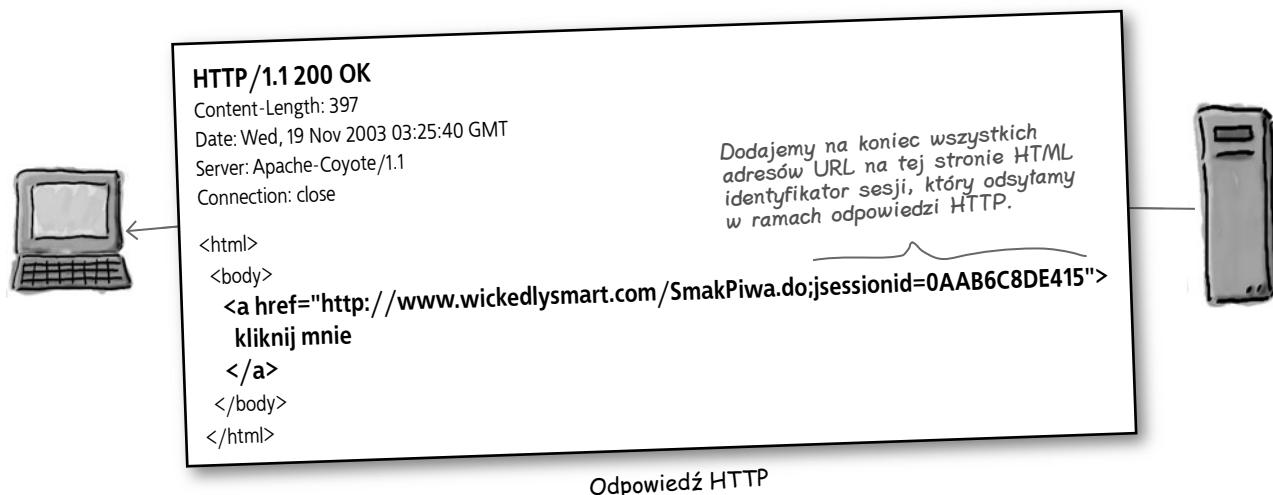
Klient z wyłączoną obsługą ciasteczek będzie ignorował nagłówki odpowiedzi "Set-Cookie".

Jeśli klient nie akceptuje ciasteczek, nie otrzymamy żadnego wyjątku. Nie będzie syren i dzwonów alarmujących o nieudanej próbie utworzenia sesji dla danego klienta. Nie, Twoje zabiegi na rzecz ustawienia znacznika z identyfikatorem sesji zostaną najzwyczajniej w świecie zignorowane. Jeśli NIE użyjesz w swoim kodzie mechanizmu przepisywania adresów URL, metoda `getSession()` zawsze będzie zwracała NOWĄ sesję (tj. taki obiekt sesji, którego metoda `isNew()` zawsze będzie zwracała wartość `true`). Klient po prostu nigdy nie odeśle żądania z dołączonym nagłówkiem ciasteczka i zawartym tam identyfikatorem sesji.

Przepisywanie adresów URL: jedno z możliwych rozwiązań naszego problemu

Jeśli klient nie akceptuje ciasteczek, możemy posilkować się mechanizmem przepisywania adresów URL. Zakładając, że właściwie przygotujesz kod swojej aplikacji internetowej, przepisywanie adresów URL *zawsze* będzie spełniało swoje zadanie — klient nie będzie reagował na nasze działania i w żaden sposób nie powinien im zapobiegać. Naszym celem pozostaje wymiana informacji o identyfikatorze sesji pomiędzy klientem a kontenerem. *Najprostszym* sposobem wymiany identyfikatorów sesji zawsze jest przekazywanie ciasteczek (znaczników kontekstu klienta). Co jednak powinniśmy zrobić, jeśli nie mamy możliwości umieszczenia tego identyfikatora w ciasteczku? Technika przepisywania adresów URL polega na pobraniu ze znacznika kontekstu klienta identyfikatora sesji i umieszczenia go na samym końcu każdego adresu URL trafiającego do danej aplikacji.

URL + ;jsessionid=1234567



Przepisywanie adresów URL jest uzasadnione TYLKO wtedy, gdy stosowanie ciasteczek jest niemożliwe, i TYLKO jeśli wymuszamy kodowanie adresów URL w odpowiedziach

Jeśli ciasteczka nie zdają egzaminu, kontener skorzysta z mechanizmu przepisywania adresów URL, ale *tylko* wtedy, gdy odpowiednio przygotujesz swoją aplikację do kodowania adresów URL we wszystkich wysyłanych odpowiedziach. Jeśli chcesz, aby kontener zawsze stosował w pierwszej kolejności ciasteczka i dopiero potem (w razie niepowodzenia) posiłkował się techniką przepisywania adresów URL, możesz być spokojny. Właśnie tak będzie działała nasza aplikacja (choć nie od razu — pożądany efekt uzyskamy dopiero za chwilę). Jeśli jednak **wprost nie wymusisz kodowania adresów URL** i okaże się, że klient nie akceptuje znaczników kontekstu klienta, **stracisz możliwość stosowania sesji**. Jeśli natomiast będziesz kodował adresy URL, kontener w pierwszej kolejności podejmie próbę użycia ciasteczek do zarządzania sesjami, a z możliwości przepisywania adresów URL skorzysta dopiero wtedy, gdy rozwiązanie oparte na znacznikach kontekstu okaże się nieskuteczne.

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException {

    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    HttpSession session = request.getSession();    zwraca sesję

    out.println("<html><body>");
    out.println("<a href=\"\" + response.encodeURL(\"/TestPiwa.do\") + \"\">kliknij mnie</a>");
    out.println("</body></html>");
}
```

Dotacza do tego adresu URL dodatkowe informacje o identyfikatorze sesji.

P: Zaczekaj chwilę... **SKĄD** kontener wie, że ciasteczka nie działają? W którym momencie kontener decyduje o zastosowaniu mechanizmu przepisywania adresów URL?

U: Kiepskich kontenerów w ogóle nie interesuje, czy ciasteczka zdały egzamin, czy nie — takie kontenery zawsze będą podejmowały próby zarówno wysyłania ciasteczek, JAK I przepisywania adresów URL, nawet wtedy, gdy okaże się, że same ciasteczka w zupełności wystarczą. Warto jednak przeanalizować rozwiązanie stosowane w tej kwestii przez dobre kontenery.

Kiedy kontener widzi wywołanie metody `getSession()` i nie znajduje identyfikatora sesji w żądaniu klienta, wie, że musi podjąć próbę rozpoczęcia nowej sesji z danym klientem. Na tym etapie kontener nie ma oczywiście pojęcia, czy ciasteczka zadziałają, zatem w swojej pierwszej odpowiedzi odsyłanej danemu klientowi próbuje stosować OBIE techniki: znaczniki kontekstu klienta (ciasteczka) i przepisywanie adresów URL.

P: Dlaczego nie można najpierw wypróbować ciasteczka... i użyć techniki przepisywania adresów URL dopiero w następnej odpowiedzi, kiedy nie otrzymamy odpowiedzi z wysłanym wcześniej znacznikiem?

U: Pamiętaj, że jeśli kontener nie otrzyma od klienta identyfikatora sesji, nie może nawet WIEDZIEĆ, że ma do czynienia z *kolejnym* żądaniem od tego samego klienta. Kontener nie dysponuje żadnymi mechanizmami, które umożliwiałyby mu zdobywanie informacji na temat niepowodzenia próby przekazania klientowi identyfikatora sesji za pośrednictwem ciasteczka. Warto pamiętać, że JEDYNYM elementem umożliwiającym rozpoznanie klienta jest nadesłany przez niego identyfikator sesji!

Kiedy więc kontener widzi, że wywołujesz metodę `request.getSession()`, i zdaje sobie sprawę z konieczności rozpoczęcia nowej sesji z danym klientem, wysyła do tego klienta odpowiedź zawierającą zarówno nagłówek „Set-Cookie” z identyfikatorem sesji, *jak i* identyfikator sesji dołączony do adresów URL (zakładając, że użyliśmy w kodzie serwletu metody `response.encodeURL()`).

Wyobraź sobie teraz następne żądanie od tego samego klienta — żądanie to będzie zawierało identyfikator sesji dołączony do adresu URL oraz, jeśli klient akceptuje ciasteczka, DODATKOWO odpowiedni znacznik kontekstu klienta. Kiedy serwlet wywoła wówczas metodę `request.getSession()`, kontener odczyta z żądania identyfikator sesji, odnajdzie odpowiednią sesję i pomyśli: „Ten klient akceptuje ciasteczka, zatem mogę ignorować wszystkie kolejne wywołania metody `response.encodeURL()`. W bieżącej odpowiedzi odeślę klientowi tylko znacznik kontekstu, ponieważ wiem, że to wystarczy, i że nie ma potrzeby dalszego stosowania techniki przepisywania adresów URL, zatem nie będę więcej tracił na to czasu...”.

Przepisywanie adresów URL z użyciem metody `sendRedirect()`

Możesz się znaleźć w sytuacji, w której konieczne będzie przekierowanie żądania na inny adres URL, ale z jednoczesnym zachowaniem wykorzystywanej dotychczas sesji. Okazuje się, że istnieje specjalna metoda kodująca adres URL, która spełnia nasze oczekiwania:

```
response.encodeRedirectURL("/TestPiwa.do")
```

P: A co z moimi statycznymi stronami HTML... przecież pełno w nich łączy `<a href>`. Jak można stosować mechanizm przepisywania adresów URL dla stron statycznych?

U: W ogóle nie można! Jedynym rozwiązaniem jest dynamiczne generowanie WSZYSTKICH stron będących częścią sesji — tylko wówczas stosowanie techniki przepisywania adresów URL jest możliwe. Nie możesz przecież zapisywać identyfikatorów sesji w swoim kodzie na stałe, ponieważ takie identyfikatory po prostu nie istnieją do czasu właściwego wykonywania kodu. Jeśli więc prawidłowe funkcjonowanie Twojej aplikacji zależy od sesji, musisz stosować mechanizm przepisywania adresów URL w roli strategii zastępczej. A ponieważ nie można wykluczyć konieczności odwołania się do tej techniki, musisz dynamicznie generować adresy URL w kodzie HTML odpowiedzi! Oczywiście oznacza to, że musisz przetwarzać odpowiedni kod HTML w czasie pracy aplikacji.

Tak, stosowanie tego mechanizmu obniża efektywność aplikacji. W związku z tym powinieneś bardzo dokładnie przeanalizować nie tylko zasadność wykorzystywania sesji w swojej aplikacji, ale także precyzyjnie określić, czy mają krytyczne, czy może tylko drugorzędne znaczenie dla jej funkcjonowania.

P: Wspomniałeś, że aby stosowanie mechanizmu przepisywania adresów URL było możliwe, strony internetowe muszą być generowane dynamicznie. Czy to oznacza, że mogą do tego celu używać stron JSP?

U: Tak! Możesz stosować technikę przepisywania adresów URL w kodzie stron JSP, istnieje nawet stosunkowo prosty znacznik biblioteki JSTL (`<c:URL>`), który bardzo ten proces ułatwia; przekonasz się o tym podczas lektury rozdziału poświęconego stosowaniu znaczników niestandardowych.

P: Czy przepisywanie adresów URL jest obsługiwane w sposób zależny od producenta kontenera?

U: Tak, przepisywanie adresów URL jest obsługiwane w sposób wybrany przez producenta kontenera. Przykładowo, Tomcat wykorzystuje znak średnika w roli separatora oddzielającego *dodatkowe informacje* od właściwego adresu URL. I o ile Tomcat dodaje do przepisanego adresu URL łańcuch `"jsessionid="`, o tyle w rozwiązaniach oferowanych przez innych producentów może być dołączany sam identyfikator sesji. Kluczowe znaczenie ma fakt, iż kontener wysyłający w odpowiedzi swój znak separatora (jakikolwiek by on nie był) może ten znak bez trudu wykryć w kolejnym żądaniu. Kiedy więc kontener widzi separator, którego wcześniej *użył* (innymi słowy, kiedy odkrywa separator, który *sam* dodał podczas przepisywania adresu URL), od razu wie, że wszystko za tym znakiem stanowi umieszczone tam *"dodatkowe informacje"*. Inaczej mówiąc, kontener wie, jak rozpoznać i przetworzyć dodatkowe dane, które *sam* (kontener) dołączył do adresu URL.

Przepisywanie adresów URL odbywa się automatycznie... ale tylko wtedy, gdy kodujesz swoje adresy URL. **MUSISZ** przepuszczać wszystkie swoje adresy przez odpowiednią metodę obiektu odpowiedzi — `encodeURL()` lub `encodeRedirectURL()` — dopiero wówczas kontener podejmie pozostałe działania.



Oglądaj to!

Kodowanie adresów URL jest obsługiwane przez obiekt odpowiedzi!

Nie zapominaj, że wywoływana przez nas metoda `encodeURL()` należy do obiektu `HttpServletResponse`! Nie wywołujemy jej za pośrednictwem obiektu żądania, odpowiedzi czy sesji. Wystarczy pamiętać, że kodowanie adresów URL dotyczy wyłącznie generowanej odpowiedzi.



Oglądaj to!

Nie daj się zwieść parametrowi żądania "jsessionid" lub nagłówkowi "JSESSIONID".

NIGDY nie powinieneś samodzielnie stosować parametru "jsessionid". Jeśli gdziekolwiek w analizowanym kodzie źródłowym zobaczysz parametr żądania "jsessionid", będzie to oznaczało, że ktoś popełnił błąd. Nigdy, w żadnym kodzie nie powinieneś znaleźć następującego wiersza:

```
String sessionId = request.getParameter("jsessionid");
```

← Nie!!

Także w żądaniach i odpowiedziach nigdy nie powinieneś mieć do czynienia z niestandardowym nagłówkiem "JSESSIONID":

```
POST /wybor/wyberzSmakPiwa.do HTTP/1.1
User-Agent: Mozilla/5.0
JSESSIONID : 0AAB6C8DE415
```

Nigdy tak nie rób!
Taki nagłówek jest
nieprawidłowy!

W praktyce JEDYNYM miejscem, gdzie może występować słowo "JSESSIONID", jest nagłówek ciasteczka:

```
POST /wybor/wyberzSmakPiwa.do HTTP/1.1
User-Agent: Mozilla/5.0
Cookie: JSESSIONID=0AAB6C8DE415
```

To jest dobrze, ale
nie robimy tego
samodzielnie.

W niektórych przypadkach słowo "jsessionid" może się pojawiać także na końcu adresu URL (wówczas występuje w roli „dodatkowych informacji” dołączanych w ramach przepisywania adresu):

```
POST /wybor/wyberzSmakPiwa.do;jsessionid=0AAB6C8DE415
```

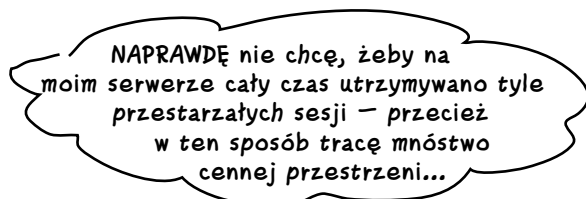
Tak może wyglądać efekt
przepisania adresu URL
(także w tym przypadku
odpowiedzialność za
przekształcenia spoczywa
na kontenerze).

KLUCZOWE ZAGADNIENIA



- Technika przepisywania adresów URL polega na dodawaniu identyfikatora sesji na koniec wszystkich adresów w kodzie HTML zapisywanym w obiekcie odpowiedzi.
- Identyfikator sesji jest następnie odsyłany wraz z żądaniem w postaci „dodatkowych” informacji, które są dopisywane na koniec adresu URL żądania.
- Mechanizm przepisywania adresów URL jest stosowany automatycznie w sytuacji, gdy znaczniki kontekstu klienta (ciasteczka) nie są obsługiwane przez klienta i — tym samym — nie spełniają swojej roli; musimy jednak wprost kodować wszystkie adresy URL zapisywane w odpowiedziach.
- Aby zakodować adres URL, należy wywołać metodę `response.encodeURL(jakiśłańcuch)`.

```
out.println("<a href='"  
+ response.encodeURL("/TestPiwa.do")  
+ "'>kliknij mnie</a>");
```
- Automatyczne stosowanie mechanizmu przepisywania adresów URL w kodzie statycznych stron HTML jest niemożliwe, zatem jeśli prawidłowe funkcjonowanie Twojej aplikacji zależy od sesji, musisz używać stron generowanych dynamicznie.



(Tak naprawdę chcę zwolnić przestrzeń na swoim komputerze tylko po to, by pograć w rozszerzenie Randka do gry The Sims).



Usuwanie sesji

Klient zgłasza swoje pierwsze żądanie, rozpoczyna sesję, zmienia zdanie i zrywa komunikację z naszą witryną internetową. Może się też zdarzyć, że pojawia się klient, rozpoczyna sesję i w tym momencie następuje nieoczekiwany błąd w jego przeglądarce. Klient może także przysłać żądanie, rozpocząć sesję i w naturalny sposób zakończyć ją przez prawidłowe dokonanie zakupów (przelewanie pieniędzy za towary umieszczone w koszyku). W dowolnym momencie może też nastąpić awaria komputera klienta. *To bez znaczenia.*

Zasadniczy problem polega na tym, że obiekty sesji zajmują zasoby. Nie chcemy, aby sesje były przechowywane dłużej, niż to konieczne. Pamiętaj, że protokół HTTP nie oferuje żadnych mechanizmów, które sygnalizowałyby serwerowi fakt zerwania komunikacji z klientem. (Stosując terminologię znaną tym z Was, którzy mają do czynienia ze środowiskami rozproszonymi — nie ma żadnego mechanizmu *dzierżawy*.)*

Skąd zatem kontener (lub *my*) ma *wiedzieć*, kiedy klient traci zainteresowanie dalszym komunikowaniem się z naszą aplikacją? Skąd kontener *wie*, kiedy przeglądarka klienta ulega awarii? **Skąd może wiedzieć, kiedy zniszczenie sesji jest bezpieczne?**

WYTEŻ UMYŚL

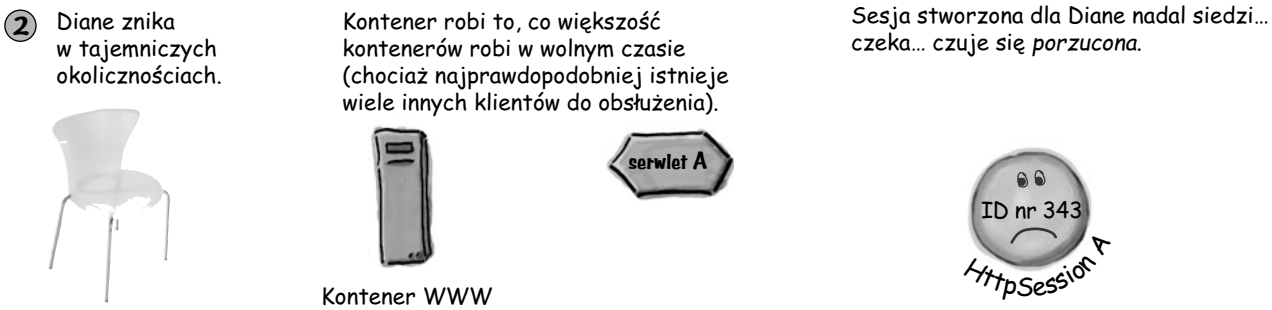
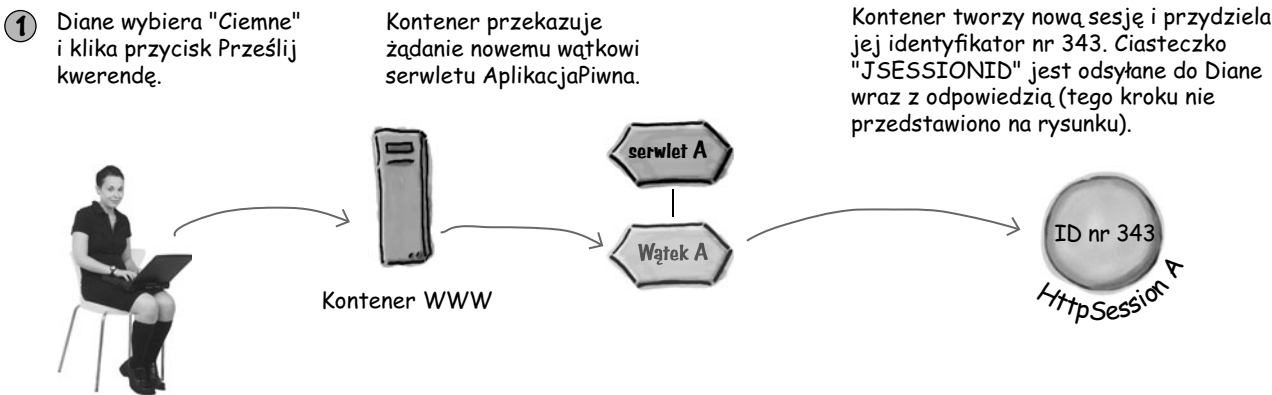
Jakich strategii możesz (Ty i Twój kontener) użyć do efektywnego zarządzania liczbą sesji i eliminowania niepotrzebnych sesji? Czy istnieją skuteczne sposoby sygnalizowania kontenerowi możliwości bezpiecznego usuwania niepotrzebnych sesji?

Przemyśl to, po czym przeanalizuj przedstawiony kilka stron dalej interfejs API HttpSession.

* Niektóre aplikacje rozproszone wykorzystują technikę *dzierżawy* właśnie do informowania serwera o zerwaniu komunikacji z klientem. Klient otrzymuje od serwera odpowiedni znacznik i musi go odnowić przed upływem określonego wcześniej limitu czasowego, aby potwierdzić swoje działanie. Jeśli czas ważności znacznika klienta się wyczerpie, serwer będzie wiedział, że może bezpiecznie zniszczyć wszelkie zasoby, które dla danego klienta przechowywał.

Jak chcielibyśmy, aby działało nasze rozwiązanie...

Chcielibyśmy, aby kontener wykrywał, kiedy sesja od zbyt długiego czasu jest nieaktywna, i aby tę sesję niszczył. Moglibyśmy się oczywiście spierać z kontenerem, co naprawdę oznacza określenie „zbyt długiego czasu”. Czy 20 minut to zbyt długo? Godzina? Dzień? (Być może istnieje nawet sposób określania w ustawieniach konfiguracyjnych kontenera odpowiadającego nam znaczenia wyrażenia „zbyt długiego czasu”).



Interfejs HttpSession

Kiedy wywołujemy metodę getSession(), interesuje nas wyłącznie uzyskanie obiektu klasy implementującej interfejs HttpSession. Przygotowanie i udostępnienie takiej klasy należy już do kontenera.

Co możemy zrobić z sesją, kiedy już dysponujemy odpowiednim obiektem?

Przez większość czasu będziemy używali sesji do odczytywania i ustawiania atrybutów należących do jej zasięgu.

Jednak możliwości tego interfejsu są oczywiście znacznie większe. Sprawdź, czy potrafisz opisać znaczenie i działanie kluczowych metod interfejsu HttpSession (odpowiedzi znajdziesz na następnej stronie, więc nie odwracaj kartki!).



```
<<interfejs>>
javax.servlet.http.HttpSession

Object getAttribute(String)
long getCreationTime()
String getId()
long getLastAccessedTime()
int getMaxInactiveInterval()
ServletContext getServletContext()
void invalidate()
boolean isNew()
void removeAttribute(String)
void setAttribute(String, Object)
void setMaxInactiveInterval(int)
// I WIELE innych metod...
```

	Co robi?	Do czego służy?
getCreationTime()		
getLastAccessedTime()		
setMaxInactiveInterval()		
getMaxInactiveInterval()		
invalidate()		

Kluczowe metody interfejsu HttpSession

Co prawda znasz już kilka metod operujących na atrybutach (`getAttribute()`, `setAttribute()` i `removeAttribute()`), ale poniżej przedstawiliśmy jeszcze kilka kluczowych metod interfejsu `HttpSession`, które mogą Ci się bardzo przydać podczas tworzenia Twoich aplikacji (a być może także na egzaminie).

	Co robi?	Do czego służy?
<i>getCreationTime()</i>	Zwraca czas pierwszego utworzenia danej sesji.	Do określania, jak długo istnieje dana sesja. W niektórych sytuacjach warto rozważyć wprowadzenie stałego ograniczenia czasu życia sesji. Możesz na przykład stwierdzić: „Po zalogowaniu masz dokładnie 10 minut na wypełnienie tego formularza...”.
<i>getLastAccessedTime()</i>	Zwraca czas (wyrażony w milisekundach) otrzymania przez kontener ostatniego żądania z danym identyfikatorem sesji.	Do określania, kiedy klient po raz ostatni korzystał z danej sesji. Możesz używać tej metody do podejmowania decyzji o wysyłaniu zapytań o dalsze zamiary do tych klientów, którzy od dłuższego czasu nie wygenerowali żadnych żądań. Jeśli nie otrzymasz odpowiedzi lub otrzymasz odpowiedź negatywną, możesz użyć metody <code>invalidate()</code> dla danej sesji.
<i>setMaxInactiveInterval()</i>	Określa maksymalny czas (wyrażony w sekundach) pomiędzy kolejnymi żądaniami klienta w ramach danej sesji.	Do wymuszania niszczenia sesji po upływie określonego czasu bez żądań ze strony klienta w ramach danej sesji. Jest to jeden ze sposobów ograniczania liczby starych sesji przechowywanych w pamięci serwera.
<i>getMaxInactiveInterval()</i>	Zwraca maksymalny czas (wyrażony w sekundach) pomiędzy kolejnymi żądaniami klienta w ramach danej sesji.	Do sprawdzania, jak długo dana sesja może być nieaktywna, ale wciąż żywa. Możesz używać tej metody w kodzie analizującym czas, jaki pozostał nieaktywnemu klientowi przed usunięciem danej sesji.
<i>invalidate()</i>	Kończy sesję. Koniec sesji obejmuje przerwanie wszelkich związków z aktualnie przechowywanymi atrybutami danej sesji (więcej informacji na ten temat znajdziesz w dalszej części tego rozdziału).	Do zabijania sesji w sytuacji, gdy klient jest od jakiegoś czasu nieaktywny lub gdy WIESZ, że dana sesja jest bezużyteczna (na przykład po przelaniu pieniędzy za zamówione towary lub po wylogowaniu). Sam egzemplarz sesji może być ponownie wykorzystywany przez kontener, ale nas już to nie interesuje. Koniec sesji oznacza, że przestaje istnieć jej identyfikator oraz że wszystkie atrybuty są usuwane z jej obiektu.



WYTEŻ UMYSŁ

Skoro zapoznałeś się już z tymi metodami, czy możesz opisać kompletną strategię eliminowania opuszczonych sesji?

Chyba nie mówisz poważnie... czy to oznacza, że muszę śledzić aktywność sesji, i że sama muszę niszczyć stare, nieaktywne sesje? Czy nie mógłby tego robić kontener?



Oglądaj to!

Definiowane w deskrytorze wdrożenia limity czasowe wyraża się w MINUTACH!

Jest to przejaw wyjątkowego braku konsekwencji, na który warto zwrócić uwagę... limity czasowe w deskrytorze wdrożenia definiujesz w MINUTACH, natomiast jeśli ustawiasz te same limity z poziomu kodu aplikacji, musisz je wyrażać w SEKUNDACH!

Ustawianie limitu czasowego dla sesji

Mam dla Ciebie dobrą wiadomość: *nie* musisz sama śledzić aktywności sesji. Widzisz metody na sąsiedniej stronie? Okazuje się, że wcale nie musisz ich używać do niszczenia i usuwania starych (nieaktywnych) sesji. Może to robić sam kontener.

Sesja może zginąć na trzy różne sposoby:

- po przekroczeniu ustalonego wcześniej limitu czasu,
- przez wywołanie metody `invalidate()` dla obiektu sesji,
- przez zakończenie pracy aplikacji (wskutek awarii lub celowego wyłączenia).

1 Konfigurowanie w deskrytorze wdrożenia limitu czasowego dla sesji

Konfigurowanie w deskrytorze wdrożenia limitu czasowego dla sesji daje niemal taki sam efekt jak wywoływanie metody `setMaxInactiveInterval()` dla każdej sesji tworzonej w kodzie naszej aplikacji.

```
<web-app ...>
  <servlet>
    ...
  </servlet>
  <session-config>
    <session-timeout>15</session-timeout>
  </session-config>
</web-app>
```

Limit czasowy (w tym przypadku "15") jest wyrażany w minutach. W ten sposób określamy, że jeśli przez 15 minut dany klient nie przysłał żadnego żądania w ramach danej sesji, sesja ta zostanie zniszczona.

2 Ustawianie limitu czasowego dla konkretnej sesji

Jeśli chcesz zmienić wartość limitu czasowego sesji dla konkretnego obiektu sesji (nie wpływając na długość tego limitu w pozostałych sesjach tej samej aplikacji internetowej), możesz użyć następującej metody:

```
session.setMaxInactiveInterval(20*60);
```

Metoda będzie miała wpływ tylko na tę sesję, dla której ją wywołasz.

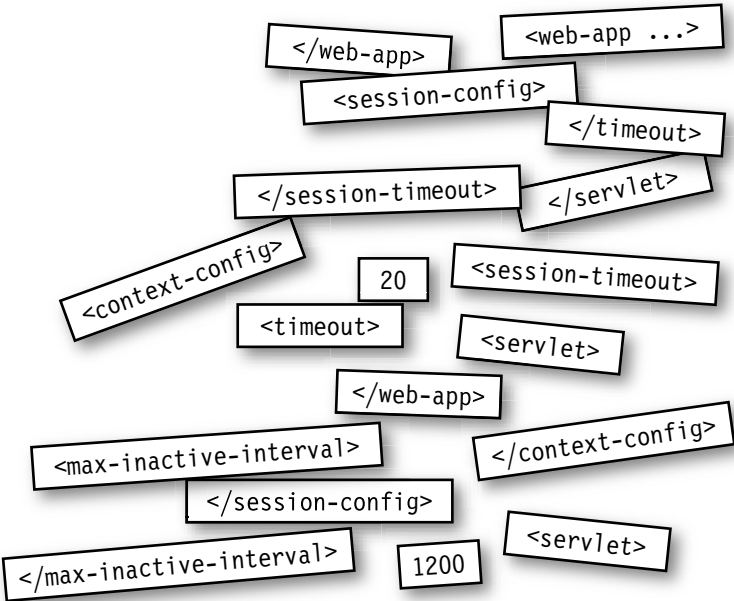
Argument tej metody jest czasem wyrażanym w sekundach, zatem w ten sposób określiliśmy, że sesja ma zostać zabita w sytuacji, gdy dany klient nie przysłał żadnego żądania przez 20 minut.



Magnesiki z kodem

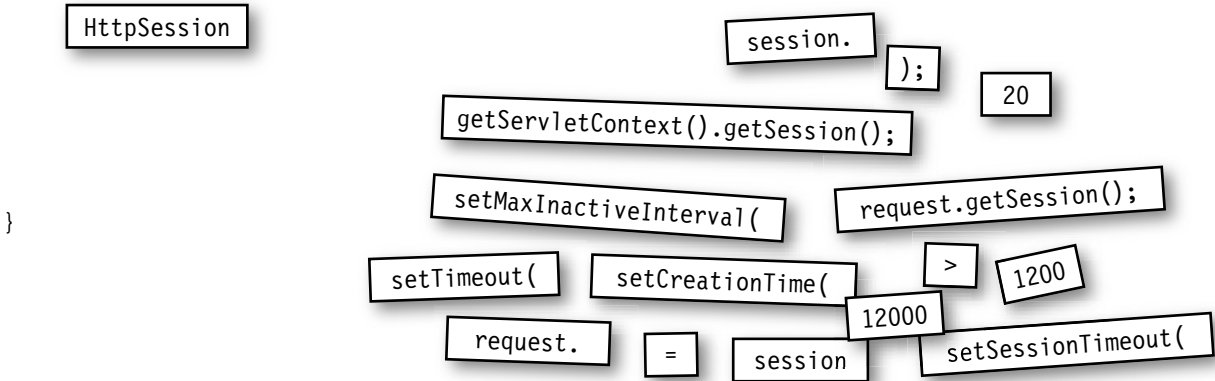
Spróbuj przygotować taki kod deskryptora wdrożenia i aplikacji internetowej, który będzie prawidłowo niszczył sesję w sytuacji, gdy dany klient nie przyśle żadnego żądania przez 20 minut. Dla ułatwienia w kodzie serwletu ustawiliśmy jeden magnes na właściwym miejscu; może się okazać, że nie wszystkie magnesy będą potrzebne do realizacji tego ćwiczenia.

— DD —



— Serwlet —

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws IOException {
```



BĄDŹ kontenerem



Oba poniższe listingi reprezentują fragment kodu klasy rozszerzającej klasę bazową `HttpServlet`, która pomyślnie przeszła proces kompilacji.

Twoim zadaniem jest wczucie się w rolę kontenera i określenie, co się stanie, kiedy każdy z tych serwetów zostanie wywołany dwukrotnie przez tego samego klienta. Dokładnie opisz przewidywane zachowanie tych serwetów podczas obsługi pierwszego i drugiego żądania tego samego klienta.

```
❶ public void doGet(HttpServletRequest request, HttpServletResponse response)
                                           throws IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    HttpSession session = request.getSession();
    session.setAttribute("foo", "42");
    session.setAttribute("bar", "420");
    session.invalidate();
    String foo = (String) session.getAttribute("foo");
    out.println("Foo: " + foo);
}
```

```
❷ public void doGet(HttpServletRequest request, HttpServletResponse response)
                                           throws IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    HttpSession session = request.getSession();
    session.setAttribute("foo", "42");
    session.setMaxInactiveInterval(0);
    String foo = (String) session.getAttribute("foo");
    if (session.isNew()) {
        out.println("To jest nowa sesja.");
    } else {
        out.println("Witam ponownie!");
    }

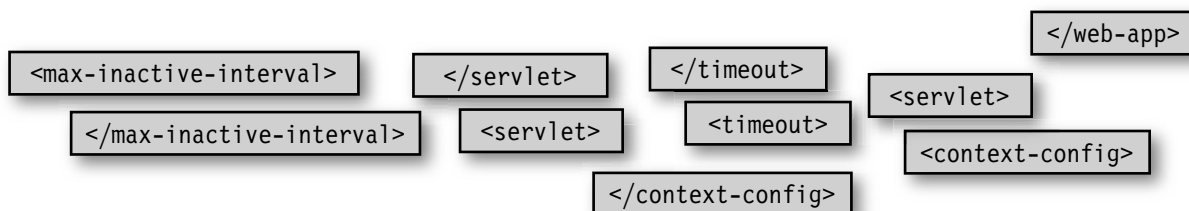
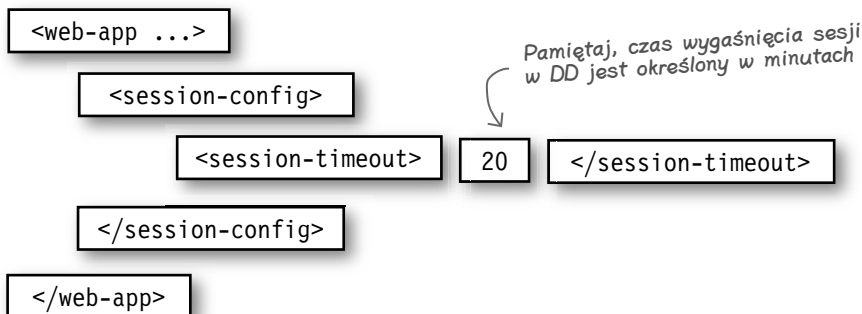
    out.println("Foo: " + foo);
}
```



Magnesiki z kodem – Odpowiedzi

Spróbuj przygotować taki kod deskryptora wdrożenia i aplikacji internetowej, który będzie prawidłowo niszczył sesję w sytuacji, gdy dany klient nie przysłał żadnego żądania przez 20 minut.

– DD



– Servlet

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws IOException {
```

```
    HttpSession session = request.getSession();
```

```
    session.setMaxInactiveInterval(1200);
```

W kodzie limit czasowy
określamy w SEKUNDACH.

```
    session.setTimeout(20);
```

```
    request.setTimeout(12000);
```

```
    request.getServletContext().getSession();
```



BĄDŹ kontenerem
— odpowiedzi

```

1 public void doGet(HttpServletRequest request, HttpServletResponse response)
                                   throws IOException {

    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    HttpSession session = request.getSession();
    session.setAttribute("foo", "42");
    session.setAttribute("bar", "420");
    session.invalidate(); ← tutaj unieważniamy sesję

    String foo = (String) session.getAttribute("foo");
    out.println("Foo: " + foo);
}

```

Och nie! Za późno na wywoływanie metody `getAttribute()` dla danej sesji, ponieważ JEST już nieważna!

Wynik: otrzymujemy wyjątek czasu wykonywania (`IllegalStateException`), ponieważ nie możemy odczytać wartości atrybutu PO unieważnieniu sesji.

```

2 public void doGet(HttpServletRequest request, HttpServletResponse response)
                                   throws IOException {

    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    HttpSession session = request.getSession();
    session.setAttribute("foo", "42");
    session.setMaxInactiveInterval(0); ← W tym miejscu wymuszamy NATYCHMIASTOWE
                                         przekroczenie limitu czasowego sesji, ponieważ
                                         stwierdzamy: „Ważność sesji kończy się już po
                                         0 sekundach nieaktywności.”

    String foo = (String) session.getAttribute("foo");

    if (session.isNew()) { ← Nie możemy wywołać metody isNew()
        out.println("To jest nowa sesja.");
    } else {
        out.println("Witam ponownie!");
    }

    out.println("Foo: " + foo);
}

```

Nie możemy wywołać metody `isNew()` dla sesji, która straciła swoją ważność. Mamy więc do czynienia z identycznym problemem jak w kodzie powyżej... nie możemy wywoływać metod dla unieważnionej sesji.

Wynik: otrzymujemy wyjątek czasu wykonywania (`IllegalStateException`), ponieważ nie można wywołać metody `isNew()` dla sesji już PO jej unieważnieniu. Ograniczenie maksymalnego odstępu czasu pomiędzy żądaniami (czasu nieaktywności) do zera oznacza, że limit czasowy sesji jest od razu przekroczony, zatem sama sesja jest natychmiast unieważniana!

Czy ciasteczek można używać do innych celów, czy służą tylko do przesyłania identyfikatorów sesji?



Chociaż ciasteczka (znaczniki kontekstu klienta) były początkowo projektowane z myślą o ułatwieniu obsługi stanu sesji, *możesz* z powodzeniem używać własnych, niestandardowych ciasteczek do innych celów. Pamiętaj, że znacznik kontekstu klienta nie jest niczym więcej, niż tylko małym fragmentem danych (parą łańcuchów nazwa-wartość) wymienianym pomiędzy klientem a serwerem. Serwer *wysyła* ciasteczko do klienta, który *zwraca* to samo ciasteczko wraz ze swoim kolejnym żądaniem.

Jedną z zalet znaczników kontekstu klienta jest to, że *użytkownik* w żaden sposób nie musi się angażować w ich obsługę — wymiana znaczników jest w pełni zautomatyzowana (oczywiście pod warunkiem, że w przeglądarce internetowej użytkownika jest włączona ich obsługa).

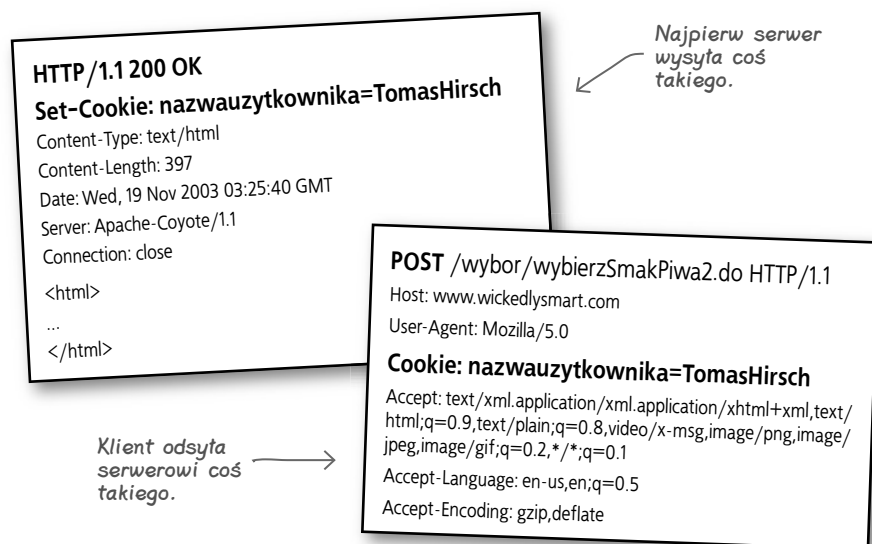
Ciasteczka domyślnie żyją tylko przez okres istnienia sesji; kiedy klient zamknie swoją przeglądarkę, znacznik jego kontekstu znika. Właśnie tak działa ciasteczko "JSESSIONID". **Możesz jednak wymusić na ciasteczku, by nie ginęło PO zamknięciu okna przeglądarki.**

W ten sposób umożliwisz swojej aplikacji internetowej odczytywanie informacji zawartych w tym znaczniku nawet wtedy, gdy sesja danego klienta od dawna nie będzie istniała. Wyobraź sobie, że Kim chce, aby jego aplikacja wyświetlała nazwisko za każdym razem, gdy użytkownik wraca na witrynę poświęconą poradom piwnym. Kim ustawia więc odpowiednie ciasteczko w momencie, gdy po raz pierwszy uzyskuje dane klienta, i jeśli otrzyma to ciasteczko ponownie (w ramach któregoś z późniejszych żądań), będzie już wiedział, że nie musi ponownie pytać o nazwisko użytkownika. *I nie ma znaczenia, czy dany użytkownik zamknął i ponownie uruchomił swoją przeglądarkę i czy nie odwiedzał witryny Kima przez tydzień!*

Możesz używać znaczników kontekstu klienta do wymiany par łańcuchów nazwa-wartość pomiędzy serwerem a klientem.

Serwer wysyła ciasteczko do klienta, który odsyła je wraz z każdym kolejnym żądaniem.

Znaczniki kontekstu klienta z identyfikatorami sesji znikają w momencie zamknięcia okna przeglądarki, ale **MOŻESZ** wymusić trwanie znacznika po stronie klienta także po wyłączeniu przeglądarki.



Stosowanie ciasteczek przy użyciu interfejsu API serwletów

Okazuje się, że nagłówki związane z ciasteczkami *można* wyodrębnić z obiektów żądania i odpowiedzi. Wszystkie mechanizmy niezbędne do obsługi tych ciasteczek zaimplementowano w trzech klasach interfejsu API serwletów: `HttpServletRequest`, `HttpServletResponse` i `Cookie`.

```
<<interfejs>>
javax.servlet.http.HttpServletRequest

getContextPath()
getCookies()
getHeader(String)
getQueryString()
getSession()
// I WIELE innych metod...
```

```
<<interfejs>>
javax.servlet.http.HttpServletResponse

addCookie()
addHeader()
encodeRedirectURL()
sendError()
setStatus()
// I WIELE innych metod...
```

```
javax.servlet.http.Cookie

Cookie(String, String)

String getDomain()
int getMaxAge()
String getName()
String getPath()
boolean getSecure()
String getValue()
void setDomain(String)
void setMaxAge(int)
void setPath(String)
void setValue(String)
// kilka innych metod...
```

Tworzenie nowego obiektu klasy `Cookie`

```
Cookie cookie = new Cookie("nazwaUzytkownika", nazwa );
```

Konstruktor klasy `Cookie` otrzymuje na wejściu parę tańcuchów nazwa-wartość.

Ustawianie długości życia znacznika kontekstu po stronie klienta

```
cookie.setMaxAge(30*60);
```

Wysyłanie ciasteczka do klienta

```
response.addCookie(cookie);
```

Odczytywanie ciasteczka (ciasteczek) z żądania klienta

```
Cookie[] cookies = request.getCookies();
for (int i = 0; i < cookies.length; i++) {
    Cookie cookie = cookies[i];
    if (cookie.getName().equals("nazwaUzytkownika")) {
        String nazwaUzytkownika = cookie.getValue();
        out.println("Witaj, " + nazwaUzytkownika);
        break;
    }
}
```

Metoda `setMaxAge()` definiuje czas życia znacznika w SEKUNDACH. Przedstawiony obok fragment kodu mówi: „będziesz żył przez 30*60 sekund” (30 minut). Ustawienie maksymalnego czasu życia równego -1 powoduje, że znacznik kontekstu zniknie w momencie zamknięcia okna przeglądarki. Jeśli więc wywołasz metodę `getMaxAge()` dla ciasteczka „JSESSIONID”, jaką wartość otrzymasz?

Nie istnieje metoda `getCookie(String)`... dostęp do znaczników kontekstu klienta jest możliwy tylko za pośrednictwem tablicy obiektów klasy `Cookie`, zatem znalezienie interesującego nas znacznika wymaga iteracyjnego przeszukania tej tablicy w odpowiedniej pętli.

Przykład prostego ciasteczka niestandardowego

Wyobraź sobie, że Kim chce umieścić na swojej stronie formularz, w którym poprosi użytkownika o przesłanie jego identyfikatora. Formularz wywoła serwlet, który odczyta parametr żądania reprezentujący nazwę użytkownika i użyje tej nazwy do ustawienia ciasteczka w obiekcie odpowiedzi.

Kiedy następnym razem ten sam użytkownik przyśle żądanie adresowane do KTÓREGOKOLWIEK z serwletów tej aplikacji internetowej, znacznik kontekstu klienta zostanie odesłany w ramach tego żądania (zakładając, że dany znacznik nadal istnieje, w zależności od wartości reprezentowanej w polu `maxAge` obiektu ciasteczka). Serwlet należący do tej aplikacji, widząc to ciasteczko, może umieszczać reprezentowaną tam nazwę użytkownika w dynamicznie generowanych odpowiedziach, a logika biznesowa wie, że nie musi już prosić użytkownika o ponowne wpisanie identyfikatora.

Niniejszy kod jest uproszczoną wersją testową dla opisanego powyżej scenariusza.

Serwlet tworzący i USTAWIAJĄCY ciasteczko

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class TestCiasteczek extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException {

        response.setContentType("text/html");

        String nazwa = request.getParameter("nazwauzytkownika");

        Cookie cookie = new Cookie("nazwauzytkownika", nazwa);

        cookie.setMaxAge(30*60);

        response.addCookie(cookie);

        RequestDispatcher view = request.getRequestDispatcher("cookiewynik.jsp");
        view.forward(request, response);
    }
}
```

Zwraca wpisaną w formularzu nazwę użytkownika.

Tworzy nowy znacznik kontekstu klienta, w którym będzie przechowywana nazwa użytkownika.

Ciasteczko będzie przechowywane po stronie klienta przez 30 minut.

Dodaje ciasteczko do obiektu odpowiedzi w postaci nagłówka „Set-Cookie”.

Przekazuje wskazanej stronie JSP odpowiedzialność za wygenerowanie kodu odpowiedzi.

Strona JSP, która wizualizuje widok na podstawie informacji otrzymanych od serwletu

```
<html><body>
  <a href="sprawdzcookie.do">kliknij tutaj</a>
</body></html>
```

No dobrze, wszystko rozumiem, nie ma tu zbyt dużo kodu JSP, ale nie znosimy przekazywać z serwletu nawet TYŁE kodu HTML. Fakt, że przekazujemy żądanie do strony JSP, w żaden sposób nie wpływa na ustawienia ciasteczka, które znajduje się w obiekcie odpowiedzi już w czasie jego przekazania do JSP...

Przykład prostego ciasteczka niestandardowego, ciąg dalszy...

Serwlet ODCZYTUJĄCY znacznik kontekstu klienta

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class TestCiasteczek extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException {

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        Cookie[] cookies = request.getCookies();

        for (int i = 0; i < cookies.length; i++) {
            Cookie cookie = cookies[i];
            if (cookie.getName().equals("nazwa uzytkownika")) {
                String nazwaUzytkownika = cookie.getValue();
                out.println("Witaj, " + nazwaUzytkownika);
                break;
            }
        }
    }
}
```

Zwraca znaczniki kontekstu klienta
zawarte w obiekcie żądania.

Przegląda kolejno elementy tablicy
ciasteczek w poszukiwaniu
ciasteczka nazwanego
„nazwa uzytkownika”. Jeśli
odpowiednie ciasteczko zostanie
znalezione, odczytuje i wyświetla
jego wartość.



Oglądaj to!

**Nie myl znaczników kontekstu klienta
z nagłówkami!**

Kiedy dodajemy **nagłówek** do odpowiedzi, przekazujemy łańcuchy
nazwy i wartości w postaci argumentów metody `addHeader()`:

```
response.addHeader("foo", "bar");
```

Kiedy jednak dodajemy do odpowiedzi **ciasteczko**, przekazujemy za
pośrednictwem argumentu metody `addCookie()` obiekt klasy `Cookie`.
Nazwę i wartość znacznika ustawiamy w konstruktorze tej klasy.

```
Cookie cookie = new Cookie("nazwa", nazwa);
response.addCookie(cookie);
```

Musisz też pamiętać, że interfejs API serwletów oferuje zarówno
metodę `setHeader()`, jak i metodę `addHeader()` (metoda `addHeader()`
dodaje nową wartość do istniejącego nagłówka, jeśli wcześniej
zdefiniowano nagłówek z taką samą nazwą, natomiast metoda
`setHeader()` zastępuje istniejącą wartość). **NIE** istnieje jednak metoda
`setCookie()`; API serwletów definiuje tylko metodę `addCookie()`.



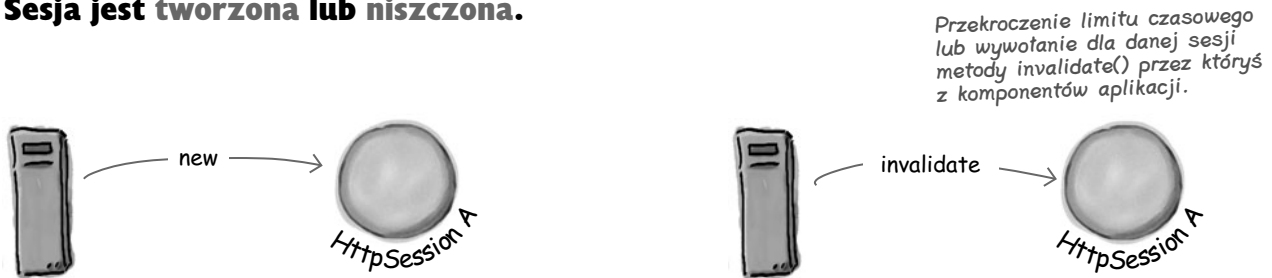
Relax Nie musisz znać **WSZYSTKICH**
prezentowanych tutaj metod
klasy `Cookie`.

Przed egzaminem nie musisz się uczyć na pamięć
wszystkich metod udostępnianych w klasie
`Cookie`, ale musisz znać metody odczytujące
i dodające ciasteczka do obiektów żądań
i odpowiedzi. Powinieneś także wiedzieć, jak
korzystać z konstruktora klasy `Cookie` oraz metod
`getMaxAge()` i `setMaxAge()`.

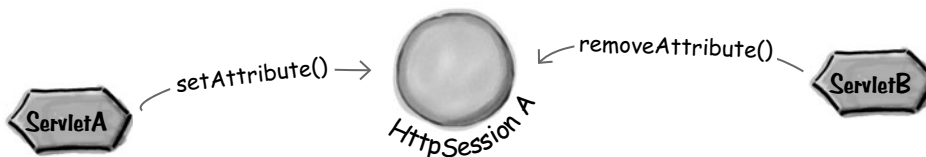
Kluczowe momenty w życiu obiektu HttpSession

Najważniejsze zdarzenia w życiu obiektu klasy HttpSession:

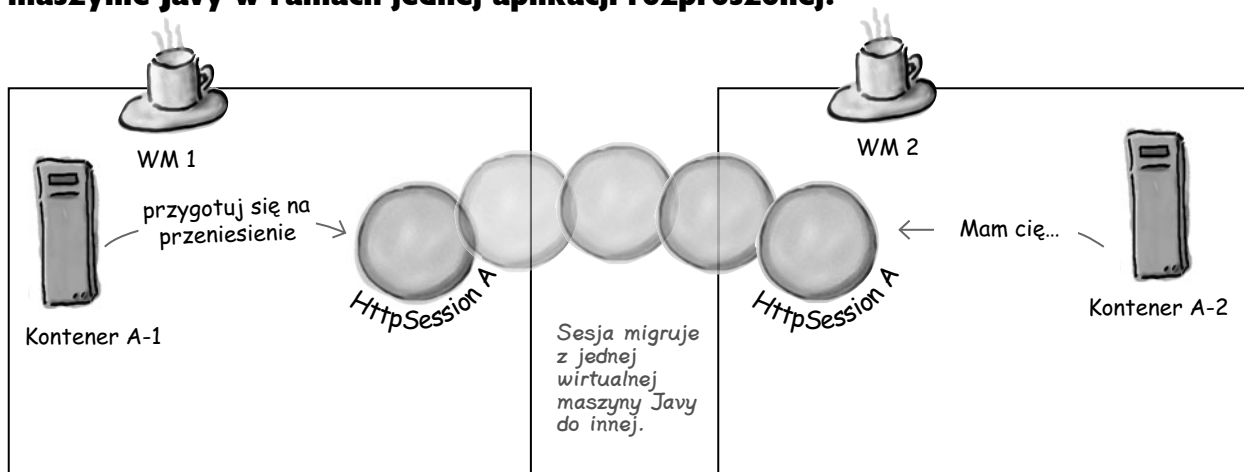
Sesja jest tworzona lub niszczona.



Atrybuty sesji są dodawane, usuwane lub zastępowane przez pozostałe składniki aplikacji internetowej.



Sesja jest dezaktywowana w jednej i aktywowana w innej wirtualnej maszynie Javy w ramach jednej aplikacji rozproszonej.



Zdarzenia z cyklu życia sesji

Zdarzenie

Cykl życia

Sesja została *utworzona*

Kiedy kontener po raz pierwszy tworzy sesję. Na tym etapie sesja wciąż jest uważana za *nową* (innymi słowy, klient nie nadesłał jeszcze żadnego żądania z identyfikatorem tej sesji).

Sesja została *zniszczona*

Kiedy kontener unieważnia sesję (albo wskutek przekroczenia limitu czasowego, albo dlatego, że któryś ze składników danej aplikacji wywołał dla tej sesji metodę `invalidate()`).

Atrybuty

Atrybut został *dodany*

Kiedy któryś ze składników danej aplikacji internetowej wywołuje metodę `setAttribute()` dla sesji.

Atrybut został *usunięty*

Kiedy któryś ze składników danej aplikacji internetowej wywołuje metodę `removeAttribute()` dla sesji.

Atrybut został *zastąpiony*

Kiedy któryś ze składników danej aplikacji internetowej wywołuje dla sesji metodę `setAttribute()` i okazuje się, że podana nazwa atrybutu została wcześniej związana z tą sesją.

Migracja

Sesja *ma* zostać *dezaktywowana*

Kiedy kontener planuje migrację (przeniesienie) sesji do innej wirtualnej maszyny Javy. Zdarzenie jest wywoływane *przed* przeniesieniem sesji, dzięki czemu atrybuty mają możliwość przygotowania się do procesu migracji.

Sesja została *aktywowana*

Kiedy kontener *właśnie* przeniósł sesję do innej wirtualnej maszyny Javy. Zdarzenie jest wywoływane, zanim którykolwiek ze składników aplikacji będzie mógł wywołać metodę `getAttribute()` dla danej sesji, dzięki czemu przeniesione przed chwilą atrybuty mają możliwość przygotowania się do przyszłych operacji dostępu.

Typ zdarzenia i interfejsu nasłuchującego

HttpSessionEvent



HttpSessionListener

HttpSessionBindingEvent



HttpSessionAttributeListener

HttpSessionEvent



HttpSessionActivationListener

Nie zapomnij o interfejsie HttpSessionBindingListener

Zdarzenia wymienione na poprzedniej stronie są ściśle związane z kluczowymi momentami w życiu *sesji*, natomiast interfejs nasłuchujący HttpSessionBindingListener obsługuje kluczowe momenty w życiu *atributu sesji*. Zapewne pamiętasz z rozdziału 5. (w którym analizowaliśmy możliwe zastosowania interfejsów nasłuchujących), że możemy informować atrybut np. o tym, że jest dodawany do sesji, aby zapewnić mu możliwość zsynchronizowania jego wartości z zawartością bazy danych (i zaktualizowania bazy danych, kiedy będzie usuwany z sesji). Poniżej przedstawiamy krótkie przypomnienie zagadnień zaprezentowanych w rozdziale 5.

Ten interfejs nasłuchujący służy właśnie informowaniu *mnie*, że *jestem* umieszczany w sesji (lub usuwany z sesji). Z metod tego interfejsu nie dowiem się jednak niczego o pozostałych zdarzeniach związanych z sesją.

```
package com.example;
```

```
import javax.servlet.http.*;
```

```
public class Pies implements HttpSessionBindingListener {
```

```
    private String rasa;
```

```
    public Pies(String rasa) {
        this.rasa = rasa;
    }
```

```
    public String getRasa() {
        return rasa;
    }
```

```
    public void valueBound(HttpSessionBindingEvent event) {
        // kod wykonywany teraz, skoro wiem już, że jestem w sesji
    }
```

```
    public void valueUnbound(HttpSessionBindingEvent event) {
        // kod wykonywany teraz, skoro wiem, że nie jestem już częścią sesji
    }
```

```
}
```

Ten interfejs nasłuchujący należy do pakietu javax.servlet.http.

Tym razem atrybut Pies jest **JEDNOCZEŚNIE** obiektem klasy implementującej interfejs HttpSessionBindingListener, zatem nasłuchuje zdarzeń polegających na dodawaniu lub usuwaniu samego obiektu Pies z sesji.

W tym przypadku słowo „Bound” oznacza, że ktoś **DODAŁ** ten atrybut do sesji.

Pewnie się domyślasz, co oznacza słowo „Unbound”.



Oglądaj to!

NIE WSZYSTKIE obiekty nasłuchujące dla związków atrybutów z sesjami konfigurowujemy w deskrytorze wdrożenia!

Jeśli klasa atrybutu (podobna do przedstawionej przed chwilą klasy Pies) implementuje interfejs HttpSessionBindingListener, kontener wywołuje metody zwrotne obsługujące zdarzenia (valueBound() i valueUnbound()) w momencie, w którym jakiś egzemplarz tej klasy jest odpowiednio dodawany lub usuwany z sesji. To wszystko. To po prostu działa. Ta zasada **NIE** jest jednak stosowana w przypadku pozostałych interfejsów (wymienionych na poprzedniej stronie) nasłuchujących dla sesji. Implementacje interfejsów HttpSessionListener i HttpSessionAttributeListener muszą być dodatkowo rejestrowane w deskrytorze wdrożenia, ponieważ mają związek z samą sesją, nie z umieszczanymi w niej pojedynczymi atrybutami.

Migracja sesji

Zapewne pamiętasz, jak w poprzednim rozdziale wspominaliśmy o rozproszonych aplikacjach internetowych, których pewne składniki mogą być powielane (replikowane) w wielu węzłach połączonych siecią komputerową. W środowisku klastrowym kontener może stosować mechanizm *równoważenia obciążeń* przez odbieranie żądań klientów jednym wirtualnym maszynom Javy i przekazywanie ich innym, mniej obciążonym maszynom (które mogą, ale nie muszą pracować na innym fizycznym komputerze, co z naszego punktu widzenia nie ma najmniejszego znaczenia). Najważniejsze jest to, że aplikacja jest rozproszona na wielu serwerach.

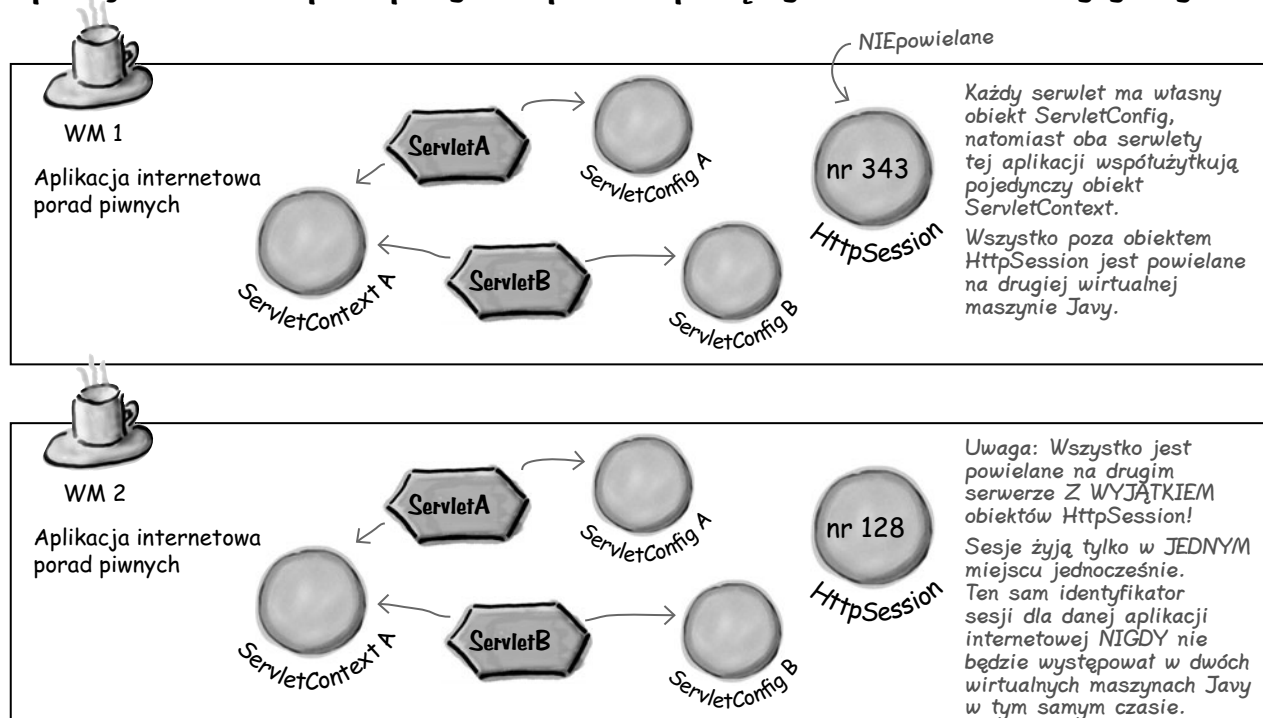
Oznacza to, że za każdym razem, gdy jakiś klient generuje i przysyła żądanie, może ono zostać przydzielone do *innego* egzemplarza tego samego serwletu. Innymi słowy, żądanie *A* odwołujące się do serwletu *A* może być zrealizowane na jednej wirtualnej maszynie Javy, natomiast żądanie *B* odwołujące się do tego samego serwletu *A* może być wykonywane na zupełnie innej wirtualnej maszynie Javy. W tej sytuacji naturalne jest pytanie o to, co dzieje się z takimi składnikami aplikacji jak obiekty `ServletContext`, `ServletConfig` i `HttpSession`.

Odpowiedź jest prosta, jej skutki już takie proste nie są:

Tylko obiekty `HttpSession` (i zawarte w nich atrybuty) są przenoszone pomiędzy wirtualnymi maszynami Javy.

Istnieje dokładnie jeden obiekt `ServletContext` w *każdej* wirtualnej maszynie Javy. Istnieje dokładnie jeden obiekt `ServletConfig` dla *każdego* serwletu w *każdej* wirtualnej maszynie Javy. **Ale w każdej aplikacji internetowej istnieje tylko jeden obiekt `HttpSession` dla określonego identyfikatora sesji, niezależnie od liczby wirtualnych maszyn Javy, na których pracuje ta aplikacja rozproszona.**

Aplikacja internetowa porad piwnych rozproszona pomiędzy dwie wirtualne maszyny Javy



Migracja sesji w akcji

Sposób zarządzania klastrami i stosowanie mechanizmów rozpraszania aplikacji internetowych pomiędzy węzły środowiska rozproszonego zależy od producentów poszczególnych rozwiązań, a sama specyfikacja standardu J2EE nie wymusza implementowania obsługi aplikacji rozproszonych. Niniejszy schemat ma na celu jedynie przedstawienie ogólnej koncepcji funkcjonowania tego typu rozwiązań. Kluczowe znaczenie ma fakt, że o ile pozostałe składniki aplikacji są *powielane (replikowane)* w każdym węzle (tj. w każdej wirtualnej maszynie Javy), o tyle obiekty sesji są pomiędzy tymi węzłami *przenoszone*. Tego możemy być pewni (niezależnie od producenta kontenera). Innymi słowy, jeśli jakiś producent *zaimplementuje* obsługę aplikacji rozproszonych, jego kontener będzie *musiał* zawierać mechanizm migracji sesji pomiędzy wirtualnymi maszynami Javy. To wymaganie dotyczy także migracji wszystkich atrybutów składowanych w przenoszonych sesjach.

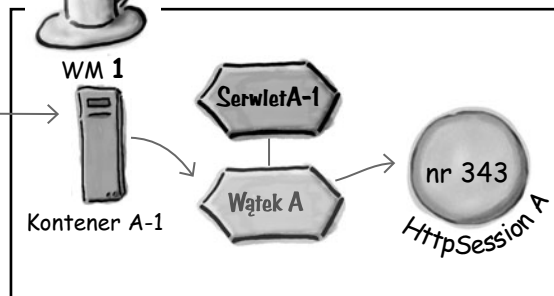
- ① Diane wybiera "Słabe" i klika przycisk Prześlij kwerendę.

Serwer równoważenia obciążeń decyduje o przekazaniu tego żądania do kontenera A-1 w WM 1.



Serwer/kontener równoważenia obciążeń

Kontener tworzy nową sesję i przydziela jej identyfikator 343. Ciasteczko "JSESSIONID" jest odsyłane z powrotem do Diane w ramach odpowiedzi (tego kroku nie przedstawiono).



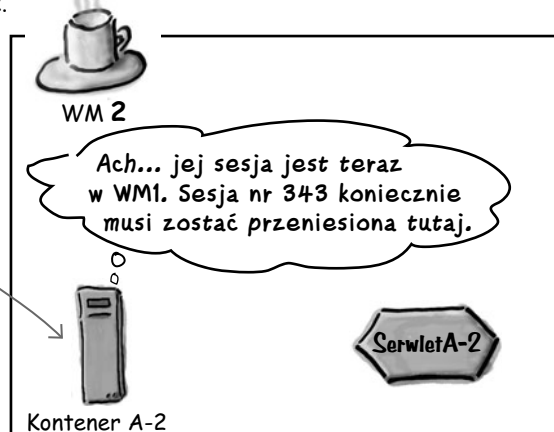
- ② Diane wybiera "Mocne" i klika przycisk Prześlij kwerendę. Jej żądanie dodatkowo zawiera znacznik kontekstu klienta "JSESSIONID" z identyfikatorem sesji nr 343.

Tym razem serwer równoważenia obciążeń decyduje o przekazaniu tego żądania do kontenera A-2 w WM 2.



Serwer/kontener równoważenia obciążeń

Kontener otrzymuje żądanie, widzi zawarty w nim identyfikator sesji i uświadamia sobie, że odpowiedni obiekt sesji jest przechowywany w innej wirtualnej maszynie Javy, w WM 1!

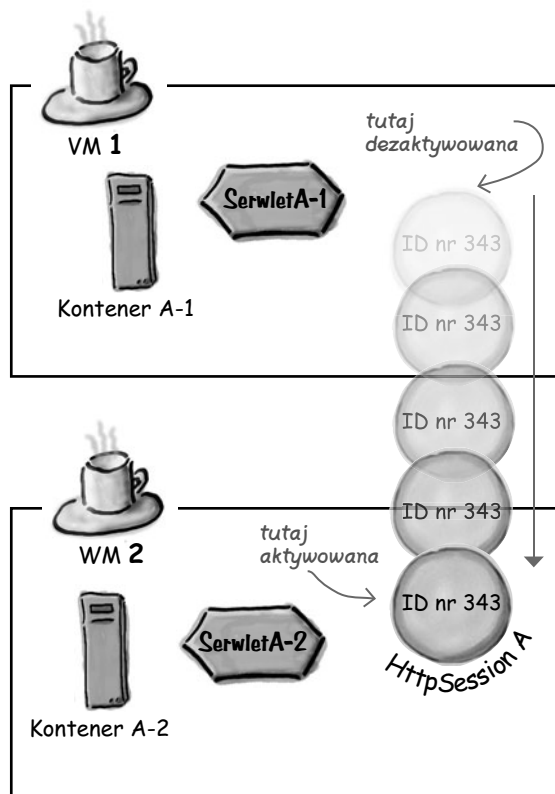


- 3 Sesja nr 343 migruje z pierwszej wirtualnej maszyny Javy (WM 1) do drugiej (WM 2). Innymi słowy, kiedy już zostanie przeniesiona do WM 1, automatycznie **przestanie istnieć w WM 1**.

Taka migracja w praktyce oznacza, że sesja jest z jednej strony **dezaktywowana w WM 1** i z drugiej strony **aktywowana w WM 2**.



Serwer/kontener
równoważenia obciążeń



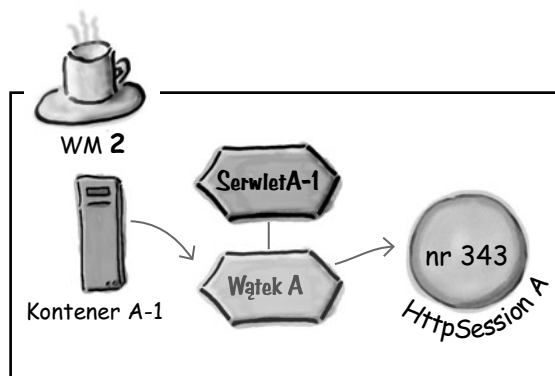
4

Kontener tworzy nowy wątek dla serwletu A i wiąże nowe żądanie Diane z przeniesioną przed chwilą sesją nr 343.

Do tego samego wątku serwletu dociera nowe żądanie Diane i wszyscy są zadowoleni. Diane nie ma pojęcia, co się stało (zauważyła jedynie nieznaczne opóźnienie w realizacji żądania w związku z przeniesieniem sesji).



Serwer/kontener
równoważenia obciążeń



HttpSessionActivationListener umożliwia atrybutom przygotowanie się do wielkiej przeprowadzki...

Ponieważ istnieje *możliwość* migracji obiektu `HttpSession` z jednej wirtualnej maszyny Javy do innej, projektanci specyfikacji serwetów doszli do wniosku, że dobrym rozwiązaniem będzie umożliwienie sygnalizowania atrybutom należącym do sesji, że także one zostaną za moment przeniesione. W ten sposób możemy zagwarantować, że uprzedzone atrybuty będą odpowiednio przygotowane do szykowanej wycieczki.

Ten interfejs nasłuchujący został stworzony po to, abym jako atrybut przenoszony sesji wiedział z wyprzedzeniem o własnej przeprowadzce do innej wirtualnej maszyny i abym mógł się upewnić, że moje zmienne klasowe są na to gotowe...

Jeśli wszystkie Twoje atrybuty mają postać prostych, serializowalnych obiektów (implementujących interfejs `Serializable`), dla których nie ma znaczenia miejsce występowania, najprawdopodobniej nigdy nie będziesz musiał korzystać z odpowiedniego interfejsu nasłuchującego. Wydaje nam się, że w praktyce 95,324% wszystkich działających aplikacji internetowych nigdy nie korzysta z tego interfejsu. Warto jednak pamiętać o jego istnieniu na wypadek, gdybyśmy rzeczywiście go potrzebowali — jego najbardziej prawdopodobnym zastosowaniem jest umożliwienie atrybutom przygotowania ich zmiennych klasowych do procesu serializacji.



Migracja sesji i serializacja

Na tym etapie wszystko jest już nieco trudniejsze...

Migracja serializowalnych atrybutów wymaga udziału kontenera (który zakłada, że wszystkie zmienne klasowe w ramach atrybutu są albo serializowalne, albo równe `null`).

Udział kontenera nie jest jednak wymagany w samym procesie serializacji przeprowadzanej w celu umożliwienia migracji obiektu `HttpSession`!

<<interfejs>> HttpSessionActivationListener <code>sessionDidActivate(HttpSessionEvent)</code> <code>sessionWillPassivate(HttpSessionEvent)</code>	
<code>javax.servlet.http.HttpSessionActivationListener</code>	

Co to właściwie oznacza dla nas? To proste: musimy doprowadzić do sytuacji, w której klasy naszych atrybutów będą implementowały interfejs `Serializable`, wówczas raz na zawsze pozbędziemy się tego problemu. Jeśli jednak klasy te z jakiegoś powodu *nie* będą serializowalne (np. wskutek tego, że jeden z typów atrybutów klasy nie jest serializowalny), klasy atrybutów będą musiały implementować interfejs `HttpSessionActivationListener` i korzystać z metod zwrotnych dla zdarzeń aktywacji i dezaktywacji.



Oglądaj to!

Kontener nie jest ZOBOWIĄZANY do przeprowadzania procesu serializacji, zatem wcale nie mamy gwarancji, że metody `readObject()` i `writeObject()` zostaną wywołane w czasie serializacji atrybutu lub jednej z jego zmiennych klasowych!

Jeśli serializacja nie jest Ci obca, z pewnością wiesz, że klasa implementująca interfejs `Serializable` opcjonalnie może implementować także metodę `writeObject()`, która będzie wywoływana przez wirtualną maszynę Javy za każdym razem, gdy jakiś obiekt danej klasy będzie serializowany, oraz metodę `readObject()` wywoływaną za każdym razem, gdy obiekt będzie deserializowany. Serializowalny obiekt może używać tych metod np. do przypisywania nieserializowalnym polom wartości `null` w procesie serializacji (metoda `writeObject()`), a następnie do odtwarzania tych pól w procesie deserializacji (metoda `readObject()`). (Nie przejmuj się, jeśli NIE masz doświadczenia z serializowaniem obiektów i nie znasz szczegółów tego procesu). Warto jednak pamiętać, że wymienione metody nie zawsze są wywoływane podczas migracji sesji! Jeśli więc musisz zachować i odtworzyć stan należący do atrybutu zmiennej klasowej, użyj interfejsu nasłuchującego `HttpSessionActivationListener` i jego dwóch metod zwrotnych obsługujących żądania (`sessionDidActivate()` oraz `sessionWillPassivate()`) w taki sam sposób, w jaki wykorzystywałbyś metody `readObject()` i `writeObject()`.

Przykłady klas nasłuchujących

Studiując kolejne trzy strony, zwracaj szczególną uwagę na typy obiektów zdarzeń oraz na to, czy klasa nasłuchująca jest jednocześnie klasą atrybutu.

Licznik sesji

Interfejs nasłuchujący umożliwia nam śledzenie liczby aktywnych sesji w danej aplikacji internetowej. To bardzo proste.

```
package com.example;
import javax.servlet.http.*;
```

```
public class LicznikSesjiPiwnych implements HttpSessionListener {
```

```
    static private int aktywneSesje;
```

```
    public static int getAktywneSesje() {
        return aktywneSesje;
    }
```

Ta klasa zostanie umieszczona w katalogu WEB-INF/classes tak jak pozostałe klasy tej aplikacji internetowej, dzięki czemu wszystkie serwlety i inne klasy pomocnicze będą miały dostęp do tej metody.

```
    public void sessionCreated(HttpSessionEvent event) {
        aktywneSesje++;
    }
```

Te metody otrzymują na wejściu obiekty klasy HttpSessionEvent.

```
    public void sessionDestroyed(HttpSessionEvent event) {
        aktywneSesje--;
    }
}
```

Konfigurowanie licznika sesji w deskrytorze wdrożenia

```
<web-app ...>
...
    <listener>
        <listener-class>
            com.example.LicznikSesjiPiwnych
        </listener-class>
    </listener>
</web-app>
```

Do Twojej wiadomości: przedstawione rozwiązanie nie będzie działało prawidłowo, jeśli dana aplikacja będzie rozproszona pomiędzy wiele wirtualnych maszyn Javy, ponieważ nie uwzględniono w nim żadnych mechanizmów synchronizowania zmiennych statycznych. Jeśli dana klasa zostanie załadowana na więcej niż jednej wirtualnej maszynie Javy, każda z jej kopii będzie utrzymywała własną wartość statycznej zmiennej licznika.

Przykłady klas nasłuchujących

Klasa nasłuchująca dla atrybutów

Przedstawiona poniżej klasa nasłuchująca umożliwia nam śledzenie zdarzeń polegających na dodawaniu, usuwaniu lub zastępowaniu atrybutów w ramach obiektu sesji.

```
package com.example;
import javax.servlet.http.*;
```

W tym interfejsie nasłuchującym zastosowano niespójny schemat nazewnictwa — jest to interfejs `Attribute`, mimo że jego metody otrzymują na wejściu zdarzenie `Binding`.

```
public class AtrybutyPiwa implements HttpSessionAttributeListener {

    public void attributeAdded(HttpSessionBindingEvent event) {
        String nazwa = event.getName();
        Object wartosc = event.getValue();
        System.out.println("Dodano atrybut: " + nazwa + ": " + wartosc);
    }

    public void attributeRemoved(HttpSessionBindingEvent event) {
        String nazwa = event.getName();
        Object wartosc = event.getValue();
        System.out.println("Usunięto atrybut: " + nazwa + ": " + wartosc);
    }

    public void attributeReplaced(HttpSessionBindingEvent event) {
        String nazwa = event.getName();
        Object wartosc = event.getValue();
        System.out.println("Zastąpiono atrybut: " + nazwa + ": " + wartosc);
    }
}
```

Zdarzenie typu `HttpSessionBindingEvent` umożliwia nam odczytywanie nazwy i wartości atrybutu, dla którego zostało wywołane.

Konfigurowanie powyższej klasy nasłuchującej w deskrytorze wdrożenia

```
<web-app ...>
...
<listener>
  <listener-class>
    com.example.AtrybutyPiwa
  </listener-class>
</listener>
</web-app>
```

P: Hej, gdzie do diabła wyświetlasz te komunikaty? Co tak naprawdę oznacza `System.out` w aplikacji internetowej?

O: Gdziekolwiek nasz kontener uzna za stosowne (miejsce przekazywania komunikatów może, ale nie musi być elementem konfigurowanym przez administratora). Innymi słowy, wszystko zależy od producenta kontenera — często jest to plik dziennika. Na przykład kontener Tomcat umieszcza dane wyjściowe w pliku `tomcat/logs/catalina.log`. Aby dowiedzieć się, gdzie Twój kontener zapisuje dane ze standardowego wyjścia, będziesz musiał przejrzeć jego dokumentację.

Przykłady klas nasłuchujących

Klasa atrybutu (nasłuchująca zdarzeń mających wpływ na JEJ działanie)

Przedstawiona poniżej klasa nasłuchująca umożliwi naszemu atrybutowi śledzenie zdarzeń, które mogą być dla niego istotne — czyli zdarzenia polegające na dodaniu lub usunięciu tego atrybutu z sesji oraz na migracji samej sesji z jednej wirtualnej maszyny Javy do innej.

```
package com.example;
import javax.servlet.http.*;
import java.io.*;

public class Pies implements HttpSessionBindingListener,
                               HttpSessionActivationListener, Serializable {

    private String rasa;
    // wyobraź sobie, że w tym miejscu zdefiniowano więcej zmiennych
    // klasowych, włącznie ze zmiennymi nieserializowalnymi

    // wyobraź sobie, że w tym miejscu zdefiniowano konstruktor i pozostałe metody get/set

    public void valueBound(HttpSessionBindingEvent event) {
        // kod wykonywany w sytuacji, gdy już wiem, że jestem w sesji
    }

    public void valueUnbound(HttpSessionBindingEvent event) {
        // kod wykonywany w sytuacji, gdy już wiem, że nie jestem już częścią sesji
    }

    public void sessionWillPassivate(HttpSessionEvent event) {
        // kod wprowadzający nieserializowalne pola w stan, który umożliwi im
        // przetrwanie w czasie przenoszenia do nowej wirtualnej maszyny Javy
    }

    public void sessionDidActivate(HttpSessionEvent event) {
        // kod odtwarzający pola... kod cofający wszystkie operacje wykonane
        // w ramach metody sessionWillPassivate()
    }
}
```

Zdarzenia
wiązania sesji.

Zdarzenia aktywacji
sesji (zwróć jednak
uwagę na fakt, iż obie
metody otrzymują na
wejściu obiekt typu
HttpSessionEvent).

Interfejsy nasłuchujące zdarzeń związanych z sesjami

Scenariusz	Interfejs/metody nasłuchujące	Typ zdarzenia	Najczęściej implementowane przez
Chcemy wiedzieć, ilu użytkowników jest jednocześnie obsługiwanych przez aplikację. Innymi słowy, chcemy śledzić aktywne sesje.	HttpSessionListener (javax.servlet.http) <i>sessionCreated</i> <i>sessionDestroyed</i>	HttpSessionEvent	<input type="checkbox"/> Klasa atrybutu <input checked="" type="checkbox"/> Jakaś inna klasa
Chcemy wiedzieć, kiedy sesja jest przenoszona z jednej wirtualnej maszyny Javy do innej.	HttpSessionActivationListener (javax.servlet.http) <i>sessionDidActivate</i> <i>sessionWillPassivate</i>	HttpSessionEvent Uwaga: nie istnieje specjalne zdarzenie <u>HttpSessionActivationEvent</u> .	<input checked="" type="checkbox"/> Klasa atrybutu <input checked="" type="checkbox"/> Jakaś inna klasa
Mamy klasę atrybutu (klasę obiektów wykorzystywanych w roli wartości atrybutu) i chcemy, aby obiekty tego typu były informowane, kiedy są wiązane z sesją i kiedy są z tej sesji usuwane.	HttpSessionBindingListener (javax.servlet.http) <i>valueBound</i> <i>valueUnbound</i>	HttpSessionBindingEvent	<input checked="" type="checkbox"/> Klasa atrybutu <input type="checkbox"/> Jakaś inna klasa
Chcemy wiedzieć, kiedy atrybut sesji jest dodawany, usuwany lub zastępowany w ramach danej sesji.	HttpSessionAttributeListener (javax.servlet.http) <i>attributeAdded</i> <i>attributeRemoved</i> <i>attributeReplaced</i>	HttpSessionBindingEvent Uwaga: nie istnieje specjalne zdarzenie <u>HttpSessionAttributeEvent</u> .	<input type="checkbox"/> Klasa atrybutu <input checked="" type="checkbox"/> Jakaś inna klasa

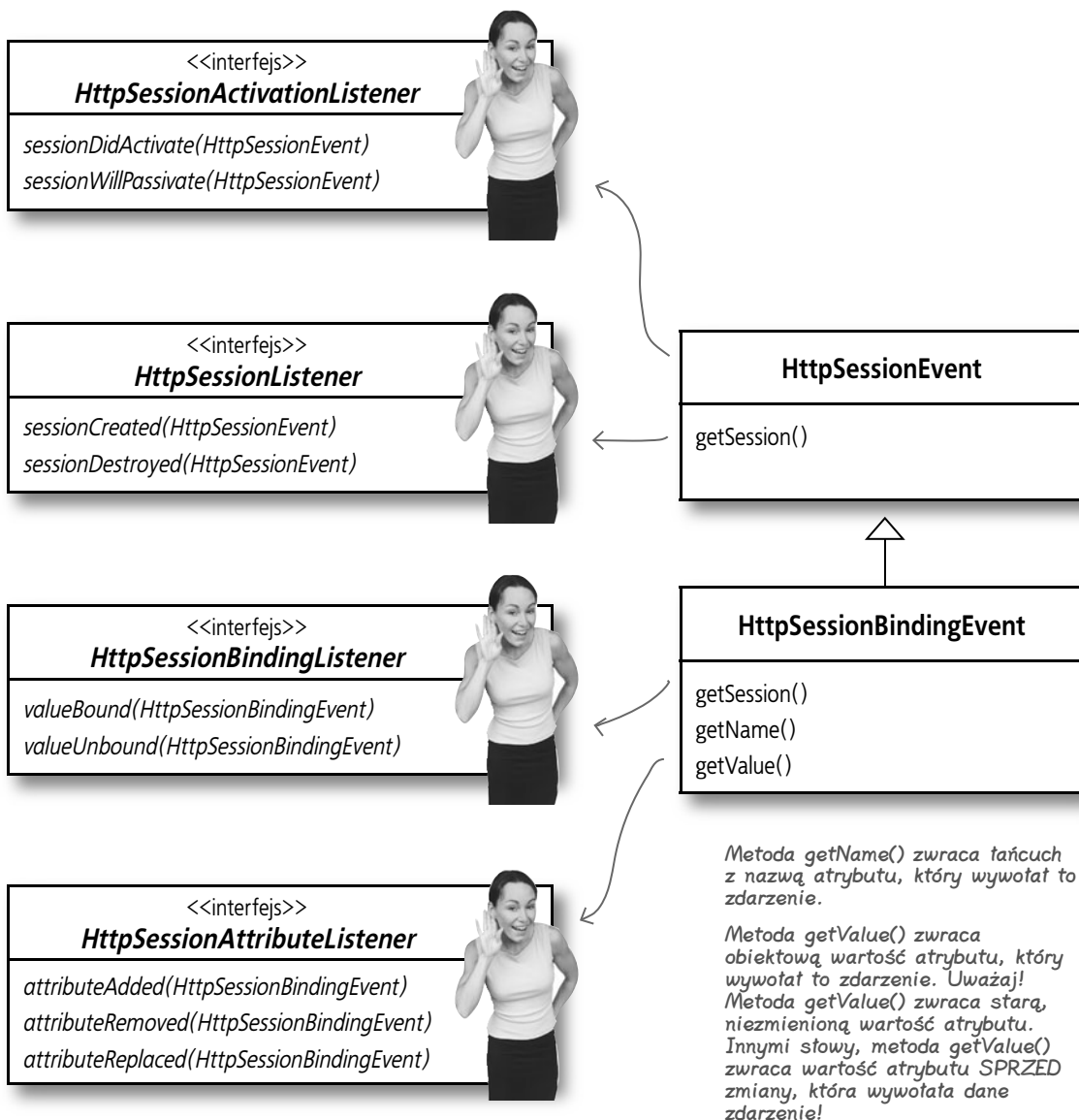


Oglądaj to!

Nazwy niektórych zdarzeń związanych z sesjami są niezgodne z podstawowymi konwencjami nazewnictwa zdarzeń.

Metody interfejsu **HttpSessionListener** otrzymują na wejściu zdarzenia typu **HttpSessionEvent**.
Metody interfejsu **HttpSessionBindingListener** otrzymują na wejściu zdarzenia typu **HttpSessionBindingEvent**.
Ale metody interfejsu **HttpSessionAttributeListener** otrzymują na wejściu zdarzenia typu **HttpSessionBindingEvent**.
A metody interfejsu **HttpSessionActivationListener** otrzymują na wejściu zdarzenia typu **HttpSessionEvent**.
Ponieważ klasy **HttpSessionEvent** i **HttpSessionBindingEvent** spełniały swoją rolę, nie było potrzeby dodawania do API serwetów dwóch kolejnych klas zdarzeń.

Przegląd interfejsów nasłuchujących zdarzeń związanych z sesjami oraz API obiektów zdarzeń





Tak, to niemal wierna kopia tabeli przedstawionej dwie strony wcześniej, więc staraj się tam nie zaglądać. Spróbuj przemyśleć znaczenie poszczególnych interfejsów nasłuchujących i wypełnij tabelę najlepiej jak potrafisz. Na egzaminie możesz się spodziewać przynajmniej dwóch (maksymalnie czterech) pytań poświęconych interfejsom nasłuchującym zdarzeń związanych z sesjami. Podczas wypełniania niniejszej tabeli powinieneś się posługiwać nie tylko swoją pamięcią, *ale także* logicznym rozumowaniem.

Scenariusz	Interfejs/metody nasłuchujące	Typ zdarzenia	Najczęściej implementowane przez
Chcemy wiedzieć, kiedy sesja jest tworzona.			<input type="checkbox"/> Klasa atrybutu <input type="checkbox"/> Jakaś <i>inna</i> klasa
Atrybut chce wiedzieć, kiedy jest przenoszony do nowej wirtualnej maszyny Javy.			<input type="checkbox"/> Klasa atrybutu <input type="checkbox"/> Jakaś <i>inna</i> klasa
Atrybut chce wiedzieć, kiedy jest zastępowany w ramach sesji.			<input type="checkbox"/> Klasa atrybutu <input type="checkbox"/> Jakaś <i>inna</i> klasa
Chcemy być informowani w momencie, w którym <i>cokolwiek</i> jest wiązane z sesją.			<input type="checkbox"/> Klasa atrybutu <input type="checkbox"/> Jakaś <i>inna</i> klasa

Wskazówka: istnieją tylko dwa typy obiektów zdarzeń.



**BAR
KAWOWY**

Examin próbny

1 Mamy dany kod:

```
10. public class MojServlet extends HttpServlet {
11.     public void doGet(HttpServletRequest req,
                        HttpServletResponse res)
12.         throws IOException, ServletException {
13.         // req.getSession().setAttribute("klucz", "wartosc");
14.         // req.getHttpSession().setAttribute("klucz", "wartosc");
15.         // ((HttpSession)req.getSession()).setAttribute("klucz", "wartosc");
16.         // ((HttpSession)req.getHttpSession()).setAttribute("klucz", "wartosc");
17.     }
18. }
```

Który wiersz (wiersze) można by wyjąć poza komentarz, nie powodując błędu kompilacji ani błędu w czasie wykonywania? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. Tylko wiersz 13.
- ☐ B. Tylko wiersz 14.
- ☐ C. Tylko wiersz 15.
- ☐ D. Tylko wiersz 16.
- ☐ E. Wiersz 13. lub wiersz 15.
- ☐ F. Wiersz 14. lub wiersz 16.

2 Zakładając, że klient NIE akceptuje ciasteczek, który z wymienionych poniżej mechanizmów zarządzania sesjami może zostać wykorzystany przez kontener WWW? (Zaznacz tylko jedną odpowiedź).

- ☐ A. Ciasteczka, ale NIE przepisywanie adresów URL.
- ☐ B. Przepisywanie adresów URL, ale NIE ciasteczka.
- ☐ C. Albo ciasteczka, albo przepisywanie adresów URL.
- ☐ D. Ani ciasteczka, ani przepisywanie adresów URL.
- ☐ E. Ciasteczka i mechanizm przepisywania adresów URL (stosowane jednocześnie).

-
- 3** Które zdania o obiektach **HttpSession** są prawdziwe? (Zaznacz wszystkie prawidłowe odpowiedzi).
- ☐ A. Sesja, dla której ustawiono limit czasowy równy **-1**, nigdy nie wygaśnie.
 - ☐ B. Sesja straci ważność w momencie, w którym użytkownik zamknie wszystkie okna swojej przeglądarki internetowej.
 - ☐ C. Sesja wygaśnie po przekroczeniu limitu czasowego zdefiniowanego przez kontener serwletu.
 - ☐ D. Sesja może być wprost unieważniona przez wywołanie metody **HttpSession.invalidateSession()**.
-
- 4** Które z wymienionych poniżej nazw NIE są typami zdarzeń nasłuchiowanych przez interfejsy należące do API J2EE 1.4? (Zaznacz wszystkie prawidłowe odpowiedzi).
- ☐ A. **HttpSessionEvent**
 - ☐ B. **ServletRequestEvent**
 - ☐ C. **HttpSessionBindingEvent**
 - ☐ D. **HttpSessionAttributeEvent**
 - ☐ E. **ServletContextAttributeEvent**
-
- 5** Które zdania na temat śledzenia sesji są prawdziwe? (Zaznacz wszystkie prawidłowe odpowiedzi).
- ☐ A. Przepisywanie adresów URL może być stosowane przez serwer w roli podstawowego mechanizmu śledzenia sesji.
 - ☐ B. SSL ma wbudowany mechanizm, który może być wykorzystywany przez kontener serwletu do uzyskiwania danych używanych do definiowania sesji.
 - ☐ C. Stosowanie ciasteczek do śledzenia sesji nie wiąże się z żadnymi ograniczeniami odnośnie do samych nazw tych ciasteczek.
 - ☐ D. Stosowanie ciasteczek do śledzenia sesji oznacza, że ciasteczko z identyfikatorem sesji musi się nazywać **JSESSIONID**.
 - ☐ E. Jeśli użytkownik wyłączył w swojej przeglądarce obsługę ciasteczek, kontener może podjąć decyzję o użyciu do śledzenia sesji użytkownika obiektu **javax.servlet.http.CookielessHttpSession**.

6 Mamy dany kod:

```
1. import javax.servlet.http.*;
2. public class MojSessionListener
    implements HttpSessionListener {
3.     public void sessionCreated() {
4.         System.out.println("Utworzono sesję");
5.     }
6.     public void sessionDestroyed() {
7.         System.out.println("Zniszczono sesję");
8.     }
9. }
```

Jaki błąd popełniono w powyższej klasie? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. Sygnatura metody w wierszu 3. NIE jest poprawna.
- ☐ B. Sygnatura metody w wierszu 6. NIE jest poprawna.
- ☐ C. Użyte polecenie importowania pakietu NIE zaimportuje interfejsu **HttpSessionListener**.
- ☐ D. Metody **sessionCreated()** i **sessionDestroyed()** NIE są jedynymi metodami definiowanymi przez interfejs **HttpSessionListener**.

7 Które zdania na temat atrybutów sesji są prawdziwe? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. Typem zwracanym przez metodę **HttpSession.getAttribute(String)** jest **Object**.
- ☐ B. Typem zwracanym przez metodę **HttpSession.getAttribute(String)** jest **String**.
- ☐ C. Atrybuty związane z obiektem sesji są dostępne dla wszystkich pozostałych serwetów należących do tego samego kontekstu (**ServletContext**) i dysponujących (jako elementy tej samej sesji) tym samym identyfikatorem żądania.
- ☐ D. Wywołanie metody **setAttribute("kluczA", "wartośćB")** dla obiektu **HttpSession**, który zawiera już jakąś wartość dla klucza **kluczA**, spowoduje wygenerowanie odpowiedniego wyjątku.
- ☐ E. Wywołanie metody **setAttribute("kluczA", "wartośćB")** dla obiektu **HttpSession**, który zawiera już jakąś wartość dla klucza **kluczA**, spowoduje, że poprzednia wartość tego atrybutu zostanie automatycznie zastąpiona łańcuchem **wartośćB**.

8 Które z wymienionych poniżej interfejsów definiują metodę `getSession()`?
(Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. `ServletRequest`
- ☐ B. `ServletResponse`
- ☐ C. `HttpServletRequest`
- ☐ D. `HttpServletResponse`

9 Mając dany obiekt `s` reprezentujący sesję oraz następujący wiersz kodu:

`s.setAttribute(„klucz”, wartość);`

zdecyduj, które interfejsy nasłuchujące otrzymają sygnał o wprowadzonej zmianie.
(Zaznacz tylko jedną odpowiedź).

- ☐ A. Tylko `HttpSessionListener`
- ☐ B. Tylko `HttpSessionBindingListener`
- ☐ C. Tylko `HttpSessionAttributeListener`
- ☐ D. `HttpSessionListener` i `HttpSessionBindingListener`
- ☐ E. `HttpSessionListener` i `HttpSessionAttributeListener`
- ☐ F. `HttpSessionBindingListener` i `HttpSessionAttributeListener`
- ☐ G. Wszystkie trzy

10 Wiedząc, że `req` jest obiektem `HttpServletRequest`, które metody tego obiektu utworzą sesję w sytuacji, gdy taka sesja jeszcze nie istnieje? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. `req.getSession();`
- ☐ B. `req.getSession(true);`
- ☐ C. `req.getSession(false);`
- ☐ D. `req.createSession();`
- ☐ E. `req.getNewSession();`
- ☐ F. `req.createSession(true);`
- ☐ G. `req.createSession(false);`

11 Wiedząc, że **s** jest obiektem reprezentującym sesję, która zawiera dwa atrybuty (**mojAtr1** i **mojAtr2**), zdecyduj, które metody tego obiektu usuną oba atrybuty z reprezentowanej sesji. (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. `s.removeAllValues();`
- ☐ B. `s.removeAttribute("mojAtr1");`
`s.removeAttribute("mojAtr2");`
- ☐ C. `s.removeAllAttributes();`
- ☐ D. `s.getAttribute("mojAtr1", UNBIND);`
`s.getAttribute("mojAtr2", UNBIND);`
- ☐ E. `s.getAttributeNames(UNBIND);`

12 Które zdania na temat obiektów klasy **HttpSession** w środowiskach rozproszonych są prawdziwe? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. Kiedy sesja jest przenoszona z jednej wirtualnej maszyny Javy do innej, wszystkie atrybuty składowane w tej sesji są gubione.
- ☐ B. Kiedy sesja jest przenoszona z jednej wirtualnej maszyny Javy do innej, tylko właściwie zarejestrowane obiekty implementujące interfejs **HttpSessionBindingListener** są o tym informowane.
- ☐ C. Kiedy sesja jest przenoszona z jednej wirtualnej maszyny Javy do innej, tylko właściwie zarejestrowane obiekty implementujące interfejs **HttpSessionActivationListener** są o tym informowane.
- ☐ D. Kiedy sesja jest przenoszona z jednej wirtualnej maszyny Javy do innej, wartości atrybutów implementujących interfejs **java.io.Serializable** migrują do nowej wirtualnej maszyny wraz z tą sesją.

13 Które zdania na temat limitów czasowych sesji są prawdziwe? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. Deklaracje limitów czasowych sesji umieszczane w deskrypcji wdrożenia są wyrażane w sekundach.
- ☐ B. Deklaracje limitów czasowych sesji umieszczane w deskrypcji wdrożenia są wyrażane w minutach.
- ☐ C. Deklaracje limitów czasowych sesji umieszczane w kodzie aplikacji są wyrażane w sekundach.
- ☐ D. Deklaracje limitów czasowych sesji umieszczane w kodzie aplikacji są wyrażane w minutach.
- ☐ E. Deklaracje limitów czasowych sesji umieszczane w kodzie aplikacji są wyrażane w minutach, jak lub sekundach.

14 Wybierz fragmenty kodu serwletu, które odczytałyby z obiektu żądania wartość ciasteczka nazwanego "ORA_UID". (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. `String wartosc = request.getCookie("ORA_UID");`
- ☐ B. `String wartosc = request.getHeader("ORA_UID");`
- ☐ C.

```
javax.servlet.http.Cookie[] cookies =
    request.getCookies();
String nazwaC = null;
String wartosc = null;
if (cookies != null) {
    for (int i = 0; i < cookies.length; i++) {
        nazwaC = cookies[i].getName();
        if (nazwaC != null &&
            nazwaC.equalsIgnoreCase("ORA_UID")) {
            wartosc = cookies[i].getValue();
        }
    }
}
```
- ☐ D.

```
javax.servlet.http.Cookie[] cookies =
    request.getCookies();
if (cookies.length > 0) {
    String wartosc = cookies[0].getValue();
}
```

15 Której metody (lub metod) można użyć do wymuszenia na kontenerze informowania naszej aplikacji bezpośrednio przed wygaśnięciem sesji? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. `HttpSessionListener.sessionDestroyed`
- ☐ B. `HttpSessionBindingListener.valueBound`
- ☐ C. `HttpSessionBindingListener.valueUnbound`
- ☐ D. `HttpSessionBindingEvent.sessionDestroyed`
- ☐ E. `HttpSessionAttributeListener.attributeRemoved`
- ☐ F. `HttpSessionActivationListener.sessionWillPassivate`

16 Jak w kodzie serwletu należałoby użyć obiektu `HttpServletResponse` do dodania ciasteczka?

- ☐ A. `<context-param>`
 `<param-name>mojeCiasteczko</param-name>`
 `<param-value>wartoscCiasteczka</param-value>`
 `</context-param>`
- ☐ B. `response.addCookie("mojeCiasteczko", "wartoscCiasteczka");`
- ☐ C. `javax.servlet.http.Cookie newCook =`
 `new javax.servlet.http.Cookie("mojeCiasteczko","wartoscCiasteczka");`
 `// ... ustawia pozostałe właściwości obiektu ciasteczka`
 `response.addCookie(newCook);`
- ☐ D. `javax.servlet.http.Cookie[] cookies = request.getCookies();`
 `String nazwaC = null;`
 `if (cookies != null) {`
 `for (int i = 0; i < cookies.length; i++) {`
 `nazwaC = cookies[i].getName();`
 `if (nazwaC != null &&`
 `nazwaC.equalsIgnoreCase("mojeCiasteczko")) {`
 `out.println(nazwaC + ": " + cookies[i].getValue());`
 `}`
 `}`
 `}`
 `}`

17 Mamy dany kod:

```
13. public class ServletX extends HttpServlet {
14.     public void doGet(HttpServletRequest req, HttpServletResponse resp)
15.         throws IOException, ServletException {
16.         HttpSession sess = new HttpSession(req);
17.         sess.setAttribute("attr1", "value");
18.         sess.invalidate();
19.         String s = sess.getAttribute("attr1");
20.     }
21. }
```

Jaki otrzymamy wynik? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. Kompilacja zakończy się niepowodzeniem.
- ☐ B. Zmienna `s` będzie miała wartość `null`.
- ☐ C. Zmienna `s` będzie reprezentowała łańcuch `"value"`.
- ☐ D. Zostanie wygenerowany wyjątek `IOException`.
- ☐ E. Zostanie wygenerowany wyjątek `ServletException`.
- ☐ F. Zostanie wygenerowany wyjątek `IllegalStateException`.



BAR
KAWOWY

Egzamin próbny — odpowiedzi

1 Mamy dany kod:

```
10. public class MojServlet extends HttpServlet {
11.     public void doGet(HttpServletRequest req,
12.                        HttpServletResponse res)
13.         throws IOException, ServletException {
14.         // req.getSession().setAttribute("klucz", "wartosc");
15.         // req.getHttpSession().setAttribute("klucz", "wartosc");
16.         // ((HttpSession)req.getSession()).setAttribute("klucz", "wartosc");
17.         // ((HttpSession)req.getHttpSession()).setAttribute("klucz", "wartosc");
18.     }
```

(Specyfikacja serwetów 2.4, str. 59).

Który wiersz (wiersze) można by wyjąć poza komentarz, nie powodując błędu kompilacji ani błędu w czasie wykonywania? (Zaznacz wszystkie prawidłowe odpowiedzi).

☐ A. Tylko wiersz 13.

☐ B. Tylko wiersz 14.

☐ C. Tylko wiersz 15.

☐ D. Tylko wiersz 16.

☒ E. Wiersz 13. lub wiersz 15.

☐ F. Wiersz 14. lub wiersz 16.

— Odpowiedź E jest poprawna, ponieważ zarówno wiersz 13., jak i wiersz 15. zawiera prawidłowe wywołania metod. Rzutowanie do typu HttpSession NIE jest konieczne, ale odzwierciedla właściwe typy, zatem jest poprawne.

2 Zakładając, że klient NIE akceptuje ciasteczek, który z wymienionych poniżej mechanizmów zarządzania sesjami może zostać wykorzystany przez kontener WWW? (Zaznacz tylko jedną odpowiedź).

(Specyfikacja serwetów 2.4, str. 57).

☐ A. Ciasteczka, ale NIE przepisywanie adresów URL.

☒ B. Przepisywanie adresów URL, ale NIE ciasteczka.

☐ C. Albo ciasteczka, albo przepisywanie adresów URL.

☐ D. Ani ciasteczka, ani przepisywanie adresów URL.

☐ E. Ciasteczka i mechanizm przepisywania adresów URL (stosowane jednocześnie).

— Odpowiedź B jest poprawna, ponieważ w takim przypadku ciasteczka NIE MOGA być wykorzystywane, a technika przepisywania adresów NIE jest uzależniona od prawidłowej obsługi ciasteczek.

- 3** Które zdania o obiektach **HttpSession** są prawdziwe? (Zaznacz wszystkie prawidłowe odpowiedzi). (Specyfikacja serwletów 2.4, str. 59).
- ☒ A. Sesja, dla której ustawiono limit czasowy równy **-1**, nigdy nie wygaśnie.
 - ☐ B. Sesja straci ważność w momencie, w którym użytkownik zamknie wszystkie okna swojej przeglądarki internetowej. — Odpowiedź B jest niepoprawna, ponieważ w protokole HTTP nie istnieje jednoznaczny sygnał końca połączenia.
 - ☒ C. Sesja wygaśnie po przekroczeniu limitu czasowego zdefiniowanego przez kontener serwletu.
 - ☐ D. Sesja może być wprost unieważniona przez wywołanie metody **HttpSession.invalidateSession()**. — Odpowiedź D jest niepoprawna, ponieważ właściwą metodą wykorzystywaną w tej roli jest `invalidate()`.
-
- 4** Które z wymienionych poniżej nazw **NIE** są typami zdarzeń nasłuchiowanych przez interfejsy należące do API J2EE 1.4? (Zaznacz wszystkie prawidłowe odpowiedzi). (API)
- ☐ A. **HttpSessionEvent**
 - ☐ B. **ServletRequestEvent**
 - ☐ C. **HttpSessionBindingEvent**
 - ☒ D. **HttpSessionAttributeEvent** — Zdarzenia typu `HttpSessionBindingEvent` są wykorzystywane zarówno w klasach implementujących interfejs `HttpSessionBindingListener`, jak i klasach implementujących interfejs `HttpSessionAttributeListener`.
 - ☐ E. **ServletContextAttributeEvent**
-
- 5** Które zdania na temat śledzenia sesji są prawdziwe? (Zaznacz wszystkie prawidłowe odpowiedzi). (Specyfikacja serwletów 2.4, str. 57).
- ☒ A. Przepisywanie adresów URL może być stosowane przez serwer w roli podstawowego mechanizmu śledzenia sesji.
 - ☒ B. SSL ma wbudowany mechanizm, który może być wykorzystywany przez kontener serwletu do uzyskiwania danych używanych do definiowania sesji.
 - ☐ C. Stosowanie ciasteczek do śledzenia sesji nie wiąże się z żadnymi ograniczeniami odnośnie do samych nazw tych ciasteczek. — Odpowiedź C jest niepoprawna, ponieważ specyfikacja nakazuje stosowanie znacznika nazwanego `JSESSIONID` do śledzenia sesji.
 - ☒ D. Stosowanie ciasteczek do śledzenia sesji oznacza, że ciasteczko z identyfikatorem sesji musi się nazywać **JSESSIONID**.
 - ☐ E. Jeśli użytkownik wyłączył w swojej przeglądarce obsługę ciasteczek, kontener może podjąć decyzję o użyciu do śledzenia sesji użytkownika obiektu **javax.servlet.http.CookielessHttpSession**. — Odpowiedź E jest niepoprawna, ponieważ taka klasa w ogóle nie istnieje.

6 Mamy dany kod:

(Specyfikacja serwetów 2.4, str. 276).

```
1. import javax.servlet.http.*;
2. public class MySessionListener
    implements HttpSessionListener {
3.     public void sessionCreated() {
4.         System.out.println("Utworzono sesję");
5.     }
6.     public void sessionDestroyed() {
7.         System.out.println("Zniszczono sesję");
8.     }
9. }
```

Jaki błąd popełniono w powyższej klasie? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☒ A. Sygnatura metody w wierszu 3. NIE jest poprawna. — Odpowiedzi A i B są poprawne, ponieważ odpowiednie metody powinny otrzymywać na wejściu parametry typu `HttpSessionEvent`.
- ☒ B. Sygnatura metody w wierszu 6. NIE jest poprawna.
- ☐ C. Użyte polecenie importowania pakietu NIE zaimportuje interfejsu **HttpSessionListener**. — Odpowiedź C jest niepoprawna, ponieważ implementowany interfejs `HttpSessionListener` jest zdefiniowany właśnie w zaimportowanym pakiecie.
- ☐ D. Metody **sessionCreated()** i **sessionDestroyed()** NIE są jedynymi metodami definiowanymi przez interfejs **HttpSessionListener**. — Odpowiedź D jest niepoprawna, ponieważ są to jedyne dwie metody tego interfejsu.

7 Które zdania na temat atrybutów sesji są prawdziwe? (Zaznacz wszystkie prawidłowe odpowiedzi).

(Specyfikacja serwetów 2.4, str. 59).

- ☒ A. Typem zwracanym przez metodę **HttpSession.getAttribute(String)** jest **Object**.
- ☐ B. Typem zwracanym przez metodę **HttpSession.getAttribute(String)** jest **String**. — Odpowiedź B jest niepoprawna, ponieważ zwracanym typem jest **Object**.
- ☒ C. Atrybuty związane z obiektem sesji są dostępne dla wszystkich pozostałych serwetów należących do tego samego kontekstu (**ServletContext**) i dysponujących (jako elementy tej samej sesji) tym samym identyfikatorem żądania.
- ☐ D. Wywołanie metody **setAttribute("kluczA", "wartośćB")** dla obiektu **HttpSession**, który zawiera już jakąś wartość dla klucza **kluczA**, spowoduje wygenerowanie odpowiedniego wyjątku. — Odpowiedź D jest niepoprawna, ponieważ wywołanie tej metody po prostu zastąpi istniejącą wartość.
- ☒ E. Wywołanie metody **setAttribute("kluczA", "wartośćB")** dla obiektu **HttpSession**, który zawiera już jakąś wartość dla klucza **kluczA**, spowoduje, że poprzednia wartość tego atrybutu zostanie automatycznie zastąpiona łańcuchem **wartośćB**.

8 Które z wymienionych poniżej interfejsów definiują metodę `getSession()`? (Specyfikacja serwetów 2.4, str. 243).
(Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. `ServletRequest`
- ☐ B. `ServletResponse`
- ☒ C. `HttpServletRequest`
- ☐ D. `HttpServletResponse`

9 Mając dany obiekt `s` reprezentujący sesję oraz następujący wiersz kodu: (Specyfikacja serwetów 2.4, str. 80).

`s.setAttribute(„klucz”, wartość);`

zdecyduj, które interfejsy nasłuchujące otrzymają sygnał o wprowadzonej zmianie.
(Zaznacz tylko jedną odpowiedź).

- ☐ A. Tylko `HttpSessionListener`
- ☐ B. Tylko `HttpSessionBindingListener`
- ☐ C. Tylko `HttpSessionAttributeListener`
- ☐ D. `HttpSessionListener` i `HttpSessionBindingListener`
- ☐ E. `HttpSessionListener` i `HttpSessionAttributeListener`
- ☒ F. `HttpSessionBindingListener` i `HttpSessionAttributeListener`
- ☐ G. Wszystkie trzy

— Odpowiedź F jest prawidłowa, ponieważ obiekt implementujący interfejs `HttpSessionAttributeListener` jest informowany za każdym razem, gdy atrybut jest dodawany do sesji; co więcej, jeśli obiekt wartości implementuje interfejs `HttpSessionBindingListener`, także on otrzymuje stosowne powiadomienia.

10 Wiedząc, że `req` jest obiektem `HttpServletRequest`, które metody tego obiektu utworzą sesję (API) w sytuacji, gdy taka sesja jeszcze nie istnieje? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☒ A. `req.getSession();`
- ☒ B. `req.getSession(true);`
- ☐ C. `req.getSession(false);`
- ☐ D. `req.createSession();`
- ☐ E. `req.getNewSession();`
- ☐ F. `req.createSession(true);`
- ☐ G. `req.createSession(false);`

— Zarówno odpowiedź A, jak i odpowiedź B reprezentuje wywołanie metody tworzącej nową sesję, jeśli taka sesja jeszcze nie istnieje. Metoda `getSession(false)` zwróci wartość `null`, jeśli okaże się, że sesja jeszcze nie istnieje.

- 11 Wiedząc, że `s` jest obiektem reprezentującym sesję, która zawiera dwa atrybuty (`mojAtr1` i `mojAtr2`), zdecyduj, które metody tego obiektu usuną oba atrybuty z reprezentowanej sesji. (Zaznacz wszystkie prawidłowe odpowiedzi). (API)

- ☐ A. `s.removeAllValues();`
☒ B. `s.removeAttribute("mojAtr1");`
`s.removeAttribute("mojAtr2");`
☐ C. `s.removeAllAttributes();`
☐ D. `s.getAttribute("mojAtr1", UNBIND);`
`s.getAttribute("mojAtr2", UNBIND);`
☐ E. `s.getAttributeNames(UNBIND);`

— Odpowiedź B jest prawidłowa, ponieważ metoda `removeAttribute()` jest jedyną, która usuwa atrybuty z obiektu sesji (w jednym wywołaniu tej metody można usunąć tylko jeden atrybut).

- 12 Które zdania na temat obiektów klasy `HttpSession` w środowiskach rozproszonych są prawdziwe? (Zaznacz wszystkie prawidłowe odpowiedzi). (Specyfikacja serwetów 2.4, str. 60).

- ☐ A. Kiedy sesja jest przenoszona z jednej wirtualnej maszyny Javy do innej, wszystkie atrybuty składowane w tej sesji są gubione.
☐ B. Kiedy sesja jest przenoszona z jednej wirtualnej maszyny Javy do innej, tylko właściwie zarejestrowane obiekty implementujące interfejs `HttpSessionBindingListener` są o tym informowane.
☒ C. Kiedy sesja jest przenoszona z jednej wirtualnej maszyny Javy do innej, tylko właściwie zarejestrowane obiekty implementujące interfejs `HttpSessionActivationListener` są o tym informowane.
☒ D. Kiedy sesja jest przenoszona z jednej wirtualnej maszyny Javy do innej, wartości atrybutów implementujących interfejs `java.io.Serializable` migrują do nowej wirtualnej maszyny wraz z tą sesją.

— Odpowiedź A jest niepoprawna, ponieważ atrybuty serializowalne zostaną przeniesione wraz z obiektem sesji.

— Odpowiedź B jest niepoprawna, ponieważ atrybuty pozostaną związane z przeniesioną sesją.

- 13 Które zdania na temat limitów czasowych sesji są prawdziwe? (Zaznacz wszystkie prawidłowe odpowiedzi). (API)

- ☐ A. Deklaracje limitów czasowych sesji umieszczane w deskrytorze wdrożenia są wyrażane w sekundach.
☒ B. Deklaracje limitów czasowych sesji umieszczane w deskrytorze wdrożenia są wyrażane w minutach.
☒ C. Deklaracje limitów czasowych sesji umieszczane w kodzie aplikacji są wyrażane w sekundach.
☐ D. Deklaracje limitów czasowych sesji umieszczane w kodzie aplikacji są wyrażane w minutach.
☐ E. Deklaracje limitów czasowych sesji umieszczane w kodzie aplikacji są wyrażane w minutach, lub w sekundach.

— W elemencie `<session-timeout>` deskryptora wdrożenia można definiować limit czasowy wyłącznie w minutach, natomiast metoda `setMaxInactiveInterval()` obiektu `HttpSession` oferuje możliwość definiowania limitu tylko w sekundach.

14 Wybierz fragmenty kodu serwletu, które odczytałyby z obiektu żądania wartość ciasteczka nazwanego "ORA_UID". (Zaznacz wszystkie prawidłowe odpowiedzi). (API)

- ☐ A. `String wartosc = request.getCookie("ORA_UID");` — Odpowiedź A odwołuje się do metody, która nie istnieje.
- ☐ B. `String wartosc = request.getHeader("ORA_UID");`
- ☒ C. `javax.servlet.http.Cookie[] cookies = request.getCookies();`
`String nazwaC = null;`
`String wartosc = null;`
`if (cookies != null) {`
`for (int i = 0; i < cookies.length; i++) {`
`nazwaC = cookies[i].getName();`
`if (nazwaC != null &&`
`nazwaC.equalsIgnoreCase("ORA_UID")) {`
`wartosc = cookies[i].getValue();`
`}`
`}`
`}`
- W odpowiedzi C uzyskujemy tablicę ciasteczek za pomocą metody `getCookies()`, po czym szukamy ciasteczka z podaną nazwą.
- ☐ D. `javax.servlet.http.Cookie[] cookies = request.getCookies();`
`if (cookies.length > 0) {`
`String wartosc = cookies[0].getValue();`
`}`
- W odpowiedzi D odczytujemy tylko pierwszy element z tablicy wszystkich ciasteczek.

15 Której metody (lub metod) można użyć do wymuszenia na kontenerze informowania naszej aplikacji bezpośrednio przed wygaśnięciem sesji? (Zaznacz wszystkie prawidłowe odpowiedzi). (API)

- ☒ A. `HttpSessionListener.sessionDestroyed` — Odpowiedź A reprezentuje swoiste obejście problemu, ale jeśli dysponujemy klasą atrybutu, właśnie za pośrednictwem tej metody możemy być informowani o osiągnięciu limitu czasowego.
- ☐ B. `HttpSessionBindingListener.valueBound`
- ☒ C. `HttpSessionBindingListener.valueUnbound`
- ☐ D. `HttpSessionBindingEvent.sessionDestroyed` — Odpowiedź D: taka metoda nie istnieje.
- ☐ E. `HttpSessionAttributeListener.attributeRemoved` — Odpowiedź E: usuwanie atrybutu nie musi się wiązać z upłynięciem czasu życia sesji.
- ☐ F. `HttpSessionActivationListener.sessionWillPassivate` — Odpowiedź F: dezaktywacja sesji nie jest tożsama z jej wygaśnięciem.

16 Jak w kodzie serwletu należałoby użyć obiektu `HttpServletResponse` do dodania ciasteczka?

- ☐ A. `<context-param>`
- ```
<param-name>mojeCiasteczko</param-name>
<param-value>wartoscCiasteczka</param-value>
</context-param>
```
- ☐ B. `response.addCookie("mojeCiasteczko", "wartoscCiasteczka");`
- ☒ C. `javax.servlet.http.Cookie newCook =`  
`new javax.servlet.http.Cookie("mojeCiasteczko","wartoscCiasteczka");`  
`// ... ustawia pozostałe właściwości obiektu ciasteczka`  
`response.addCookie(newCook);`
- ☐ D. `javax.servlet.http.Cookie[] cookies = request.getCookies();`  
`String nazwaC = null;`  
`if (cookies != null) {`  
`for (int i = 0; i < cookies.length; i++) {`  
 `nazwaC = cookies[i].getName();`  
 `if (nazwaC != null &&`  
 `nazwaC.equalsIgnoreCase("mojeCiasteczko")) {`  
 `out.println(nazwaC + ": " + cookies[i].getValue());`  
 `}`  
`}`  
`}`

— Odpowiedź B jest niepoprawna, ponieważ metoda `addCookie()` otrzymuje na wejściu obiekt klasy `Cookie`, nie łańcuch.

— Odpowiedź D jest niepoprawna, ponieważ zawiera fragment kodu odczytującego wartość ciasteczka zamiast kodu tworzącego takie ciasteczko.

17 Mamy dany kod:

```
13. public class ServletX extends HttpServlet {
14. public void doGet(HttpServletRequest req, HttpServletResponse resp)
15. throws IOException, ServletException {
16. HttpSession sess = new HttpSession(req);
17. sess.setAttribute("attr1", "value");
18. sess.invalidate();
19. String s = sess.getAttribute("attr1");
20. }
21. }
```

Jaki otrzymamy wynik? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☒ A. Kompilacja zakończy się niepowodzeniem.
- ☐ B. Zmienna `s` będzie miała wartość `null`.
- ☐ C. Zmienna `s` będzie reprezentowała łańcuch `"value"`.
- ☐ D. Zostanie wygenerowany wyjątek `IOException`.
- ☐ E. Zostanie wygenerowany wyjątek `ServletException`.
- ☐ F. Zostanie wygenerowany wyjątek `IllegalStateException`.

— Odpowiedź A jest poprawna, ponieważ wiersz 16. zawiera błąd. Obiekt implementujący interfejs `HttpSession` uzyskujemy za pomocą metody `req.getSession()`.

## 7. Stosowanie technologii JSP

### **Być stroną JSP**



**Strona JSP staje się serwletem.** Serwletem, którego *nie* musisz tworzyć. Kontener przegląda kod Twojej strony JSP, tłumaczy go na kod źródłowy języka programowania Java i kompiluje tak przetłumaczony kod do postaci pełnowartościowej klasy serwletu Javy. Musisz jednak wiedzieć, co dzieje się w czasie konwertowania Twojego kodu strony JSP na kod Javy. W ramach kodu JSP *możesz* co prawda umieszczać kod Javy, ale czy na pewno powinieneś? A jeśli w kodzie swojej strony JSP nie umieścisz żadnych wyrażeń języka programowania Java, co *znajdzie* się w tym kodzie? Jak Twoja strona zostanie przetłumaczona na kod Javy? W niniejszym rozdziale przyjrzymy się sześciu różnym rodzajom elementów JSP, z których każdy ma określone zadanie i — jak się zapewne domyślasz — *unikatową składnię*. Nauczysz się, jak, dlaczego i co należy zamieszczać w kodzie stron JSP. Co ważne (być może nawet ważniejsze), nauczysz się także, czego *nie* należy pisać w ramach stron JSP.



### Model technologii JSP

- 6.1.** Zidentyfikuj, opisz i przygotuj kod JSP dla następujących elementów: (a) tekstu wzorca, (b) elementów skryptów (komentarzy, dyrektyw, deklaracji, skryptletów i wyrażeń), (c) akcji standardowych i akcji użytkownika oraz (d) elementów języka wyrażeń.
- 6.2.** Napisz kod JSP, który będzie wykorzystywał następujące dyrektywy: (a) *page* (z atrybutami *import*, *session*, *contentType* i *isELIgnored*), (b) *include* oraz (c) *taglib*.
- 6.3.** Napisz dokument JSP (dokument oparty na formacie XML), który będzie wykorzystywał prawidłową składnię.
- 6.4.** Opisz znaczenie poszczególnych zdarzeń składających się na sekwencję cyklu życia strony JSP: (1) tłumaczenie strony JSP, (2) kompilacja strony JSP, (3) wczytanie klasy, (4) utworzenie egzemplarza, (5) wywołanie metody *jspInit*, (6) wywołanie metody *\_jspService* oraz (7) wywołanie metody *jspDestroy*.
- 6.5.** Mając dany cel projektowy, napisz kod strony JSP, który będzie właściwie wykorzystywał niezbędne obiekty: (a) żądania, (b) odpowiedzi, (c) standardowego wyjścia, (d) sesji, (e) konfiguracji, (f) aplikacji, (g) strony, (h) kontekstu strony oraz (i) wyjątku.
- 6.6.** Skonfiguruj deskryptor wdrożenia deklarujący jedną lub więcej bibliotek znaczników, dezaktywujący język ewaluacyjny oraz dezaktywujący język skryptowy.
- 6.7.** Mając dany cel projektowy przewidujący dołączanie segmentu kodu JSP do innej strony, napisz kod JSP, który będzie wykorzystywał najwłaściwszy mechanizm włączania (dyrektywę *include* lub standardową akcję *jsp:include*).

### Uwagi wyjaśniające:

Większość wymienionych obok celów zostanie omówiona w tym rozdziale, wyjątkiem są szczegóły związane z akcjami standardowymi i akcjami użytkownika (c) oraz elementami języka wyrażeń (d) — oba te zagadnienia zostaną omówione w późniejszych rozdziałach.

Dyrektywa *page* zostanie omówiona jeszcze w tym rozdziale, natomiast dyrektywy *include* i *taglib* przedstawimy w dalszych rozdziałach.

Ten temat w ogóle nie zostanie poruszony w niniejszym rozdziale, szczegółów szukaj w rozdziale poświęconym wdrażaniu aplikacji.

Wszystkie punkty zostaną omówione w tym rozdziale. (Wskazówka: wymienione obok zagadnienia będą przedmiotem najbardziej prostych pytań na rzeczywistym egzaminie, przynajmniej dla tych Czytelników, którzy dokładnie zapoznają się z treścią rozdziału).

Wszystkie punkty zostaną omówione w niniejszym rozdziale, mimo że znaczenie większości tych obiektów nie powinno być Ci obce po lekturze dwóch poprzednich rozdziałów.

Omówimy w tym rozdziale wszystko poza deklarowaniem bibliotek znaczników — to zagadnienie zostanie szczegółowo przedstawione w rozdziale poświęconym bibliotece JSTL.

Ten temat w ogóle nie zostanie poruszony w tym rozdziale, szczegółów szukaj w rozdziale poświęconym bezskryptowym stronom JSP.

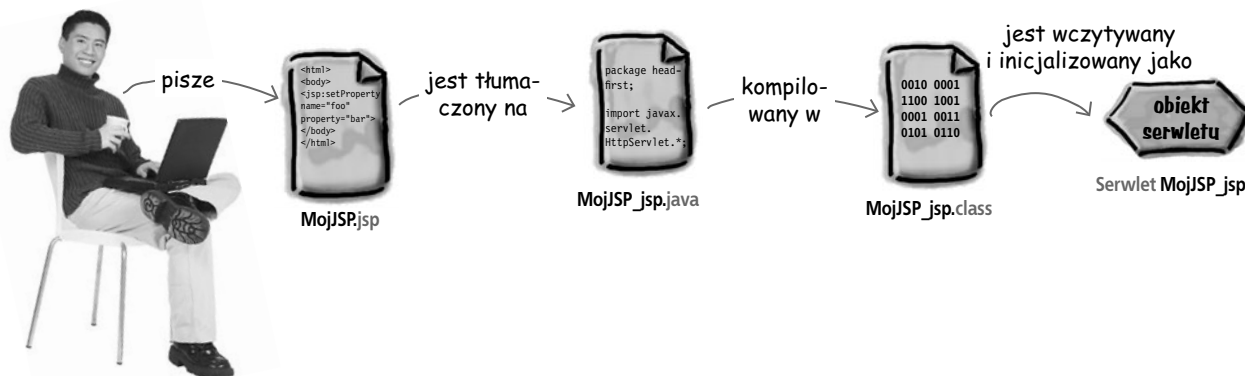
## Każda strona JSP ostatecznie staje się serwletem

Twoja strona JSP ostatecznie jest tłumaczona na pełnoprawny serwlet, który jest następnie wykonywany w ramach Twojej aplikacji internetowej. Można więc powiedzieć, że wszystkie takie serwlety są pisane *za Ciebie* przez wykorzystywany kontener WWW.

Kontener pobiera na wejściu to, co napisałeś w kodzie swojej strony JSP, **tłumaczy** ten kod na postać pliku źródłowego klasy Javy (.java), po czym **kompiluje** przetłumaczony kod na postać klasy serwletu Javy. Bezpośrednio potem odpowiedzialność za realizację żądań spada na sam serwlet, który jest wykonywany dokładnie w taki sam sposób jak serwlety napisane i skompilowane przez użytkownika. Innymi słowy, kontener wczytuje klasę serwletu, tworzy i inicjalizuje egzemplarz tej klasy, przydziela osobny wątek tego egzemplarza do każdego żądania i wywołuje metodę `service()` serwletu.

Najważniejsza kwestia analizowana w tym rozdziale jest dość prosta i sprowadza się do pytania: jaką rolę odgrywa kod Twojej strony JSP w ostatecznej klasie serwletu?

Innymi słowy, *gdzie* tak naprawdę elementy z Twojego kodu strony JSP są umieszczane w generowanym kodzie źródłowym serwletu?



Do podstawowych pytań, na które odpowiemy sobie w tym rozdziale, należą:

- Gdzie trafiają poszczególne fragmenty Twojego pliku JSP w generowanym kodzie źródłowym serwletu?
- Czy masz dostęp do „serwletowości” swojej strony JSP? Czy w kodzie strony JSP istnieje np. pojęcie obiektów `ServletConfig` i `ServletContext`?
- Jakie typy elementów możesz umieszczać w kodzie swoich stron JSP?
- Jaka jest składnia różnych elementów stron JSP?
- Jak wygląda cykl życia strony JSP i czy masz nad nim kontrolę?
- Jak poszczególne elementy strony JSP funkcjonują w ramach ostatecznej wersji serwletu?

# Opracowanie strony JSP, która wyświetli, ile razy została wywołana

Pauline chce użyć stron JSP w ramach swojej aplikacji internetowej — ma już *naprawdę* dosyć pisania kodu HTML w kodzie serwletu, w wywołaniach metody `println()` klasy `PrintWriter`.

Pauline zdecydowała się na opanowanie technologii JSP przez napisanie prostej strony generowanej dynamicznie, która będzie wyświetlała liczbę otrzymanych i zrealizowanych żądań. Pauline zdaje sobie sprawę, że może umieszczać tradycyjny kod Javy w kodzie stron JSP za pomocą tzw. *skryptletu* (ang. *scriptlet*), czyli po prostu kodu Javy osadzonego wewnątrz znacznika `<% ... %>`.

Wiem już, że mogę umieścić kod Javy w kodzie strony JSP, więc stworzę w mojej klasie `Licznik` metodę statyczną, która będzie obsługiwała statyczną zmienną licznika żądań i która będzie wywoływana przez moją stronę JSP...



ProstyLicznik.jsp

```
<html>
<body>
Licznik strony wynosi:
<%
 out.println(Licznik.getLiczba());
%>
</body>
</html>
```

Obiekt `out` jest w tym przypadku stosowany domyślnie. Wszystko, co znajduje się pomiędzy symbolami `<% a %>`, jest *skryptetem*, czyli po prostu zwykłym, tradycyjnym kodem Javy.

Licznik.java

```
package foo;

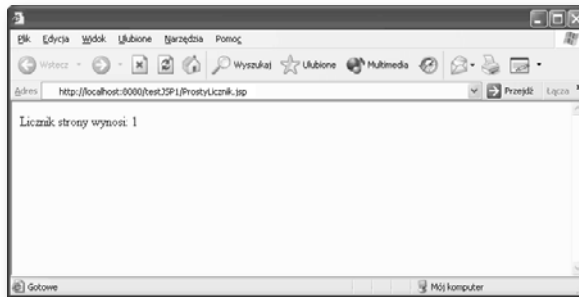
public class Licznik {
 private static int liczba;
 public static synchronized int getLiczba() {
 liczba++;
 return liczba;
 }
}
```

Prosta, tradycyjna klasa pomocnicza Javy.

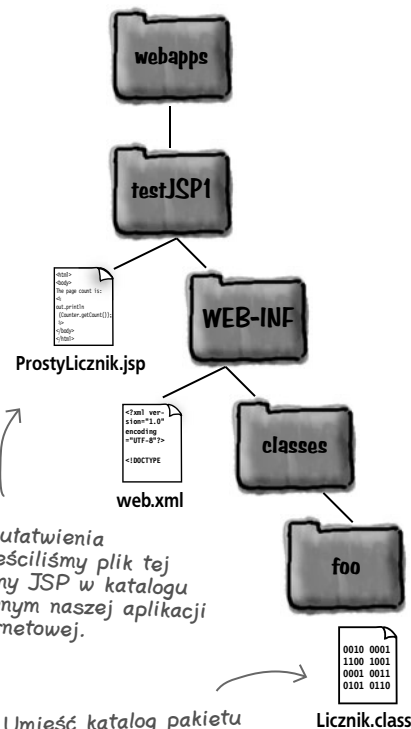
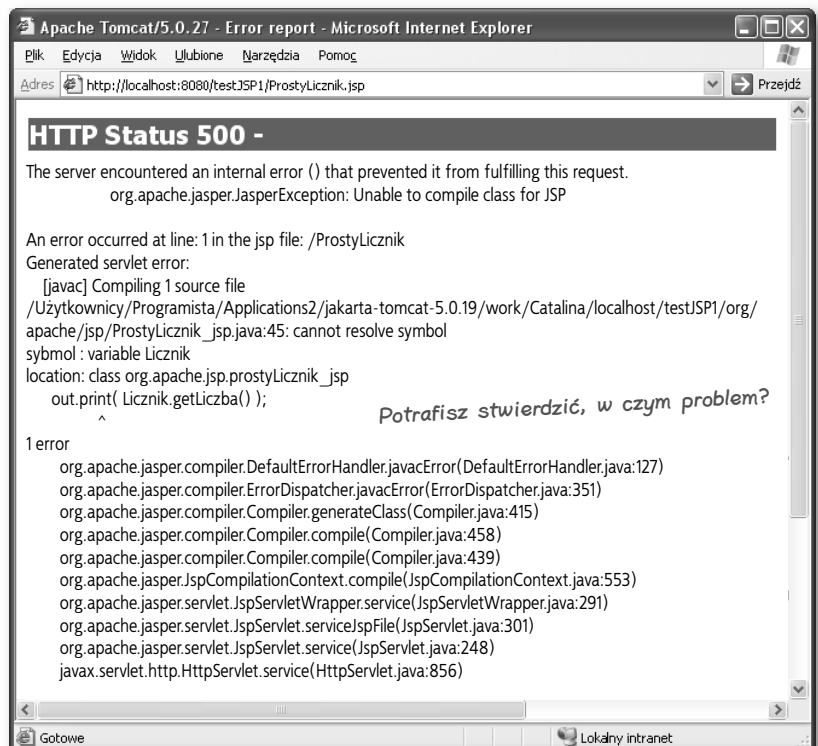
## Pauline wdraża i testuje swoją pierwszą stronę JSP

Wdrożenie i przetestowanie aplikacji internetowej, która składa się z jednej strony JSP i jednej klasy pomocniczej, jest trywialne. Jedynym z pozoru trudniejszym aspektem tej operacji jest upewnienie się, że klasa `Licznik` jest dostępna dla kodu strony JSP — okazuje się jednak, że wystarczy sprawdzić, czy klasa `Licznik` znajduje się w katalogu `WEB-INF/classes` danej aplikacji internetowej. Pauline wysłała do swojej strony JSP żądaniem bezpośrednim z przeglądarki internetowej: <http://localhost:8080/testJSP1/ProstyLicznik.jsp> ➔ *ProstyLicznik.jsp*.

### Czego oczekiwała:



### Co otrzymała:



Dla ułatwienia umieściliśmy plik tej strony JSP w katalogu głównym naszej aplikacji internetowej.

Umieść katalog pakietu i plik klasy w katalogu `WEB-INF/classes`, a wówczas klasa ta będzie dostępna dla wszystkich składników aplikacji internetowej.

Potrafisz stwierdzić, w czym problem?

## Strona JSP nie rozpoznała klasy Licznik

Klasa `Licznik` należy co prawda do pakietu `foo`, ale w kodzie strony JSP nie ma niczego, co potwierdzałoby tę przynależność. Z taką samą sytuacją będziemy mieli do czynienia w przypadku dowolnego innego kodu Javy — musimy wówczas stosować znaną regułę: importuj pakiety lub używaj w swoim kodzie w pełni kwalifikowanych nazw klas.

Domyślam się, że niezbędne jest użycie w kodzie strony JSP w pełni kwalifikowanej nazwy klasy. To zrozumiałe, przecież wszystkie strony JSP są tłumaczone przez kontener na zwykłe serwlety Javy. Z pewnością lepszym rozwiązaniem byłoby jednak umieszczenie poleceń importowania w kodzie JSP...



Licznik.java

```
package foo;
```

```
public class Licznik {
 private static int liczba;
 public static synchronized int getLiczba() {
 liczba++;
 return liczba;
 }
}
```

**Tak wyglądał nasz kod JSP:**

```
<% out.println(Licznik.getLiczba()); %>
```

**A tak ten kod powinien wyglądać:**

```
<% out.println(foo.Licznik.getLiczba()); %>
```

↑  
Teraz wszystko  
powinno działać.

MOŻESZ jednak umieszczać  
w kodzie stron JSP polecenia  
importowania pakietów...  
Wystarczy że użyjesz odpowiedniej  
dyrektywy.



## Użyj dyrektywy page do importowania pakietów

Dyrektywa jest jednym ze sposobów przekazywania specjalnych instrukcji dla kontenera na czas tłumaczenia strony JSP do postaci serwletu Javy. Istnieją trzy odmiany dyrektyw JSP: **page**, **include** oraz **taglib**. Dyrektywami `include` i `taglib` zajmiemy się w dalszych rozdziałach, na razie skupimy się wyłącznie na dyrektywie **page**, ponieważ właśnie za jej pomocą możemy *importować* niezbędne pakiety.

### Aby zaimportować pojedynczy pakiet:

```
<%@ page import="foo.*" %>
```

To jest dyrektywa page  
z atrybutem import.

(Zwróć uwagę na brak średnika  
na końcu tej dyrektywy).

```
<html>
```

```
<body>
```

```
Licznik strony wynosi:
```

```
<%
```

```
 out.println(Licznik.getLiczba());
```

```
%>
```

```
</body>
```

```
</html>
```

Skrypty są normalnymi  
wyrażeniami języka Java, zatem  
wszystkie polecenia w ramach  
skryptów muszą się kończyć  
średnikami!

### Aby zaimportować wiele pakietów:

```
<%@ page import="foo.*,java.util.*" %>
```

Użyj przecinka do oddzielania pakietów.  
Cudzysłowy otaczają całą listę pakietów!

Zauważyłeś różnice dzielące kod Javy wyświetlający wartość licznika od dyrektywy `page`?

Kod Javy znajduje się pomiędzy nawiasami ostrymi ze znakami procenta: `<%` oraz `%>`. Dyrektywa `page` dodaje jednak jeszcze jeden znak na początku elementu — symbol `@`!

Kiedy widzisz kod JSP rozpoczynający się od znaków `<%@`, od razu powinieneś wiedzieć, że masz do czynienia z dyrektywą. (Więcej szczegółów na temat dyrektywy `page` znajdziesz w dalszej części tej książki).

# Ale wtedy Kim wspomniał o „wyrażeniach”

Kiedy już Pauline myślała, że jej aplikacja jest gotowa, Kim zwrócił jej uwagę na skryptlet z wywołaniem metody `out.println()`. Przecież to JSP! Jedną z podstawowych reguł tej technologii jest *unikanie* metody `println()`! Dlatego właśnie wymyślono element *wyrażenia* JSP, który automatycznie wyświetla wszystko, co umieścisz pomiędzy znacznikami.



### Kod skryptletu:

```
<%@ page import="foo.*" %>
<html>
<body>
Licznik strony wynosi:
<% out.println(Licznik.getLiczba()); %>
</body>
</html>
```

### Kod wyrażenia:

```
<%@ page import="foo.*" %>
<html>
<body>
Licznik strony wynosi:
<%= Licznik.getLiczba() %>
</body>
</html>
```

To wyrażenie jest krótsze — nie musimy wprost wywoływać metody `println()`...

Zwróciłeś uwagę na różnicę pomiędzy znacznikiem dla kodu skryptletu a znacznikiem dla wyrażenia? Kod skryptletu jest umieszczany pomiędzy nawiasami ostrymi ze znakiem procenta: `<%` oraz `%>`, natomiast wyrażenie dodaje jeszcze jeden znak do symbolu otwierającego element — znak równości (=).

Do tej pory zapoznaliśmy się z trzema różnymi typami elementów JSP:

Skryptlet: `<% %>`

Dyrektywa: `<%@ %>`

Wyrażenie: `<%= %>`

HALO! Jeśli chcesz nas pouczyć, jak udoskonalać kod stron JSP, powinieneś się PRZYNAJMNIEJ zapoznać ze składnią Javy... przecież na końcu tego wyrażenia brakuje cholernego średnika!



A gdzie średnik?



```
<%= Licznik.getLiczba() %>
```

## Wyrażenia są tłumaczone na argumenty wywołań metody `out.print()`

Innymi słowy, kontener pobiera wszystko, co wpiszesz pomiędzy symbolami `<%=` oraz `%>`, i umieszcza to w postaci argumentu metody wyświetlającej odpowiedź — domyślnego obiektu `out` klasy `PrintWriter`.

### Kiedy kontener widzi coś takiego:

```
<%= Licznik.getLiczba() %>
```

### Tłumaczy to na następujące wywołanie:

```
out.print(Licznik.getLiczba());
```

### Gdybyś w swoim wyrażeniu umieścił średnik:

```
<%= Licznik.getLiczba(); %>
```

### Popełniłbyś zasadniczy błąd, ponieważ oznaczałoby to, że:

```
out.print(Licznik.getLiczba());;
```



O rany!! Przecież to się nigdy nie skompiluje.

### **NIGDY nie kończ wyrażenia średnikiem!**

```
<%= wTymMiejscuNigdyNieUmieszczajŚrednika %>
```

```
<%= ponieważJestToArgumentMetodyPrint() %>
```

### Nie ma niemądrych pytań

**P:** Cóż, jeśli mam używać wyrażen ZAMIAST umieszczać wywołania metody `out.println()` w skrypcie, po co nam ten domyślny obiekt `out`?

**U:** Prawdopodobnie nie będziesz zmuszony do korzystania z domyślnej zmiennej `out` na poziomie kodu swojej strony JSP, ale nie można wykluczyć sytuacji, w której konieczne będzie jej przekazanie *gdzieś indziej...* do jakiegoś innego obiektu będącego częścią tej samej aplikacji internetowej, ale niemającego bezpośredniego dostępu do strumienia wyjściowego odpowiedzi na żądanie.

**P:** Co będzie, jeśli metoda wywołana w ramach wyrażenia nie zwróci żadnych danych wyjściowych?

**U:** Wystąpi błąd!! W ramach wyrażen nie możesz, NIE MOŻESZ używać metod z typem wyjściowym `void`. Kontener jest wystarczająco inteligentny, by stwierdzić, że *nie będzie miał czego wyświetlić, jeśli metoda użyta w wyrażeniu zwraca typ void!*

**P:** Dlaczego dyrektywa importowania pakietu rozpoczyna się od słowa „page”? Dlaczego używamy polecenia `<%@ page import ... %>` zamiast po prostu `<%@ import ... %>`?

**U:** Dobre pytanie! Zamiast budować obszerny zestaw rozmaitych dyrektyw, specyfikacja technologii JSP definiuje tylko trzy dyrektywy, które jednak mogą mieć swoje atrybuty. To, co nazywasz „dyrektywą import”, jest w rzeczywistości „atrybutem import dyrektywy page”.

**P:** A co z pozostałymi atrybutami dyrektywy page?

**U:** Pamiętaj, że dyrektywa `page` ma w założeniu służyć przekazywaniu kontenerowi informacji niezbędnych do przetłumaczenia kodu strony JSP na kod serwletu Javy. Poza atrybutem `import` będą nas interesowały takie atrybuty jak `session`, `contentType` oraz `isELIgnored` (do ich omawiania będziemy jeszcze wracali w dalszej części rozdziału).



Zdecyduj, które z poniższych wyrażen są poprawne, a które są błędne — swoje odpowiedzi uzasadnij. Do tej pory nie zdążyliśmy omówić wszystkich przykładów, zatem w kilku przypadkach musisz się zdać na swoją intuicję popartą wiedzą na temat interpretowania wyrażen przez kontener. (Odpowiedzi znajdziesz w dalszej części rozdziału, zatem zacznij wykonywać to ćwiczenie już TERAZ).

**Poprawne? (Sprawdź, czy jest, czy nie jest poprawne; jeśli nie, uzasadnij swoją odpowiedź).**

☐ `<%= 27 %>`

☐ `<%= ((Math.random() + 5)*2); %>`

☐ `<%= "27" %>`

☐ `<%= Math.random() %>`

☐ `<%= String s = "foo" %>`

☐ `<%= new String[3] %>`

☐ `<% = 42*20 %>`

☐ `<%= 5 > 3 %>`

☐ `<%= false %>`

☐ `<%= new Licznik() %>`

## Kim wreszcie ujawnia sensacyjną wiadomość...

NIE potrzebujesz nawet klasy Licznik... wszystkie niezbędne zadania można bez trudu zrealizować w samym kodzie JSP.



Hmmm... Wiem, że kod JSP jest tłumaczony na serwlet, więc może mogłabym zadeklarować zmienną liczba w skrypcie? Zmienna ta zostałaby wówczas przekształcona w zmienną serwletu. Czy takie rozwiązanie ma szansę powodzenia?



### Co spróbowała zrobić:

```
<html>
<body>
<% int liczba=0; %>
Licznik strony wynosi:
<%= ++liczba %>
</body>
</html>
```

### Czy da się to skompilować?

### Czy to w ogóle zadziała?

# Deklarowanie zmiennej w skrypciecie

Taka deklaracja zmiennej jest *poprawna*, ale nowa wersja strony JSP nie będzie działała tak, jakbyśmy tego oczekiwali.

### Co spróbowała zrobić:

```
<html>
<body>
<% int liczba=0; %>
Licznik strony wynosi:
<%= ++liczba %>
</body>
</html>
```

← Nie musimy niczego importować, zatem możemy pominąć dyrektywę `page`.

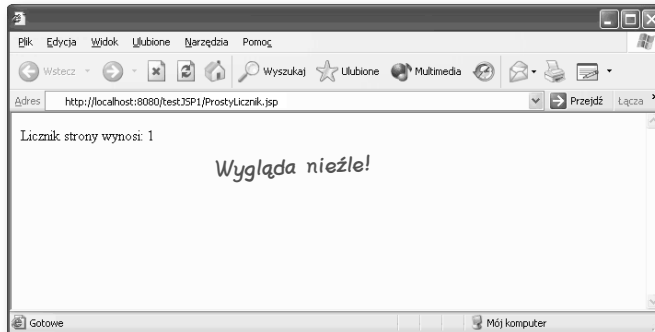
← Deklaracja zmiennej `liczba`.

← Zwiększa zmienną `liczba` i wyświetla jej wartość.

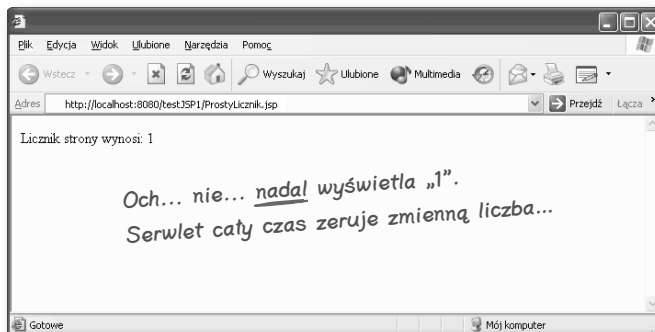
skryptlet →

wyrażenie →

### Co otrzymała, kiedy po raz pierwszy wysłała żądanie do strony:



### Co otrzymała, kiedy po raz drugi, trzeci i każdy następny wysłała żądanie do strony:



## Co NAPRAWDĘ dzieje się z Twoim kodem JSP?

Piszesz stronę JSP, ale strona ta ostatecznie *staje się* serwletem. Jedynym sposobem określenia, co naprawdę dzieje się w automatycznie wygenerowanym serwlecie, jest przeanalizowanie tego, co kontener zrobił z Twoim kodem JSP. Innymi słowy, musisz sprawdzić, jak kontener  *tłumaczy* kod Twojej strony JSP na kod serwletu Javy.

Kiedy już będziesz wiedział, gdzie poszczególne elementy JSP znalazły się w pliku klasy serwletu, z pewnością będzie Ci znacznie łatwiej planować strukturę swoich stron JSP.

Przedstawiony poniżej kod serwletu *nie* jest rzeczywistym kodem wygenerowanym przez kontener — znacznie go uprościliśmy, pozostawiając tylko jego kluczowe fragmenty. Wygenerowany przez kontener plik serwletu jest — co tu dużo mówić — dużo *brzydszy*. Rzeczywisty kod źródłowy otrzymany w procesie tłumaczenia strony JSP jest nieco trudniejszy do odczytania, ale i tak spróbujemy przeanalizować jego fragmenty kilka stron dalej. Na razie interesuje nas wyłącznie to, gdzie w klasie serwletu jest umieszczany nasz kod JSP.

### Ten kod JSP:                      staje się tym serwletem:

```
public class prostyLicznik_jsp extends JakisSpecjalnyHttpServlet {

 public void _jspService(HttpServletRequest request,
 HttpServletResponse response) throws java.io.IOException,
 ServletException {

 PrintWriter out = response.getWriter();
 response.setContentType("text/html");

 <html><body> → out.write("<html><body>");
 <% int liczba=0; %> → int liczba=0;
 Licznik strony wynosi: → out.write("Licznik strony wynosi:");
 <%= ++liczba %> → out.print(++liczba);
 </body></html> → out.write("</body></html>");

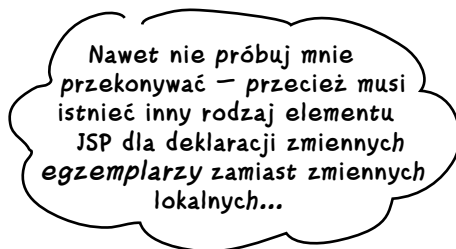
 }
}
```

Kontener umieszcza cały kod w uniwersalnej metodzie obsługującej żądania. Traktuj tę metodę jak procedurę zbiorczą dla metod doGet() i doPost().

**CAŁY kod skryptletu i wyrażenia trafia ostatecznie do metody obsługującej żądania.**

**Oznacza to, że zmienne zadeklarowane w skrypcie zawsze są zmiennymi LOKALNYMI!**

Uwaga: Jeśli chcesz się zapoznać z kodem serwletu wygenerowanym przez Tomcata, zajrzyj do katalogu `twójKatalogDomowyTomcata/work/Catalina/nazwaTwojegoSerwera/nazwaTwojejAplikacjiInternetowej/org/apache/jsp`. (Podkreślone nazwy będą oczywiście inne w zależności od Twojego systemu i Twojej aplikacji internetowej).



## Potrzebujemy innego elementu JSP...

Zadeklarowanie zmiennej *liczba* w skrypcie oznacza, że zmienna ta będzie ponownie inicjalizowana za każdym razem, gdy będzie wywoływana metoda obsługująca żądania klientów. To z kolei sugeruje, że **zmienna liczba jest zerowana z każdym żądaniem**. Musimy znaleźć sposób, by uczynić *liczba* zmienną *egzemplarza*.

Do tej pory zapoznaliśmy się z takimi strukturami jak dyrektywy, skryptlety i wyrażenia. **Dyrektywy** mają na celu przekazywanie specjalnych instrukcji dla kontenera, **skryptlety** są po prostu tradycyjnym kodem Javy, który jest w niezmienionej postaci kopiowany do wygenerowanej w serwlecie metody obsługującej żądania, natomiast wynik **wyrażenia** zawsze jest przekazywany jako argument wywołania metody `print()`.

Istnieje jednak jeszcze jeden element JSP, nazwany **deklaracją**.

```
<%! int liczba=0; %>
```

↑  
Umieść znak wykrzyknika (!) bezpośrednio za znakiem procenta (%).

↑  
To nie jest wyrażenie – w tym miejscu MUSISZ umieścić średnik!

Zadaniem deklaracji JSP jest deklarowanie składowych wygenerowanej klasy serwletu. **Przez składowe rozumiemy zarówno zmienne, jak i metody!** Innymi słowy, wszystko, co umieścisz pomiędzy znacznikiem `<%!` a znacznikiem `%>`, jest dodawane do tej klasy *poza* ciałem metody obsługującej żądania. Oznacza to, że możesz deklarować w tych znacznikach zarówno zmienne statyczne, jak i metody.

## Deklaracje JSP

Deklaracja JSP zawsze jest definiowana *wewnątrz* klasy, ale *poza* metodą obsługującą żądania (oraz poza wszystkimi innymi metodami). To proste — deklaracje służą do definiowania zmiennych statycznych, zmiennych egzemplarzy i metod. (Teoretycznie moglibyśmy nawet definiować inne składowe, włącznie z klasami wewnętrznymi, ale w 99,9999% przypadków będziemy używali deklaracji JSP do definiowania metod i zmiennych). Niniejszy kod rozwiązuje problem Pauline; od tej pory licznik będzie zwiększany za każdym razem, gdy klient zażąda danej strony.

### Deklaracja zmiennej

Ten kod JSP:

```
<html><body>
<%= int liczba=0; %>
Licznik strony wynosi:
<%= ++liczba %>
</body></html>
```

Staje się tym serwiletem:

```
public class prostyLicznik_jsp extends JakisSpecjalnyHttpServlet {
 int liczba=0;
 public void _jspService(HttpServletRequest request,
 HttpServletResponse response) throws java.io.IOException {
 PrintWriter out = response.getWriter();
 response.setContentType("text/html");
 out.write("<html><body>");
 out.write("Licznik strony wynosi:");
 out.print(++liczba);
 out.write("</body></html>");
 }
}
```

Tym razem zwiększamy wartość zmiennej egzemplarza zamiast zmiennej lokalnej.

### Deklaracja metody

Ten kod JSP:

```
<html>
<body>
<%= int podwojnaLiczba() {
 liczba = liczba*2 ;
 return liczba;
} %>
<%= int liczba=1; %>
Licznik strony wynosi:
<%= podwojnaLiczba() %>
</body>
</html>
```

Staje się tym serwiletem:

```
public class prostyLicznik_jsp extends JakisSpecjalnyHttpServlet {
 int podwojnaLiczba () {
 liczba = liczba*2;
 return liczba;
 }
 int liczba=1 ;
 public void _jspService(HttpServletRequest request,
 HttpServletResponse response) throws java.io.IOException {
 PrintWriter out = response.getWriter();
 response.setContentType("text/html");
 out.write("<html><body>");
 out.write("Licznik strony wynosi:");
 out.print(podwojnaLiczba());
 out.write("</body></html>");
 }
}
```

Ta metoda ma dokładnie taką postać, w jakiej ją zadeklarowałeś w swoim kodzie JSP.

Ponieważ jest to kod Javy, nie ma problemu odwołań do przodu (deklarowania zmiennej już PO jej użyciu w metodzie klasy).

# Czas się zapoznać z RZECZYWIŚCIE wygenerowanym serwletem

Do tej pory analizowaliśmy superuproszczoną wersję serwletu wygenerowanego przez kontener na podstawie kodu Twojej strony JSP. Oczywiście w czasie tworzenia aplikacji nie ma potrzeby przeglądania kodu generowanego przez kontener — z drugiej strony, taki kod może się bardzo przydać jako pomoc w *naucze* technologii JSP. Kiedy raz zobaczysz, co kontener zrobił z poszczególnymi elementami JSP, prawdopodobnie już nigdy nie będziesz musiał analizować plików źródłowych *.java* wygenerowanych przez kontener. Niektórzy producenci nawet *nie dopuszczają* możliwości przeglądania wygenerowanych plików źródłowych Javy i umieszczają w odpowiednich katalogach wyłącznie skompilowane pliki *.class*.

Nie przeraż się, kiedy znajdziesz w wygenerowanym kodzie nieznaną Ci elementy interfejsu API. Większość wykorzystywanych typów klas i interfejsów jest charakterystyczna tylko dla konkretnego producenta kontenera — należy do jego unikatowej implementacji i jako taka nie powinna Cię interesować.

## Co z Twoją stroną JSP robi kontener:

- Przegląda **dyrektywy** w poszukiwaniu informacji, które mogą być pomocne podczas tłumaczenia kodu JSP.
- Tworzy podklasę klasy `HttpServlet`.

W przypadku kontenera Tomcat 5 wygenerowany serwlet rozszerza klasę bazową:

```
org.apache.jasper.runtime.HttpJspBase
```

- Jeśli kod JSP zawiera **dyrektywę** `page` z atrybutem **import**, kontener zapisuje polecenia importowania na początku pliku klasy, bezpośrednio pod instrukcją określającą pakiet.

W przypadku kontenera Tomcat 5 instrukcja określająca pakiet (*o którą nie musimy się martwić*) ma postać:

```
package org.apache.jsp;
```

- Jeśli kod JSP zawiera **deklarację**, kontener zapisuje je w pliku klasy, zwykle pod deklaracją samej klasy, ale przed metodą obsługującą żądania. Kontener Tomcat 5 dodatkowo deklaruje na własne potrzeby jedną zmienną statyczną i jedną metodę klasową.
- Kontener buduje metodę **obsługującą żądania**. W rzeczywistości metoda ta nosi nazwę `_jspService()`. Za jej wywołanie odpowiada nadpisana metoda `service()` nadklasy serwletu, która automatycznie przekazuje niezbędne obiekty `HttpServletRequest` i `HttpServletResponse`. Częścią procesu budowy tej metody jest zadeklarowanie i inicjalizacja wszystkich **domyślnych obiektów**. (Więcej informacji na temat tych obiektów znajdziesz po przewróceniu strony).
- Łączy prosty, tradycyjny kod HTML (nazywany tekstem szablonu), **skrypty** oraz **wyrażenia** w jedną metodę obsługującą żądania, formatując i zapisując wszystko w obiekcie `PrintWriter` należącym do domyślnego obiektu reprezentującego odpowiedź.



**Relax** Problem generowania klas nie będzie odgrywał znaczącej roli na egzaminie.

Kod wygenerowany przez kontener prezentujemy wyłącznie po to, by ułatwić Ci zrozumienie procesu tłumaczenia stron JSP na kod serwletów. Nie musisz jednak ani znać szczegółów związanych z realizacją tego zadania przez poszczególnych producentów, ani wiedzieć, jak faktycznie wygląda kod generowany przez ich rozwiązania. Musisz jedynie znać zachowanie poszczególnych typów elementów (skryptletów, dyrektyw, deklaracji itp.) w aspekcie ich roli w wygenerowanym serwlecie. Powinieneś na przykład wiedzieć, że Twój skryptlet może się odwoływać do obiektów domyślnych, a więc musisz dysponować wiedzą na temat typów dostępnych w interfejsie API serwletów. **NIE** musisz jednak znać technik wykorzystywanych do udostępniania tych obiektów w kodzie aplikacji.

Jedynymi elementami procesu generowania kodu, z którymi z pewnością powinieneś się zapoznać przed egzaminem, są trzy metody cyklu życia JSP: `jspInit()`, `jspDestroy()` oraz `_jspService()`. (Ich dokładne omówienie znajdziesz w dalszej części tego rozdziału).

## Klasa wygenerowana przez Tomcat 5

```
package org.apache.jsp;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
```

Jeśli w kodzie JSP użyjesz dyrektyw `page` z atrybutem `import`, odpowiednie wiersze powinny się pojawić właśnie w tym miejscu (w tym przypadku nie użyto żadnych tego typu dyrektyw).

```
<html><body>
<%= int liczba=0; %>
Licznik strony wynosi:
<%= ++liczba %>
</body></html>
```

```
public class prostyLicznik_jsp extends org.apache.jasper.runtime.HttpJspBase
 implements org.apache.jasper.runtime.JspSourceDependent {
```

```
 int liczba=0;
 private static java.util.Vector _jspx_dependants;
```

```
 public java.util.List getDependants() {
 return _jspx_dependants;
 }
```

Kontener umieszcza w tym miejscu TWOJE deklaracje (zdefiniowane pomiędzy znacznikami `<%!` i `%>`) oraz wszelkie własne składowe pod deklaracją klasy.

```
 public void _jspService(HttpServletRequest request, HttpServletResponse response)
 throws java.io.IOException, ServletException {
```

```
 JspFactory _jspxFactory = null;
 PageContext pageContext = null;
 HttpSession session = null;
 ServletContext application = null;
 ServletConfig config = null;
 JspWriter out = null;
 Object page = this;
 JspWriter _jspx_out = null;
 PageContext _jspx_page_context = null;
```

Kontener deklaruje kilka własnych zmiennych lokalnych, w tym te reprezentujące „obiekty domyślne”, które mogą być wykorzystywane przez kod Twojej strony JSP (a więc np. `out` i `request`).

```
 try {
 _jspxFactory = JspFactory.getDefaultFactory();
 response.setContentType("text/html");
 pageContext = _jspxFactory.getPageContext(this, request, response, null, true, 8192, true);
 _jspx_page_context = pageContext;
 application = pageContext.getServletContext();
 config = pageContext.getServletConfig();
 session = pageContext.getSession();
 out = pageContext.getOut();
 _jspx_out = out;
 out.write("\r<html><body>\r");
 out.write("\rLicznik strony wynosi: \r");
 out.print(++liczba);
 out.write("\r</body></html>\r");
 } catch (Throwable t) {
 if (!(t instanceof SkipPageException)){
 out = _jspx_out;
 if (out != null && out.getBufferSize() != 0)
 out.clearBuffer();
 if (_jspx_page_context != null) _jspx_page_context.handlePageException(t);
 }
 } finally {
 if (_jspxFactory != null) _jspxFactory.releasePageContext(_jspx_page_context);
 }
 }
}
```

Teraz kontener próbuje właściwie zainicjalizować domyślne obiekty.

Próbuje też uruchomić i przekazać na standardowe wyjście kod HTML, kod skryptów i kod wyrażen Twojej strony JSP.

Oczywiście zawsze coś może pójść nie tak.

# Zmienna out nie jest jedynym obiektem domyślnym...

Kiedy kontener tłumaczy stronę JSP na kod serwletu, początek budowanej metody obsługującej żądania składa się z kilku deklaracji i operacji inicjalizujących **obiekty domyślne**.

Dzięki obiektom domyślnym możesz pisać swoje strony JSP z pełnym przekonaniem, że Twój kod będzie prawidłową częścią wygenerowanego później serwletu. Innymi słowy, możesz wykorzystywać serwletowość swojej aplikacji, mimo że *bezpośrednio* nie uczestniczysz w pisaniu klasy serwletu.

Wróć myślami do rozdziałów 4., 5. i 6. Potrafisz wymienić kilka najczęściej wykorzystywanych obiektów? W jaki sposób uzyskiwałeś z poziomu serwletu dostęp do parametrów inicjalizacji serwletu? Jak odczytywałeś parametry inicjalizacji kontekstu? Jakim sposobem Twój serwlet uzyskiwał sesję? Jak odczytywał parametry reprezentujące dane wpisane przez klienta w formularzu internetowym?

Istnieje przynajmniej kilka powodów, dla których Twoja strona JSP może potrzebować pewnych obiektów, które zazwyczaj stosuje się w kodzie serwletów. Wszystkie wspomniane już obiekty domyślne są w rzeczywistości odwzorowaniami składowych interfejsu API serwletów i stron JSP. Przykładowo, obiekt domyślny *request* jest referencją do obiektu **HttpServletRequest** przekazanego do metody obsługującej żądania przez kontener.

API	Obiekt domyślny
JspWriter	out
HttpServletRequest	request
HttpServletResponse	response
HttpSession	session
ServletContext	application
ServletConfig	config
Throwable	exception
PageContext	pageContext
Object	page

Które z tych obiektów reprezentują zasięg atrybutów żądania, sesji i aplikacji? (No dobrze, czuję, że to będzie dla Ciebie zbyt proste). Zwróć jednak uwagę na zupełnie NOWY, czwarty zasięg „poziom strony” — atrybuty tego zasięgu są przechowywane w obiekcie pageContext typu PageContext.

Ten obiekt domyślny jest dostępny wyłącznie dla specjalnych „stron błędów”. (Więcej informacji na ten temat znajdziesz w dalszej części tej książki).

Obiekt PageContext zawiera także inne obiekty domyślne, zatem jeśli przekazujesz referencję do tego obiektu któremuś ze swoich obiektów pomocniczych, taki docelowy obiekt pomocniczy może tę referencję wykorzystywać do uzyskiwania referencji do POZOSTAŁYCH obiektów domyślnych i — tym samym — atrybutów ze wszystkich zasięgów.

**P:** Jaka jest różnica pomiędzy udostępnianymi za pośrednictwem obiektu typu **HttpServletResponse** obiektami klas **JspWriter** i **PrintWriter**?

**U:** JspWriter nie należy do hierarchii obejmującej klasę PrintWriter, zatem jej obiekty nie mogą być stosowane w miejsce obiektów typu PrintWriter. Okazuje się jednak, że klasa JspWriter oferuje niemal identyczne metody jak PrintWriter (wzbogacone tylko o mechanizmy buforowania).



## Ćwiczenie

## BĄDŹ kontenerem



Każdy z przedstawionych poniżej listingów jest osobną stroną JSP. Twoim zadaniem jest określenie, co się stanie, kiedy kontener spróbuje przekształcić te strony w serwlety. Czy kontener w ogóle będzie w stanie przetłumaczyć Twój kod JSP na poprawny, możliwy do skompilowania kod serwletu Javy? Jeśli nie, wyjaśnij dlaczego. Jeśli tak, opisz zachowanie strony JSP podczas realizacji żądania klienta.

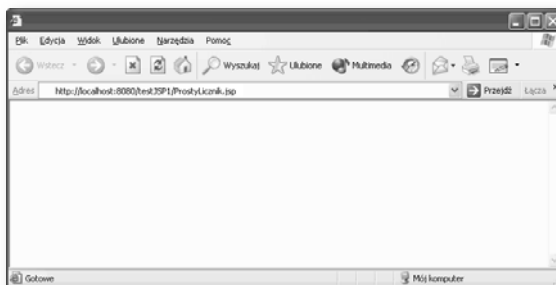
1 `<html><body>`  
Testowe skryptlety...  
`<% int y=5+x; %>`  
`<% int x=2; %>`  
`</body></html>`



2 `<%@ page import="java.util.*" %>`  
`<html><body>`  
Testowe skryptlety...  
`<% ArrayList list = new ArrayList();`  
`list.add(new String("foo")); %>`  
`<%= list.get(0) %>`  
`</body></html>`



3 `<html><body>`  
Testowe skryptlety...  
`<%! int x = 42; %>`  
`<% int x = 22; %>`  
`<%= x %>`  
`</body></html>`





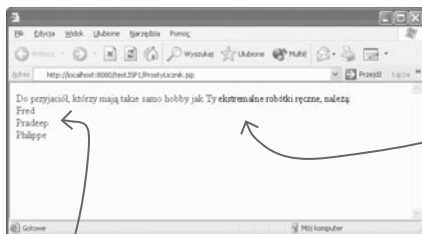
### Magnesiki egzaminu próbnego

Przestudiuj niniejszy scenariusz (i wszystkie pozostałe wskazówki zawarte na tej stronie), po czym rozmieść magnesy z kodem JSP w taki sposób, aby powstał prawidłowy plik generujący właściwe wyniki. Żadnego z dostępnych magnesów nie musisz używać więcej niż raz, nie wykorzystasz także wszystkich magnesów. W ćwiczeniu zakładamy, że istnieje serwlet (którego znajomość nie jest potrzebna do prawidłowego zrealizowania opisanego zadania), który pobiera pierwotne żądanie, wiąże atrybut z zasięgiem żądania i przekazuje je dalej do przygotowanej przez Ciebie strony JSP.

(Uwaga: nazwaliśmy to ćwiczenie „Magnesiki egzaminu próbnego” zamiast „Magnesiki z kodem”, ponieważ na egzaminie PEŁNO jest podobnych pytań wymagających przeciągania fragmentów kodu).

### Cel projektu

Stworzenie strony JSP, która wygeneruje następujący wynik:



Wyrażenie „ekstremalne robotki ręczne” pochodzi z parametru żądania (podanego przez użytkownika w formularzu HTML). Będziesz musiał odczytać ten parametr ze swojej strony JSP. Żądanie w pierwszej kolejności dociera do serwletu (który przekazuje je do Twojej strony JSP), co jednak w żaden sposób nie wpływa na sposób, w jaki odczytujesz wartość tego parametru na poziomie tej strony.

Te trzy nazwy pochodzą z atrybutu żądania nazwanego imioną będącego typu **ArrayList**. Będziesz musiał odczytać ten atrybut z obiektu żądania. Przyjmij, że serwlet ma dostęp do tego żądania i ustawił odpowiedni atrybut w zasięgu żądania.

### Formularz HTML

```
<html><body>
<form method="POST"
 action="StronaHobby.do">
 Wybierz swoje hobby:<p>
 <select name="hobby" size="1">
 <option>narciarstwo konne
 <option>ekstremalne robotki ręczne
 <option>nurkowanie alpejskie
 <option>randkowanie na czas
 </select>

 <center>
 <input type="SUBMIT">
 </center>
</form>
</body></html>
```

Ta nazwa wskazuje na serwlet, który ustawia atrybut żądania, po czym przekazuje cały obiekt żądania do napisanej przez Ciebie strony JSP.

### Ważne porady i wskazówki

- Atrybut żądania jest obiektem typu `java.util.ArrayList`.
- Domyślna zmienna dla obiektu klasy `HttpServletRequest` została nazwana *request* — możesz jej swobodnie używać w ramach skryptletów lub wyrażeń, ale *nie* wewnątrz dyrektyw czy deklaracji. Cokolwiek możesz zrobić z tym obiektem w kodzie serwletu, możesz zrobić także w kodzie strony JSP.
- Metoda serwletu wygenerowanego na podstawie strony JSP może przetwarzać parametry żądania, ponieważ — jak zapewne pamiętasz — Twój kod zostanie umieszczony wewnątrz metody obsługującej żądania w nowym serwlecie. Nie musisz się martwić o to, której z metod protokołu HTTP (GET czy POST) użyto w żądaniu klienta.

Dla ułatwienia umieściliśmy już kilka wierszy kodu na właściwym miejscu. Kod, który rozmieścisz na tej stronie JSP, **MUSI** pasować do porożstawianych wierszy. Kiedy skończysz, kod strony JSP powinien się bezproblemowo kompilować i generować wynik zgodny z przykładami zaprezentowanymi na poprzedniej stronie (PRZYJMIJ, że istnieje już działający serwet, który otrzymuje żądanie, ustawia atrybut żądania nazwany imiona oraz przekazuje to uzupełnione żądanie do tej strony JSP).

## STOP!

To ćwiczenie nie jest dobrowolne.  
Jest częścią lekcji poświęconej składni JSP!

Do przyjaciół, którzy mają takie samo hobby jak Ty (

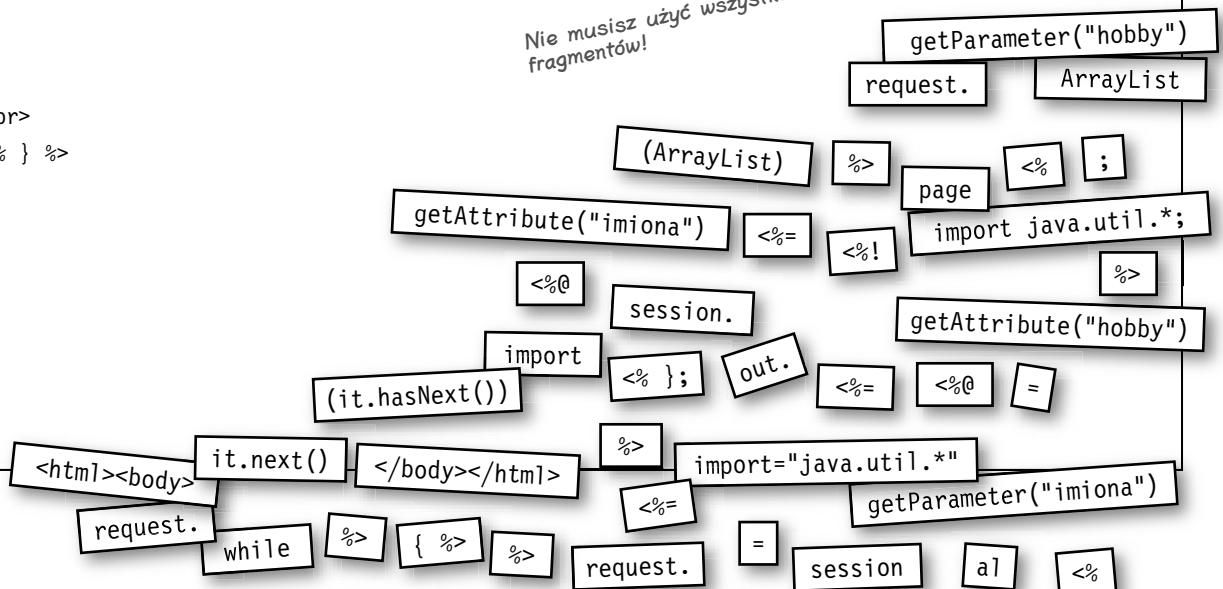
), należą: <br>

<% Iterator it = al.iterator();

<br>

<% } %>

Nie musisz użyć wszystkich fragmentów!



# BĄDŹ kontenerem — odpowiedzi



Punkt 2. jest oczywisty i działa bez najmniejszych kłopotów. Punkt 1. reprezentuje jeden z podstawowych problemów związanych z programowaniem w języku Java (stosowanie zmiennej lokalnej przed jej zadeklarowaniem); podobnie, punkt 3. przedstawia nieco inny problem, z którym borykają się początkujący programiści Javy — stosowanie zmiennej egzemplarza i zmiennej lokalnej z taką samą nazwą. Jak widać, jeśli spróbujesz przetłumaczyć kod JSP na kod serwletu Javy, będziesz mógł bez trudu stwierdzić, jaki będzie rzeczywisty wynik. Kiedy już Twój kod JSP znajdzie się wewnątrz serwletu, będzie zwykłym kodem Javy.

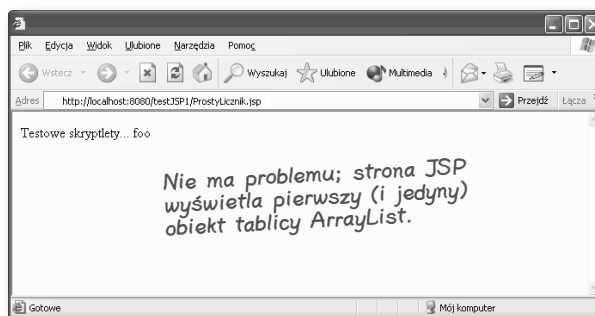
❶ `<html><body>`  
Testowe skryptlety...  
`<% int y=5+x; %>`  
`<% int x=2; %>`  
`</body></html>`

Ten kod się nie skompiluje! Efekt będzie dokładnie taki sam, jakbyś napisał metodę w postaci:

```
void foo() {
 int y = 5 + x ;
 int x = 2;
}
```

Próbujesz użyć zmiennej *x* PRZED jej zdefiniowaniem. Język Java nie zezwala na takie rozwiązania, a kontener nie ma obowiązku (i nie jest w stanie) przestawiania kodu w Twoich skryptletach, aby zapewnić możliwość jego skompilowania.

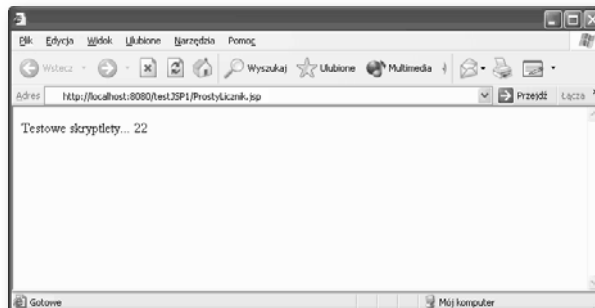
❷ `<%@ page import="java.util.*" %>`  
`<html><body>`  
Testowe skryptlety...  
`<% ArrayList list = new ArrayList();`  
    `list.add(new String("foo")); %>`  
`<%= list.get(0) %>`  
`</body></html>`



Nie ma problemu; strona JSP wyświetla pierwszy (i jedyny) obiekt tablicy ArrayList.

❸ `<html><body>`  
Testowe skryptlety...  
`<%! int x = 42; %>`  
`<% int x = 22; %>`  
`<%= x %>`  
`</body></html>`

Twój skryptlet deklaruje zmienną lokalną *x* (która ukrywa zmienną egzemplarza *x*), zatem jeśli chcesz wyświetlić wartość zmiennej egzemplarza *x* (42) zamiast wartości zmiennej lokalnej *x* (22), musisz zmienić dotychczasowe wyrażenie `<%= x %>` na `<%= this.x %>`.





## Odpowiedzi do magnesików z kodem

Jeśli Twoja odpowiedź wygląda trochę inaczej, ale nadal sądzisz, że opracowany przez Ciebie kod powinien zadziałać — po prostu go wypróbuj w praktyce! Będziesz musiał przygotować serwet odczytujący z żądania dane formularza, ustawiający odpowiedni atrybut i przekazujący to żądanie do strony JSP.

```
<%@ page import="java.util.*" %>
```

Ze względu na użyte w kodzie klasy `ArrayList` i `Iterator` musimy zastosować dyrektywę importującą `page`.

```
<html><body>
```

Do przyjaciół, którzy mają takie samo hobby jak Ty (

```
<%= request.getParameter("hobby") %>
```

), należą: `<br>`

```
<% ArrayList al = (ArrayList) request.getAttribute("imiona"); %>
```

```
<% Iterator it = al.iterator();
```

Tutaj zacznij swój skryptlet...

```
while (it.hasNext()) { %>
```

i zakończ go w tym miejscu.

```
<%= it.next() %>
```

Użyj wyrażenia.

```


<% } %>
```

Zakończ blok pętli `while`! (Gdybyś o tym zapomniał, strona JSP w ogóle nie zostałaby skompilowana).

```
</body></html>
```

```
<%@ page import="java.util.*" %>
session.
out.
import java.util.*;
request.
= session.
getAttribute("hobby")
getParameter("imiona")
};
```

### Komentarz...

Tak, w swoim kodzie JSP możesz umieszczać komentarze. Jeśli jesteś programistą Javy z bardzo niewielkim doświadczeniem w świecie HTML-a, najprawdopodobniej w pierwszym odruchu, bez zastanowienia spróbujesz wpisać:

```
// to jest komentarz
```

Jeśli jednak użyjesz takiej konstrukcji poza skryptletem lub znacznikiem deklaracji, szybko się przekonasz, że Twój komentarz jest WYŚWIETLANY klientowi w ramach odpowiedzi. Innymi słowy, zastosowane przez Ciebie dwa ukośniki dla kontenera nie są niczym niezwykłym i są traktowane na równi ze standardowymi zwrotami „Witaj!” lub „Oto mój adres poczty elektronicznej:”.

W kodzie JSP można umieszczać dwa różne rodzaje komentarzy:

#### ➤ **<!-- Komentarz HTML-->**

Kontener przekazuje to wyrażenie bezpośrednio do klienta, gdzie przeglądarka interpretuje je jako standardowy komentarz HTML.

#### ➤ **<%-- Komentarz JSP --%>**

Komentarze w tej postaci są przeznaczone dla twórców stron internetowych i (podobnie jak zwykle komentarze Javy w plikach źródłowych tego języka) nie są kopiowane do tłumaczonych stron. Jeśli piszesz swoją aplikację w formie kodu strony JSP i chcesz tam umieścić komentarze na temat podejmowanych czynności (w podobny sposób jak programiści komentują swój kod w Javie), powinieneś stosować właśnie komentarze JSP. Jeśli jednak chcesz, aby Twoje komentarze były częścią generowanych dla klienta odpowiedzi HTML (ale niewidocznych dla użytkownika oglądającego wyłącznie wizualizację kodu HTML), używaj komentarzy HTML.



Zaostrz ołówek

### ROZWIĄZANIA

#### Poprawne i błędne wyrażenia

##### Poprawne?



```
<%= 27 %>
```

Wszystkie proste stałe są dopuszczalne.



```
<%= ((Math.random() + 5)*2); %>
```

NIE! Ten średnik nie ma racji bytu.



```
<%= "27" %>
```

Stała łańcuchowa jest w porządku.



```
<%= Math.random() %>
```

Tak, ta metoda zwraca liczbę typu double.



```
<%= String s = "foo" %>
```

NIE! nie możesz w tym miejscu deklarować zmiennej.



```
<%= new String[3] %>
```

Tak, ponieważ nowa tablica łańcuchów jest obiektem, a każdy obiekt można bez problemu przekazać do wywołania metody `println()`.



```
<% = 42*20 %>
```

NIE! Działanie arytmetyczne jest dopuszczalne, ale zwróć uwagę na spację pomiędzy znakiem procenta (%) a znakiem równości (=). Wyrażenie nie może się zaczynać od znaków `<%=`, musi to być ciąg `<%=`.



```
<%= 5 > 3 %>
```

Pewnie, wynikiem tego wyrażenia jest wartość logiczna, zatem zostanie wyświetlone "true".



```
<%= false %>
```

Wspominaliśmy już, że proste stałe są dopuszczalne.



```
<%= new Licznik() %>
```

Żaden problem. To tak, jakbyś użył w tym miejscu `String[]`..., efektem tego wyrażenia będzie wynik wykonania metody `toString()`.

## API dla generowanych serwletów

Kontener generuje na podstawie Twojej strony JSP klasę implementującą interfejs `HttpJspPage`. Interfejs `HttpJspPage` jest jedyną częścią API generowanych serwletów, z którą koniecznie musisz się zapoznać. Nie jest ważne, że np. w kontenerze Tomcat wygenerowany serwlet rozszerza klasę bazową:

```
org.apache.jasper.runtime.HttpJspBase
```

Musisz tylko pamiętać o istnieniu trzech kluczowych metod interfejsu `HttpJspPage`:

### ➤ `jspInit()`

Ta metoda jest wywoływana z poziomu metody `init()`. Możesz tę metodę nadpisać. (Potrafisz określić, jak można to zrobić?)

### ➤ `jspDestroy()`

Ta metoda jest wywoływana z poziomu metody `destroy()`. Także tę metodę możesz nadpisać.

### ➤ `_jspService()`

Ta metoda jest wywoływana z poziomu metody `service()` serwletu, co oznacza, że dla każdego żądania jest wykonywana w osobnym wątku. Kontener automatycznie przekazuje do tej metody niezbędne obiekty żądania i odpowiedzi. Tej metody nie możesz nadpisać! W rzeczywistości sam z tą metodą nie możesz zrobić NICZEGO (z wyjątkiem zapisania kodu JSP, który zostanie w tej metodzie umieszczony przez kontener), a sposób tłumaczenia i dopasowywania Twojego kodu JSP do odpowiedniej metody `_jspService()` jest całkowicie uzależniony od kontenera.



Oglądaj to!

**Zwróć uwagę na znak podkreślenia na początku metody `_jspService()`**

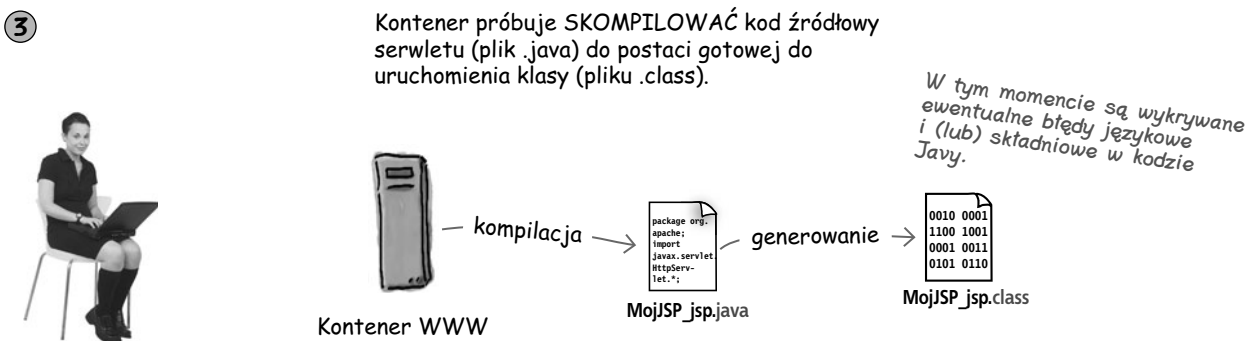
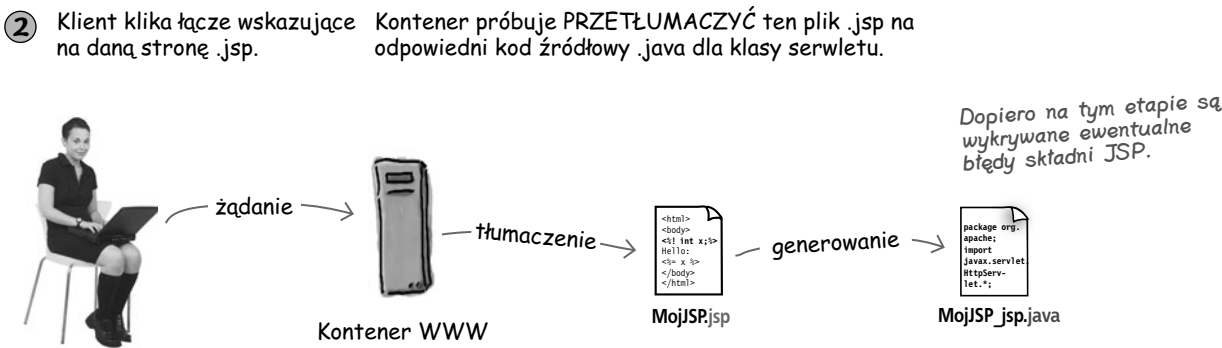
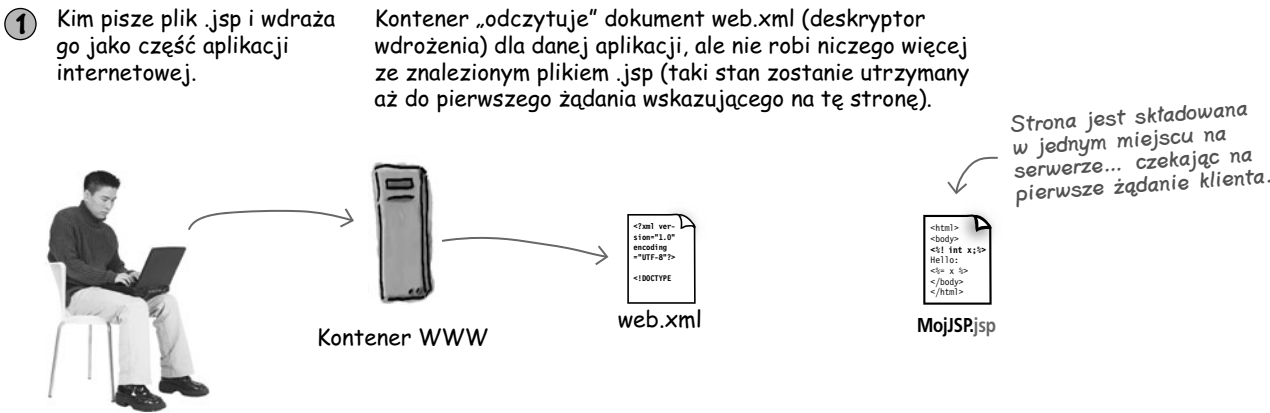
Znaku podkreślenia **NIE** ma na początku dwóch pozostałych metod: `jspInit()` i `jspDestroy()`. Możesz ten znak traktować jak wyraźny sygnał: „nie ruszaj!”.

Można zatem przyjąć, że brak znaku podkreślenia przed nazwą metody oznacza, że możesz ją nadpisać. Jeśli jednak nazwa metody **ROZPOCZYNA SIĘ** od znaku podkreślenia, **NIE** możesz próbować jej nadpisać!

# Cykl życia strony JSP

Piszesz plik **.jsp**.

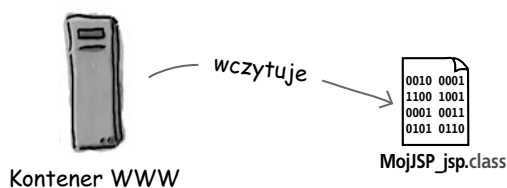
**Kontener** zapisuje plik **.java** dla serwletu, na który zostaje przetłumaczona Twoja strona JSP.



# Cykl życia strony JSP, ciąg dalszy...

4

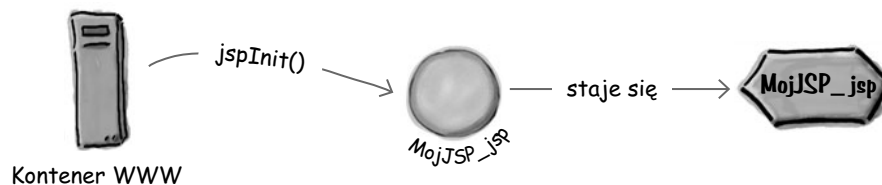
Kontener Wczytuje nowo wygenerowaną klasę serwletu.



5

Kontener tworzy egzemplarz serwletu i wymusza wywołanie należącej do tego serwletu metody `jspInit()`.

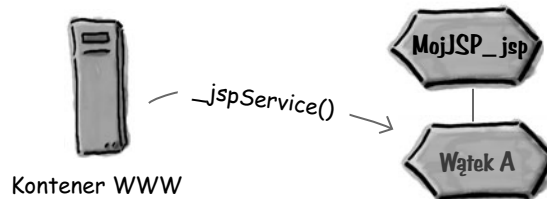
Obiekt jest teraz pełnowartościowym serwletem gotowym na przyjmowanie i realizację żądań klientów.



6

Kontener tworzy nowy wątek, którego zadaniem będzie obsługa bieżącego żądania klienta; bezpośrednio potem następuje wywołanie metody `_jspService()` serwletu.

Działania realizowane już po wywołaniu metody `_jspService()` niczym się nie różnią od tradycyjnej obsługi żądania przez serwlet Javy.



Serwlet ostatecznie odsyła odpowiedź do klienta (lub przekazuje żądanie dalej do innego komponentu aplikacji internetowej).

Wspaniale, jestem pod dużym wrażeniem. Nigdy bym nie przypuszczała, że realizacja żądania kierowanego do JSP wiąże się z koniecznością wykonania niemal tylu czynności co wywołanie metody komponentu EJB. Obawiam się tylko, że w tej sytuacji klient musi czekać, powiedzmy, pięć minut na całe to tłumaczenie, kompilowanie i inicjalizowanie.

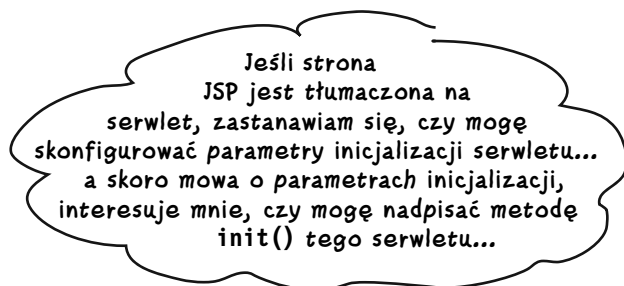


## Tłumaczenie i kompilowanie odbywa się tylko RAZ

Kiedy wdrażasz swoją aplikację internetową ze stroną JSP, proces tłumaczenia i kompilacji tej strony ma miejsce tylko raz w cyklu życia Twojego kodu JSP. Kiedy już strona zostanie przetłumaczona i skompilowana, jest traktowana jak zwykły serwlet. I tak jak każdy inny serwlet, kiedy odpowiedni kod zostanie wczytany i zainicjalizowany, jedyną czynnością, jaka pozostaje do wykonania w czasie realizacji żądania, jest stworzenie i przydzielenie wątku dla metody obsługującej żądanie. Rysunek z poprzednich dwóch stron należy więc traktować jak schemat funkcjonowania kontenera w odpowiedzi na *pierwsze żądanie* wskazujące na daną stronę JSP.

**P:** No dobrze, zatem mam rozumieć, że tylko pierwszy klient przesyłający żądanie wskazujące na daną stronę JSP musi czekać na przeprowadzenie całej procedury. **MUSI** jednak istnieć sposób takiego konfigurowania serwerów, aby procesy wstępnego tłumaczenia i kompilacji były realizowane z wyprzedzeniem... prawda?

**U:** Chociaż tylko pierwszy klient będzie musiał poczekać na przetłumaczenie i skompilowanie kodu strony JSP, większość producentów kontenerów UWZGLĘDNIĄ możliwość przeprowadzenia całego procesu tłumaczenia i kompilacji z wyprzedzeniem, aby nawet pierwsze żądanie do strony JSP było realizowane przez odpowiedni serwlet równie szybko jak wszystkie następne. Musisz jednak uważać — tego typu rozwiązania zależą wyłącznie od producentów kontenerów, zatem ich dostępność nie we wszystkich pakietach jest przesądzona. W specyfikacji technologii JSP (JSP 11.4.2) tylko WSPOMNIANO o propozycji protokołu dla prekompilacji JSP. Protokół ten przewiduje możliwość dołączania do żądania JSP łańcucha zapytania "`?jsp_precompile`", który powodowałby, że kontener (oczywiście pod warunkiem, że obsługiwałby ten protokół) od razu wykonałby procedury tłumaczenia i kompilacji, nie czekając na pierwsze żądanie pochodzące od prawdziwego klienta.



## Zaostrz ołówek

Dobrze przemyśl niniejsze pytania. Jeśli uznasz, że jest to konieczne, przekartkuj wcześniejsze strony (i rozdziały), ale nie zaglądaj na następną stronę aż do sformułowania swojej pełnej odpowiedzi.

Tak, **MOŻESZ** uzyskać dostęp do parametrów inicjalizacji serwletu z poziomu strony JSP. W związku z tym powstają naturalne pytania:

- 1) Jak można *odczytywać* wartości tych parametrów w kodzie JSP? (Wspaniała, niesamowita, godna uwielbienia wskazówka: odczytywanie parametrów przebiega dosyć podobnie jak w przypadku „normalnych” serwletów. Za pośrednictwem którego obiektu zazwyczaj odczytujesz wartości parametrów inicjalizacji serwletu? Czy obiekty te są dostępne dla Twojego kodu JSP?)
- 2) Jak (gdzie) należy *skonfigurować* parametry inicjalizacji serwletu?
- 3) Przypuśćmy, że *naprawdę* chcesz nadpisać metodę `init()`. Jak masz zamiar to zrobić? Czy istnieje jakieś inne rozwiązanie, które mogłoby przynieść identyczny efekt?

# Inicjalizacja Twojej strony JSP

Możesz umieścić polecenia inicjalizujące serwlet w kodzie swojej strony JSP, ale taka inicjalizacja będzie się *nieznacznie* różniła od tego, do czego są przyzwyczajeni programiści tradycyjnych serwletów.

## Konfigurowanie parametrów inicjalizacji serwletu

Parametry inicjalizacji serwletów dla strony JSP konfiguruje się niemal w taki sam sposób, w jaki konfigurowujemy odpowiednie parametry w zwykłym serwlecie. Jedyna różnica wiąże się z koniecznością dodania w deskrytorze wdrożenia elementu `<jsp-file>` wewnątrz znacznika `<servlet>`.

```
<web-app ...>
```

```
...
```

```
<servlet>
```

```
 <servlet-name>MojTestInicjal</servlet-name>
```

```
 <jsp-file>/TestInicjal.jsp</jsp-file>
```

```
 <init-param>
```

```
 <param-name>email</param-name>
```

```
 <param-value>ikickedbutt@wickedlysmart.com</param-value>
```

```
 </init-param>
```

```
</servlet>
```

```
<servlet-mapping>
```

```
 <servlet-name>MojTestInicjal</servlet-name>
```

```
 <url-pattern>/TestInicjal.jsp</url-pattern>
```

```
</servlet-mapping>
```

```
</web-app>
```


To jest jedyny wiersz odróżniający naszą konfigurację od konfiguracji tradycyjnego serwletu. Najprościej mówiąc, w tym wierszu stwierdzamy: „zastosuj całą zawartość znacznika `<servlet>` dla serwletu utworzonego na podstawie tej strony JSP...”

Decydując się na zdefiniowanie serwletu dla strony JSP musisz się liczyć z koniecznością zdefiniowania odwzorowania tego serwletu na daną stronę.

## Nadpisywanie metody `jspInit()`

Tak, to zupełnie proste. Jeśli zaimplementujesz metodę `jspInit()`, kontener wywoła ją na samym początku życia Twojej strony JSP w postaci serwletu. Metoda ta jest wywoływana z poziomu metody `init()`, zatem w momencie jej wykonywania serwlet ma dostęp do obiektów `ServletConfig` i `ServletContext`. Oznacza to, że z poziomu swojej metody `jspInit()` możesz bez problemu wywołać metody `getServletConfig()` i `getServletContext()`.

W niniejszym przykładzie wykorzystaliśmy metodę `jspInit()` do odczytania wartości parametru inicjalizacji serwletu (odpowiednio skonfigurowanego w deskrytorze wdrożenia) i użycia tej wartości do ustawienia atrybutu zasięgu aplikacji.

```
<%!  Nadpisujemy metodę jspInit() za pomocą deklaracji.
```

```
public void jspInit() {
```

```
 ServletConfig sConfig = getServletConfig();
```

```
 String adrEmail = sConfig.getInitParameter("email");
```

```
 ServletContext ctx = getServletContext();
```

```
 ctx.setAttribute("email", adrEmail);
```

```
}
```

```
%>
```

Jesteśmy w serwlecie, zatem możemy w prosty sposób wywołać odziedziczoną metodę `getServletConfig()`!

DOKŁADNIE to samo zrobilibyśmy w normalnym serwlecie.

Uzyskujemy referencję do obiektu `ServletContext` i ustawiamy atrybut w zasięgu aplikacji.

# Atrybuty w JSP


Przykład pokazany na poprzedniej stronie ilustruje sposób ustawiania z poziomu kodu JSP atrybutu w zasięgu aplikacji — wykorzystano tam deklarację metody nadpisującej oryginalną metodę `jspInit()`. W większości przypadków będziesz jednak używał jednego z czterech *obiektów domyślnych* do odczytywania i ustawiania atrybutów odpowiadających wspomnianym już czterem zasięgom dostępnym w technologii JSP.

Tak, czterem. Pamiętaj, że poza stosowanymi w serwletach standardowymi zasięgami żądania, sesji i aplikacji (kontekstu), w technologii JSP dodano jeszcze czwarty zasięg strony, udostępniany za pośrednictwem obiektu `pageContext`.

W większości sytuacji w ogóle nie będziesz potrzebował zasięgu strony (być może nawet zapomnisz o jego istnieniu), chyba że zdecydujesz się na tworzenie własnych znaczników — nie będziemy więc wchodzić w szczegóły aż do rozdziału poświęconego takim znacznikom.

	W serwlecie	W kodzie JSP (za pośrednictwem obiektów domyślnych)
Aplikacja	<code>getServletContext().setAttribute("foo", barOb);</code>	<code>application.setAttribute("foo", barOb);</code>
Żądanie	<code>request.setAttribute("foo", barOb);</code>	<code>request.setAttribute("foo", barOb);</code>
Sesja	<code>request.getSession().setAttribute("foo", barOb);</code>	<code>session.setAttribute("foo", barOb);</code>
Strona	Nie ma zastosowania!	<code>pageContext.setAttribute("foo", barOb);</code>

To jednak nie jest jeszcze cała historia! Specyfikacja JSP przewiduje także *inny* sposób odczytywania i zapisywania atrybutów w *dowolnym* zasięgu — wyłącznie za pośrednictwem domyślnego obiektu `pageContext`. Więcej informacji na ten temat znajdziesz po odwróceniu strony...



Oglądaj to!

Nie istnieje coś takiego jak zasięg „kontekstu”...  
mimo że atrybuty w zasięgu aplikacji są związane z obiektem ServletContext.

Stosowana konwencja nazewnicza może być myląca i skłonić Cię do myślenia, że atrybuty składowane w obiekcie ServletContext są... atrybutami zasięgu kontekstu. Takie atrybuty jednak w ogóle nie istnieją. Pamiętaj, że widząc słowo „Context”, powinieneś myśleć „aplikacja”. Istnieje pewna niespójność pomiędzy nazwami stosowanymi w serwletach a nazwami używanymi w kodzie stron JSP do odczytywania atrybutów zasięgu aplikacji — w serwlecie mówisz:

```
getServletContext().getAttribute("foo")
```

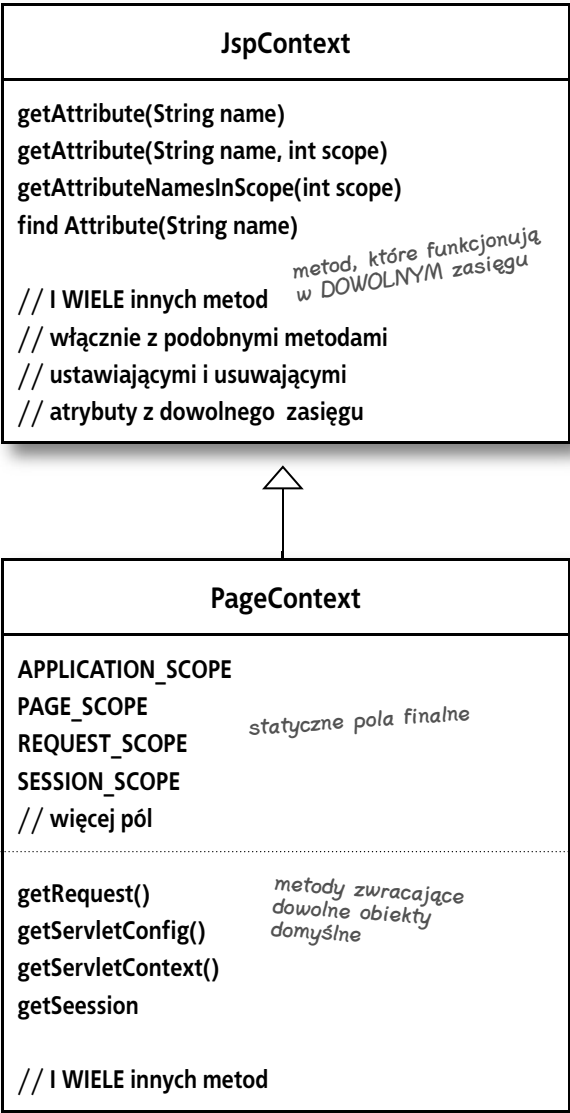
ale już w kodzie JSP mówisz:

```
application.getAttribute("foo")
```

## Używanie obiektu `PageContext` do obsługi atrybutów

Referencja do obiektu klasy `PageContext` może służyć do odczytywania atrybutów dowolnego zakresu, w tym atrybutów zasięgu strony związanych właśnie z obiektem `PageContext`.

Metody, które stworzono z myślą o pozostałych zasięgach, otrzymują na wejściu argument `int` określający właśnie interesujący nas zasięg. Chociaż metody operujące na atrybutach pochodzą z obiektu klasy `JspContext`, znajdziesz tam stałe reprezentujące zasięgi wewnątrz klasy `PageContext`.



## Przykłady stosowania obiektu `pageContext` do odczytywania i ustawiania atrybutów

### Ustawianie atrybutów zasięgu strony

```
<% Float jeden = new Float(42.5); %>
<% pageContext.setAttribute("foo", jeden); %>
```

### Odczytywanie atrybutów zasięgu strony

```
<%= pageContext.getAttribute("foo") %>
```

### Używanie obiektu `pageContext` do ustawiania atrybutów zasięgu sesji

```
<% Float dwa = new Float(22.5); %>
<% pageContext.setAttribute("foo", dwa, PageContext.SESSION_SCOPE); %>
```

### Używanie obiektu `pageContext` do odczytywania atrybutów zasięgu sesji

```
<%= pageContext.getAttribute("foo", PageContext.SESSION_SCOPE) %>
```

(znaczenie tego wyrażenia jest identyczne jak w przypadku wyrażenia `<%= session.getAttribute("foo") %>`)

### Używanie obiektu `pageContext` do odczytywania atrybutów zasięgu aplikacji

Adres poczty elektronicznej:

```
<%= pageContext.getAttribute("email", PageContext.APPLICATION_SCOPE) %>
```

W kodzie strony JSP znaczenie powyższego fragmentu kodu jest takie samo jak znaczenie wierszy:

Adres poczty elektronicznej:

```
<%= application.getAttribute("email") %>
```

### Używanie obiektu `pageContext` do odnajdywania atrybutów, których zasięg jest nam nieznany

```
<%= pageContext.findAttribute("foo") %>
```

*Gdzie należy szukać tego atrybutu?*

Gdzie metoda `findAttribute()` odnajduje wskazany atrybut? W pierwszej kolejności metoda ta przeszukuje kontekst strony, zatem jeśli w zasięgu tego kontekstu istnieje atrybut "foo", wówczas wywołanie metody `findAttribute(String name)` dla obiektu `PageContext` daje dokładnie taki sam efekt jak wywołanie metody `getAttribute(String name)` dla tego obiektu. Jeśli jednak w kontekście strony nie istnieje atrybut "foo", metoda `findAttribute()` rozpocznie przeszukiwanie pozostałych zasięgów, poczynwszy od najwęższego — innymi słowy, metoda ta będzie kolejno przeszukiwała zasięg żądania, sesji i wreszcie zasięg aplikacji. **Liczy się tylko ten zasięg, w którym atrybut z daną nazwą zostanie znaleziony jako pierwszy.**



Oglądaj to!

**Metoda `getAttribute(String)` obiektu `pageContext` dotyczy zasięgu strony**

*Istnieją DWIE przeciężone metody `getAttribute()`, które możesz wywoływać dla obiektu `pageContext`: wersja jednoargumentowa otrzymuje sam łańcuch, natomiast wersja dwuargumentowa otrzymuje zarówno łańcuch, jak i liczbę całkowitą. Działanie jednoargumentowej wersji metody `getAttribute()` nie różni się od funkcjonowania wszystkich pozostałych metod tego typu — odczytuje atrybuty ZWIĄZANE z danym obiektem `pageContext`. Wersja dwuargumentowa może być wykorzystywana do odczytywania atrybutów z DOWOLNEGO z czterech dostępnych zasięgów.*

# Skoro już poruszyliśmy ten temat, porozmawiajmy o trzech dyrektywach

Analizowaliśmy już dyrektywę wykorzystywaną do wprowadzania poleceń importujących do kodu klasy serwletu generowanego na podstawie naszej strony JSP. Była to dyrektywa *page* (jeden z trzech dostępnych typów dyrektyw) z atrybutem *import* (jednym z trzynastu atrybutów dyrektywy *page*). Przyjrzyjmy się teraz pozostałym dyrektywom i atrybutom, chociaż niektórych z nich nie będziemy szczegółowo omawiali w tym rozdziale, a pewne rozwiązania *w ogóle* nie zostaną przeanalizowane w tej książce, ponieważ są stosowane wyjątkowo rzadko.

## ① Dyrektywa *page*

```
<%@ page import="foo.*" session="false" %>
```

Definiuje właściwości specyficzne dla danej strony JSP, w tym schemat kodowania znaków, typ zawartości odpowiedzi dla tej strony oraz to, czy strona powinna mieć przydzielony domyślny obiekt sesji. Dyrektywa *page* może zawierać nie więcej niż trzynaście różnych atrybutów (w tym atrybut *import*), warto jednak pamiętać, że pytania egzaminacyjne będą dotyczyć tylko czterech atrybutów.

## ② Dyrektywa *taglib*

```
<%@ taglib tagdir="/WEB-INF/tags/cool" prefix="cool" %>
```

Definiuje biblioteki znaczników dostępne dla danej strony JSP. Nie wspominaliśmy jeszcze o stosowaniu własnych znaczników i standardowych akcji, zatem ich szczegółowa analiza w tym miejscu nie miałaby większego sensu. Na tym etapie możemy więc pominąć ten temat... przed nami jeszcze dwa rozdziały dotyczące bibliotek znaczników.

## ③ Dyrektywa *include*

```
<%@ include file="wickedHeader.html" %>
```

Definiuje tekst i kod dodawany do bieżącej strony w czasie jej tłumaczenia na kod serwletu. W ten sposób możemy konstruować fragmenty wielokrotnego użytku (takie jak standardowe nagłówki stron lub paski nawigacji), które mogą być następnie dodane do każdej strony aplikacji bez konieczności powielania odpowiedniego kodu we wszystkich stronach JSP.

**P:** Nie rozumiem... w nagłówku tej strony widnieje tekst „Skoro już poruszyliśmy ten temat...”, a ja nie widzę związku pomiędzy *dyrektywami* a obiektem *pageContext* i atrybutami.

**U:** Faktycznie, związek pomiędzy tymi zagadnieniami jest niewielki. Właśnie stwierdziliśmy, że nagłówek miał na celu podkreślenie naciąganego nieistniejącego przejścia pomiędzy niezwiązanymi ze sobą zagadnieniami. Mieliśmy nadzieję, że nikt nie zwróci na to uwagi, ale NIE... może mógłbyś to po prostu przemilczeć, zrobisz to dla nas?

## Atrybuty dyrektywy page

Z dostępnych trzynastu atrybutów dyrektywy page wymienionych w specyfikacji JSP 2.0 tylko *cztery* mogą być przedmiotem pytań egzaminacyjnych. NIE musisz zapamiętywać całej listy; spróbuj tylko poczuć, co tracisz z powodu zawężonego zakresu egzaminu. (Atrybut `isELIgnored` i parę atrybutów związanych z błędami przeanalizujemy bardziej szczegółowo w kolejnych rozdziałach).

### BYĆ MOŻE pojawi się na egzaminie

<b>import</b>	Definiuje polecenia importowania Javy, które są następnie dodawane do wygenerowanego kodu klasy serwletu. Niektóre operacje tego typu zostaną umieszczone w tym kodzie bez Twojego udziału (domyślnie): <code>java.lang</code> (to chyba oczywiste), <code>javax.servlet</code> , <code>javax.servlet.http</code> oraz <code>javax.servlet.jsp</code> .
<b>isThreadSafe</b>	Określa, czy wygenerowany serwlet powinien implementować interfejs <code>SingleThreadModel</code> , co — jak zapewne pamiętasz — jest wyjątkowo złym rozwiązaniem. Domyślną wartością tego atrybutu jest... "true", co oznacza mniej więcej tyle: „Moja aplikacja gwarantuje bezpieczeństwo przetwarzania wielowątkowego, zatem NIE muszę implementować interfejsu <code>SingleThreadModel</code> , o którym wiem, że ze swej natury jest zły”. Jedynym powodem używania tego atrybutu w dyrektywie page jest chęć zmiany jego wartości na "false", która oznacza, że chcemy wygenerować serwlet implementujący interfejs <code>SingleThreadModel</code> , <b>co w naszym przypadku jest wykluczone</b> .
<b>contentType</b>	Definiuje typ MIME (i opcjonalne kodowanie znaków) dla odpowiedzi JSP. <i>Zapewne wiesz, jaka jest wartość domyślna.</i>
<b>isELIgnored</b>	Określa, czy wyrażenia EL będą ignorowane w czasie tłumaczenia danej strony JSP. Do tej pory w ogóle nie wspominaliśmy o języku EL; jego szczegółowe omówienie znajdziesz w kolejnym rozdziale. Na razie musisz jedynie wiedzieć, że istnieje możliwość wybrania opcji ignorowania składni wyrażań EL na stronie JSP oraz że atrybut <code>isELIgnored</code> jest jednym z dwóch sposobów sygnalizowania takiego zamiaru kontenerowi.
<b>isErrorPage</b>	Określa, czy bieżąca strona reprezentuje stronę błędu <i>innej</i> strony JSP. Wartością domyślną tego atrybutu jest "false", jeśli jednak ustawimy wartość "true", strona będzie miała dostęp do domyślnego obiektu <i>wyjątku</i> (czyli po prostu referencji do obiektu klasy implementującej interfejs <code>Throwable</code> ). Jeśli natomiast pozostawimy domyślną wartość "false", obiekt wyjątku nie będzie dostępny dla danej strony JSP.
<b>errorPage</b>	Definiuje adres URL zasobu, do którego będą przekazywane te obiekty klasy implementującej interfejs <code>Throwable</code> , które nie zostały przechwycone i właściwie obsłużone. Jeśli wskażesz w tym miejscu jakąś stronę JSP, wówczas <i>ta strona</i> będzie miała ustawiony atrybut <code>isErrorPage="true"</code> w <i>swojej</i> dyrektywie page.

### NA PEWNO NIE pojawi się na egzaminie

<b>language</b>	Definiuje język skryptowy wykorzystywany w skryptletach, wyrażeniach i deklaracjach. Obecnie jedyną możliwą wartością tego atrybutu jest "java", a samo istnienie tego atrybutu wynika z przeświadczenia twórców technologii JSP o jej przyszłych rozszerzeniach, włącznie z rozwiązaniami wykorzystującymi inne języki programowania.
<b>extends</b>	Definiuje nadklasę dla klasy, na którą zostanie przetłumaczona dana strona JSP. Nie powinniśmy stosować tego atrybutu, chyba że RZECZYWIŚCIE wiemy, co chcemy w ten sposób osiągnąć — zmieniając wartość atrybutu <code>extends</code> , przykrywamy hierarchię klas udostępnianą przez kontener.
<b>session</b>	Określa, czy dana strona będzie miała przydzielony domyślny obiekt <i>sesji</i> . Wartością domyślną tego atrybutu jest "true".
<b>buffer</b>	Definiuje sposób, w jaki mechanizm buforowania będzie obsługiwany przez domyślny obiekt <i>out</i> (czyli referencję do obiektu klasy <code>JspWriter</code> ).
<b>autoFlush</b>	Określa, czy bufor danych wyjściowych będzie automatycznie opróżniany. Wartością domyślną tego atrybutu jest "true".
<b>info</b>	Definiuje łańcuch, który zostanie umieszczony na przetłumaczonej stronie i który będzie <i>zwracany</i> za pośrednictwem odziedziczonej metody <code>getServletInfo()</code> wygenerowanego serwletu.
<b>pageEncoding</b>	Definiuje schemat kodowania znaków dla danej strony JSP. Domyślnym schematem jest "ISO-8859-1" (chyba że atrybut <code>contentType</code> zdefiniował już jakieś inne kodowanie lub strona wykorzystuje składnię dokumentu XML).

To TAKI ładny rozdział  
z BARDZO sympatyczną prezentacją  
techniki umieszczania kodu Javy na stronie  
JSP, ale... hmmm... spójrz na notatkę,  
którą właśnie otrzymałam od szefa  
działu technicznego.



Międzydziałowa notatka od szefa działu technicznego

---

### UWAGA

Każdy pracownik przyłapany na stosowaniu w swoim kodzie JSP skryptletów, wyrażeń lub deklaracji będzie natychmiast zawieszany w swoich prawach i obowiązkach (włącznie z prawem do wynagrodzenia) aż do momentu sprawdzenia rzeczywistej winy programisty lub udowodnienia, że była to próba właściwego zagospodarowania kodu opracowanego przez INNEGO idiotę.

Jeśli jednak dochodzenie wykaże, że dany programista faktycznie odpowiada za to karygodne posunięcie, firma pójdzie dalej w swoich działaniach i zerwie współpracę z tym pracownikiem.

---

Rick Forester

Szef Działu Technicznego

---

"Pamiętaj: pojęcie "JA" nie istnieje w ZESPOLE"

„Pisz swój kod tak, jakby facet\*, który będzie go musiał czytać, był maniakałnym zabójcą znającym Twój adres."

[\*Uwaga dla działu kadr: używanego przez nas określenia "facet" nie należy traktować jako przejaw dyskryminacji płci.]

## Czy skrypty JSP można uznać za niebezpieczne?

Czy to prawda? Czy umieszczanie wszystkich tych wyrażeń Javy w kodzie stron JSP rzeczywiście *może* stanowić problem? Czy nie na tym miała polegać cała ta zwariowana IDEA stosowania stron JSP? Czy pisanie kodu Javy w czymś, co w gruncie rzeczy jest stroną HTML, jest czymś gorszym niż pisanie kodu HTML w klasie Javy?

Niektórzy ludzie sądzą (no dobrze, przyznajemy, że tak twierdzi *mnóstwo* osób, nie wyłączając członków zespołów pracujących nad specyfikacjami JSP i serwletów), że umieszczanie całego niezbędnego kodu Javy w stronie JSP jest *złą praktyką*.

Dlaczego? Wyobraź sobie, że zostałeś zatrudniony do zbudowania dużej witryny internetowej. Do Twojego zespołu należy niewielka, ale bardzo przydatna grupa programistów Javy zajmujących się oprogramowaniem działającym w tle aplikacji internetowych oraz znacznie liczniejsza grupa „projektantów stron internetowych” — artystów grafików oraz twórców stron, którzy używają Dreamweavera i Photoshopa do budowania stron internetowych z bajeczną oprawą wizualną. Nie są to *programiści* (no cóż, może z wyjątkiem tych, którzy nadal sądzą, że pisanie w HTML-u jest „kodowaniem”).



Początkujący aktorzy pracujący jako projektanci stron internetowych w oczekiwaniu na oferty ze strony show biznesu.

Dwa pytania: DLACZEGO zmuszasz nas do nauki czegoś, co nie jest zalecane, i JAKIE jest rozwiązanie alternatywne? Co jeszcze możemy, k\*\*\*\*, STOSOWAĆ w kodzie stron JSP poza HTML-em, skoro nie możemy tam umieszczać skryptletów, deklaracji i wyrażeń?



## W przeszłości NIE było rozwiązań alternatywnych

Oznacza to, że już teraz istnieją góry plików JSP wypchanych po brzegi kodem Javy, powciskanych we wszystkie możliwe szczeliny stron JSP, usadowionym wygodnie pomiędzy znacznikami skryptletów, wyrażeń i deklaracji. Ten kod już tam jest i nikt nie może tego w żaden sposób zmienić, ponieważ musiałby zmienić przeszłość. Zatem wiemy już, że umiejętność czytania i rozumienia tego typu elementów, a także konserwowania stron napisanych w oparciu o te elementy, jest absolutnie niezbędna (chyba że otrzymałeś misję gruntownego przeprojektowania stron JSP aplikacji internetowej).

Mówiąc między nami, sądzimy, że umieszczanie niewielkich fragmentów kodu Javy w stronach JSP w celach testowych i eksperymentalnych (np. dla sprawdzenia funkcjonowania niektórych mechanizmów serwera WWW), nadal nie jest niczym zdrożnym. W zdecydowanej większości przypadków nie należy jednak stosować tego typu rozwiązań w rzeczywistych, wykorzystywanych w praktyce stronach.

Powodem testowania wiedzy z tej dziedziny podczas egzaminu jest fakt, że odpowiednie rozwiązania *alternatywne* wciąż są stosunkowo nowe, zatem większość obecnie działających stron internetowych nadal funkcjonuje zgodnie ze „starą szkołą”.

**Na razie powinieneś jeszcze zachować umiejętności niezbędne do pracy z tego typu stronami!** Jednocześnie, kiedy nowe techniki oddzielania Javy od kodu stron internetowych osiągną masę krytyczną, wymienione na początku tego rozdziału cele prawdopodobnie zostaną wyeliminowane z zakresu materiału objętego egzaminem, a my wszyscy odetchniemy z ulgą na wieść o ostatecznej śmierci koncepcji Javy-w-JSP.

Dzisiaj żyjemy jednak w innych czasach.

(Uwaga dla zaniepokojonych rodziców i nauczycieli: użyte w dymku słowo rozpoczynające się od litery „k” poprzedzającej cztery gwiazdki wcale NIE jest tym, o czym myślicie. Jest to zwykłe słowo, co do którego po krótkich wahaniach doszliśmy do wniosku, że jest zbyt śmieszne, aby umieszczać je tutaj, nie rozpraszając uwagi czytelnika na tematy poboczne — stąd decyzja o jego zakropkowaniu. Ponieważ jest zabawne. Nie dlatego, że jest *złe*).

Ach, gdyby tylko istniał sposób stosowania w kodzie stron JSP prostych znaczników, które powodowałyby wywołania metod Javy, ale które nie wymagałyby umieszczania kodu Javy na stronie.



## EL — rozwiązanie, cóż, wszystkich problemów

Lub *prawie* wszystkich. Ale język EL z pewnością jest rozwiązaniem dwóch najważniejszych problemów związanych z umieszczaniem wyrażeń języka Java w kodzie stron JSP:

- 1) Projektanci stron WWW nie powinni być zmuszani do nauki programowania w Javie.
- 2) Kod Javy w JSP jest trudny do modyfikowania i konserwacji.

EL (język wyrażeń, od ang. *Expression Language*) stał się oficjalną częścią specyfikacji technologii JSP począwszy od wersji 2.0. Język EL niemal w każdym przypadku zapewnia znacznie prostszy sposób realizacji pewnych działań od tradycyjnych rozwiązań opartych na skryptletach i wyrażeniach.

Zapewne myślisz teraz: „A jeśli chcę, aby moja strona JSP wykorzystywała moje własne metody, jak mogę te metody zadeklarować i napisać, skoro nie mogę użyć Javy?”.

Ehhhh... tworzenie rzeczywistej funkcjonalności (kodu metody) *nie* jest celem języka EL. Celem tego języka jest jedynie zapewnienie prostszego sposobu *wywoływania* kodu Javy — jednak sam kod należy umieszczać *gdzieś indziej*, tj. w tradycyjnej klasie Javy będącej komponentem JavaBean, klasą z metodami statycznymi albo czymś, co programiści nazywają klasą obsługi znacznika (ang. *tag handler*). Innymi słowy, jeśli chcesz pracować w zgodzie z aktualnymi zaleceniami, nie powinieneś umieszczać kodu metody w ramach swojej strony JSP. Powinieneś zapisać kod metody Javy *gdzieś indziej* i *wywoływać* ją za pomocą poleceń języka EL.

# Ukradkowe spojrzenie na język EL

Cały kolejny rozdział poświęcimy językowi wyrażeń (EL), zatem wchodzenie w szczegóły na tym etapie nie miałyby większego sensu. Jedynym powodem, dla którego wspominamy o tym języku już teraz, jest istnienie jeszcze jednego typu elementów JSP (wraz z jego własną składnią) oraz to, że cele egzaminacyjne dla tego rozdziału obejmują umiejętność rozpoznawania wszystkiego, co może się znaleźć w kodzie strony JSP.

Wyrażenie języka EL ZAWSZE ma następującą postać: `${cokolwiek}`.

Innymi słowy, takie wyrażenie ZAWSZE jest umieszczane w nawiasach klamrowych i poprzedzane znakiem dolara (\$).

## To wyrażenie języka EL:

Proszę o kontakt: `${applicationScope.email}`

## Odpowiada następującemu wyrażeniu Javy:

Proszę o kontakt: `<%= application.getAttribute("email") %>`

Nie ma  
niemądrych pytań

**P:** Nie żebym się czepiał, ale nie jestem pewien, czy widzę jakąś uderzającą różnicę pomiędzy przedstawionym wyrażeniem języka EL a wyrażeniem Javy. To fakt, wyrażenie EL jest trochę krótsze, ale czy to wystarczający powód, aby wprowadzać całkowicie nowy język skryptowy i wywracać do góry nogami dotychczasowe techniki kodowania stron JSP?

**U:** Nie miałeś jeszcze okazji zapoznać się ze wszystkimi korzyściami wynikającymi ze stosowania języka EL. Różnice pomiędzy prezentowanymi podejściami staną się oczywiste w następnym rozdziale, kiedy poddamy te zagadnienia bardziej szczegółowej analizie. Musisz jednak pamiętać, że dla programisty Javy język EL NIEkoniecznie musi być wymarzoną drogą rozwoju umiejętności programistycznych. W praktyce dla programisty Javy wprowadzenie języka EL oznacza jedno: „jeszcze jedna rzecz (z własną składnią itp.), której muszę się nauczyć, mimo ZNAJOMOŚCI Javy...”.

Podobne odczucia wcale nie muszą być Twoim udziałem. Nauka języka EL *znacznie* łatwiej i szybciej przychodzi osobom, które nie programują w Javie. Z drugiej strony, dla programisty Javy wciąż prostszym rozwiązaniem jest pielęgnacja stron bezskryptowych.

Tak, musimy się jeszcze wiele nauczyć. Język wyrażeń EL co prawda nie w pełni zwalnia projektantów stron internetowych z konieczności poznawania nowych technologii, ale szybko się przekonasz, że język ten jest dla niedoświadczonych programistów znacznie bardziej intuicyjny i naturalny od tradycyjnej Javy. Na razie (w tym rozdziale) musisz się nauczyć wyłącznie *rozpoznawania* wyrażeń języka EL tam, gdzie będziesz miał z nimi do czynienia. Na tym etapie nie musisz się jeszcze martwić o zdolność oceny, czy dane wyrażenie jest poprawne — póki co interesuje nas jedynie to, czy potrafisz właściwie wskazać wyrażenia EL w kodzie strony JSP.

JAK Twoim zdaniem mogę  
zmusić moich programistów do  
zaprzestania stosowania elementów  
skryptów w kodzie ich stron  
JSP?



To proste — możesz  
umieścić w deskrytorze  
wdrożenia element, który wyłącza  
obsługę wszystkich elementów  
skryptowych!



W ten sposób wykluczamy możliwość  
stosowania elementów skryptowych  
we **WSZYSTKICH** stronach JSP danej  
aplikacji internetowej (ponieważ  
we wzorcu URL użyliśmy symbolu  
wieloznacznego \*.jsp).

To nie działa! Możliwość  
stosowania atrybutu  
isScriptingEnabled nie jest już  
przewidywana w specyfikacji  
technologii JSP!

## Stosowanie elementu < scripting-invalid >

To proste — możesz sprawić, że umieszczanie elementów  
skryptowych (skryptletów, wyrażeń Javy lub deklaracji)  
w kodzie stron JSP będzie nieprawidłowe. Wystarczy umieścić  
w deskrytorze wdrożenia znacznik <scripting-invalid>:

```
<web-app ...>
...
<jsp-config>
 <jsp-property-group>
 <url-pattern>*.jsp</url-pattern>
 <scripting-invalid>
 true
 </scripting-invalid>
 </jsp-property-group>
</jsp-config>
...
```

**Bądź ostrożny** — być może spotkałeś się w innych książkach  
lub artykułach z dyrektywą page, która także miała na celu  
wykluczanie możliwości stosowania elementów skryptowych  
w kodzie stron JSP. We *wstępnej* wersji specyfikacji 2.0  
rzeczywiście istniał następujący atrybut dyrektywy page:

```
<%@ page isScriptingEnabled="false" %>
```

**ale został usunięty z ostatecznej wersji specyfikacji!**

**Jedynym** sposobem wyłączania obsługi skryptów jest  
obecnie umieszczenie w deskrytorze wdrożenia elementu  
<scripting-invalid>.

# Możesz wymusić ignorowanie wyrażeń EL

Tak, wiemy już, że język EL jest jednym z tych osiągnięć ludzkości, które mogą uratować świat. Mogą jednak zaistnieć sytuacje, w których najlepszym rozwiązaniem będzie całkowite wyłączenie obsługi wyrażeń tego języka. Dlaczego?

Wróć myślami do czasów, kiedy do języka Java (w wersji 1.4) dodano słowo kluczowe *assert*. Nagle okazało się, że zupełnie poprawny, dotychczas niezastrzeżony identyfikator "assert" nabrał jakiegoś specjalnego *znaczenia* dla kompilatora Javy. Jeśli więc stosowałeś w swoim kodzie np. zmienną nazwaną *assert*, niespodziewanie popadałeś w konflikt z kompilatorem. Na szczęście J2SE w wersji 1.4 miało domyślnie wyłączone asercje. Oznacza to, że jeśli wiedziałeś, że w napisanym (lub ponownie kompilowanym) przez Ciebie kodzie nie występuje słowo *assert* w roli identyfikatora, mogłeś bez obaw włączyć obsługę asercji.

Podobna sytuacja ma miejsce w przypadku wyłączania obsługi wyrażeń języka EL — jeśli przypadkiem miałeś w swoim szablonie tekstowym (tradycyjnym kodzie HTML lub zwykłym tekście), którego użyłeś w kodzie JSP, wyrażenia wyglądające jak elementy języka EL (`{cokolwiek}`), stanąłbyś przed Wielkim Problemem, gdybyś nie mógł nakazać kontenerowi ignorowania wszystkiego, co wygląda jak wyrażenie języka EL (i traktowania odpowiednich fragmentów jak zwykłego, nieprzetworzonego tekstu). Warto przy tej okazji zwrócić uwagę na istotną różnicę pomiędzy wyrażeniami języka EL a asercjami:

### Obsługa języka EL jest domyślnie włączona!

Jeśli chcesz, aby fragmenty Twojego kodu JSP, które przypominają wyrażenia języka EL, były ignorowane, musisz to powiedzieć wprost (albo za pomocą dyrektywy `page`, albo odpowiedniego elementu deskryptora wdrożenia).

## Umieszczanie znacznika `<el-ignored>` w deskrytorze wdrożenia

```
<web-app ...>
...
<jsp-config>
 <jsp-property-group>
 <url-pattern>*.jsp</url-pattern>
 <el-ignored>
 true
 </el-ignored>
 </jsp-property-group>
</jsp-config>
...
</web-app>
```

## Stosowanie atrybutu `isELIgnored` dyrektywy `page`

```
<%@ page isELIgnored="true" %>
```

W przeciwieństwie do odpowiedniego znacznika deskryptora wdrożenia użyty tutaj atrybut dyrektywy `page` rozpoczyna się od słowa „is”!



Oglądaj to!

**Dyrektywa `page` ma wyższy priorytet od odpowiednich ustawień zdefiniowanych w deskrytorze wdrożenia!**

W razie konfliktu pomiędzy ustawieniem znacznika `<el-ignored>` w deskrytorze wdrożenia a atrybutem `isELIgnored` dyrektywy `page`, dyrektywa zawsze będzie miała pierwszeństwo! Dzięki temu możesz swobodnie określać domyślne zachowania w deskrytorze wdrożenia i w razie konieczności przykrywać je dla konkretnych stron JSP za pomocą dyrektyw `page`.



Oglądaj to!

**Uważaj na niespójności w nazewnictwie atrybutów i znaczników!**

W deskrytorze wdrożenia stosowany jest znacznik `<el-ignored>`, zatem ktoś mógłby pomyśleć, że odpowiednim atrybutem dyrektywy `page` będzie... może `elignored`? Okazuje się jednak, że jest inaczej — szukanie tego typu z pozoru naturalnych związków prowadzi do błędnych wniosków. Znacznik używany w deskrytorze wdrożenia i atrybut dyrektywy `page` dla ignorowania wyrażeń języka EL mają zupełnie inne nazwy. Nie daj się więc zwieść pozorom i nie myśl, że istnieje element `<is-el-ignored>`.

## Ale zaczekaj! Istnieje jeszcze jeden element JSP, którego jeszcze nie analizowaliśmy — akcje

Do tej pory zapoznałeś się z pięcioma różnymi typami elementów, które mogą występować w kodzie JSP: skryptletami, dyrektywami, deklaracjami, wyrażeniami Javy oraz wyrażeniami języka EL.

Jednak nie miałeś jeszcze do czynienia z **akcjami**. Istnieją dwa główne typy akcji: *standardowe* i... *nie*.

### Akcja standardowa:

```
<jsp:include page="dziwnaStopka.jsp" />
```

### Inna akcja:

```
<c:set var="ranking" value="32" />
```

Na razie nie musisz się martwić o to, co poszczególne wiersze JSP oznaczają i jak są interpretowane, wystarczy, że będziesz potrafił prawidłowo rozpoznać akcję, kiedy zobaczysz podobną składnię w analizowanym pliku JSP. Szczegóły omówimy później.

Przedstawiony podział może być jednak mylący, ponieważ istnieją akcje, których nie uważa się za *akcje standardowe*, mimo że są częścią bibliotek standardowych. Innymi słowy, z dalszej części tej książki dowiesz się, że niektóre spośród akcji niestandardowych (w celach egzaminacyjnych nazwano je *akcjami użytkownika*) w istocie są... standardowe, choć oficjalnie nadal nie są uważane za „akcje standardowe”. Tak, to prawda — istnieją ustandaryzowane niestandardowe akcje użytkownika. Czy teraz ten podział jest dla Ciebie jasny?

W jednym z kolejnych rozdziałów, kiedy dojdziemy do zagadnienia „stosowania znaczników”, poznamy nieco bogatsze słownictwo, które ułatwi nam bardziej szczegółową dyskusję na ten temat, nie powinienes się więc przejmować chwilowymi niejasnościami w tej kwestii. **Na razie interesuje nas wyłącznie Twoja zdolność rozpoznawania akcji w przeglądany kodzie JSP!**



Zaostrz ołówek

Przyjrzyj się składni definiowania akcji w kodzie JSP i porównaj ją ze składnią pozostałych rodzajów elementów JSP. Spróbuj następnie odpowiedzieć na następujące dwa pytania:

1) Jakie widzisz różnice pomiędzy elementem akcji a skryptletem?

2) Na jakiej podstawie rozpoznasz właśnie element akcji w analizowanym kodzie JSP?

Ćwiczenie z wyznaczania wartości



Ćwiczenie

Tabela wyników

Zastanów się nad praktycznymi następstwami wystąpień poszczególnych ustawień (lub kombinacji ustawień). Rozwiązania obu niniejszych ćwiczeń znajdziesz po odwróceniu strony, więc nie zaglądaj tam PRZED wypełnieniem tabel.

1 Wyliczanie wyrażeń EL

Umieść znak zaznaczenia ALBO w kolumnie przetwarzone, jeśli ustawienia zawarte w dwóch pierwszych kolumnach tabeli spowodują, że wartość wyrażania języka EL będzie przetwarzana, ALBO w kolumnie ignorowane, jeśli wyrażenie języka EL będzie traktowane jak zwykły fragment nieprzetworzonego tekstu. W żadnym wierszu tabeli nie możesz oczywiście umieścić dwóch znaków zaznaczenia.

Konfiguracja deskryptora wdrożenia <el-ignored>	Dyrektywa page isELIgnored	przetwarzane	ignorowane
nieokreślona	nieokreślona		
false	nieokreślona		
true	nieokreślona		
false	false		
false	true		
true	false		

2 Obsługa skryptów

Umieść znak zaznaczenia ALBO w kolumnie przetwarzone, jeśli ustawienie z pierwszej kolumny tabeli spowoduje, że wartość wyrażenia skryptowego zostanie normalnie wyznaczona, ALBO w kolumnie błąd, jeśli wystąpienie skryptu w kodzie JSP spowoduje błąd w procesie jego tłumaczenia na kod serwletu Javy.

Konfiguracja deskryptora wdrożenia <scripting-invalid>	przetwarzane	błąd
nieokreślona		
true		
false		



## JSP Element Magnets — Magnesiki z elementami JSP

Dopasuj elementy JSP do odpowiednich etykiet przez umieszczenie wybranych fragmentów w ramkach oznaczonych etykietami reprezentującymi właściwe typy elementów. Pamiętaj, że na rzeczywistym egzaminie będziesz musiał wykonywać podobne zadania typu „przeciągnij i upuść”, zatem nie powinieneś tego ćwiczenia lekceważyć!

### Typ elementu JSP

**dyrektywa**

**deklaracja**

**wyrażenie EL**

**skryptlet**

**wyrażenie**

**akcja**

### Fragment kodu JSP

*Przeciągnij i upuść te fragmenty w ramach oznaczonych odpowiednimi etykietami.*

```
<% Float jeden = new Float(42.5); %>
```

```
<%! int y = 3; %>
```

```
<%@ page import="java.util.*" %>
```

```
<jsp:include file="foo.html" />
```

```
<%= pageContext.getAttribute("foo") %>
```

```
email: ${applicationScope.email}
```



### Magnesiki z elementami JSP, ciąg dalszy

Wiesz już, jak nazywają się poszczególne elementy JSP, ale czy pamiętasz, **jak te elementy są rozmieszczane w generowanym kodzie serwletu**? Oczywiście musisz to pamiętać. Potraktuj niniejsze ćwiczenie jak drobne wsparcie (dodatkowe zajęcia praktyczne) przed nauką zupełnie innych zagadnień w kolejnych rozdziałach.

(Przenieś elementy do właściwych pól reprezentujących miejsca w wygenerowanym kodzie klasy serwletu, w których te elementy zostaną umieszczone w procesie tłumaczenia kodu JSP. Zwróć uwagę na fakt, że użyte w tym ćwiczeniu magnesy nie reprezentują RZECZYWISTEGO kodu, który zostanie wygenerowany, a jedynie oryginalne fragmenty kodu JSP).

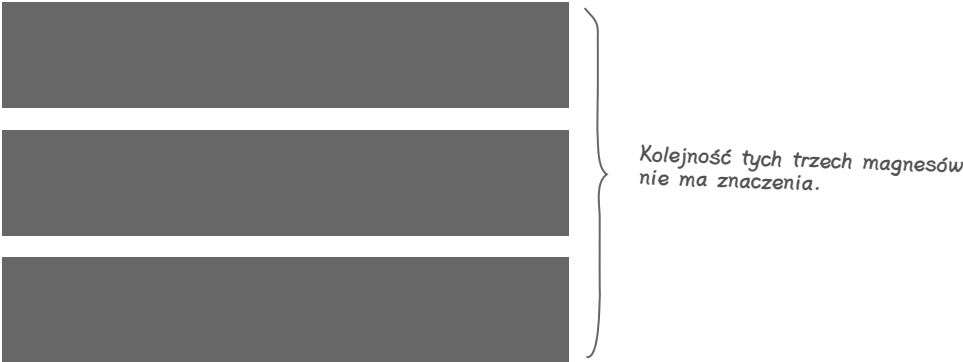
```
public class prostyLicznik_jsp extends org.apache.jasper.runtime.HttpJspBase
 implements org.apache.jasper.runtime.JspSourceDependent {

 ...

 public void _jspService(HttpServletRequest request, HttpServletResponse response)
 throws java.io.IOException, ServletException {

 ...

 ...
 }
}
```



```
<%= request.getAttribute("foo") %>
```

```
email: ${applicationScope.email}
```

```
<%@ page import="java.util.*" %>
```

```
<% Float jeden = new Float(42.5); %>
```

```
<%! int y = 3; %>
```



Tabela wyników  
ROZWIĄZANIA

1 Wyliczanie wyrażeń EL

Konfiguracja deskryptora wdrożenia <el-ignored>	Dyrektywa page isELIgnored	przetwarzane	ignorowane
nieokreślona	nieokreślona	✓	
false	nieokreślona	✓	
true	nieokreślona		✓
false	false	✓	
false	true		✓
true	false	✓	

2 Obsługa skryptów

Konfiguracja deskryptora wdrożenia <scripting-invalid>	przetwarzane	błąd
nieokreślona	✓	
true		✓
false	✓	



# JSP Element Magnets — Magnesiki z elementami JSP

## ROZWIĄZANIA

```
<%@ page import="java.util.*" %>
```

**dyrektywa**

```
<%! int y = 3; %>
```

**deklaracja**

```
email: ${applicationScope.email}
```

**wyrażenie EL**

```
<% Float jeden = new Float(42.5); %>
```

**skryptlet**

```
<%= pageContext.getAttribute("foo") %>
```

**wyrażenie**

```
<jsp:include file="foo.html" />
```

**akcja**



Oglądaj to!

**Samo słowo „wyrażenie” oznacza „wyrażenie skryptowe”, NIE „wyrażenie EL”.**

Słowo „wyrażenie” w odniesieniu do elementów JSP ma oczywiście więcej niż jedno znaczenie. Jeśli natkniesz się na określenie „wyrażenie” lub „wyrażenie skryptowe”, możesz przyjąć, że odnosi się ono do tego samego — wyrażenia zbudowanego w oparciu o składnię języka Java:

```
<%= foo.getName() %>
```

Słowo „wyrażenie”, które odwołuje się do języka EL, zawsze jest dodatkowo oznaczone skrótem „EL” (albo w samym opisie, albo w użytej etykiecie)! Powinieneś więc zawsze zakładać, że domyślnym znaczeniem słowa „wyrażenie” jest „wyrażenie skryptowe (wyrażenie Javy)”, nie kod języka EL.



## JSP Element Magnets: the Sequel — Magnesiki z elementami JSP, ciąg dalszy

ROZWIĄZANIA

```
<%@ page import="java.util.*" %>
```

Dyrektywa `page` z atrybutem `import` jest przekształcana w instrukcję importującą języka Java.

```
public class prostyLicznik_jsp extends org.apache.jasper.runtime.HttpJspBase
 implements org.apache.jasper.runtime.JspSourceDependent {
```

```
<%! int y = 3; %>
```

Deklaracje JSP są przekształcane w deklaracje SKŁADOWYCH generowanej klasy serwletu, zatem są umieszczane wewnątrz tej klasy, ale poza jakąkolwiek metodą.

```
public void _jspService(HttpServletRequest request, HttpServletResponse response)
 throws java.io.IOException, ServletException {
```

```
...
```

```
<%= request.getAttribute("foo") %>
```

Wyrażenia są przekształcane w wywołania metody `write()` wewnątrz metody obsługującej żądania.

```
<% Float jeden = new Float(42.5); %>
```

Skryptlety są umieszczane wewnątrz metody obsługującej żądania.

```
email: ${applicationScope.email}
```

Wyrażenia języka EL są umieszczane wewnątrz metody obsługującej żądania.

```
...
```

```
}
```

```
}
```

(Uwaga: kolejność tych trzech elementów nie ma znaczenia).

UWAGA: musisz pamiętać, że w praktyce kod JSP nie jest KOPIOWANY do serwletu w ten sposób... Przeciwnie, w całości podlega procesowi tłumaczenia na kod języka Java. Celem tego ćwiczenia jest jedynie pokazanie Ci, do których części wygenerowanej klasy serwletu TRAFIAJĄ te elementy, ale nie demonstrujemy kodu Javy faktycznie wygenerowanego na podstawie tych elementów. Przykładowo, na podstawie deklaracji JSP w postaci <%! int y = 3; %> zostalaby wygenerowana następująca deklaracja zmiennej składowej Javy: int y = 3;



---

1 Mamy dany następujący element deskryptora wdrożenia:

- 47. `<jsp-property-group>`
- 48. `<url-pattern>*.jsp</url-pattern>`
- 49. `<el-ignored>true</el-ignored>`
- 50. `</jsp-property-group>`

Jaki jest cel stosowania tego typu elementów? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. Wszystkie pliki z określonym rozszerzeniem (w tym przypadku *.jsp*) powinny być traktowane przez kontener JSP jak prawidłowo sformatowane pliki XML.
- ☐ B. Wszystkie pliki z określonym rozszerzeniem (w tym przypadku *.jsp*) powinny zawierać jakikolwiek kod języka EL (ang. *Expression Language*), który będzie odpowiednio interpretowany przez kontener JSP.
- ☐ C. Żaden z plików z określonym rozszerzeniem (w tym przypadku *.jsp*) domyślnie NIE powinien zawierać żadnego kodu języka EL, który byłby interpretowany i przetwarzany przez kontener JSP jako wyrażenie.
- ☐ D. Żaden, ten znacznik jest NIEZROZUMIAŁY dla kontenera.
- ☐ E. Chociaż przedstawiony znacznik jest prawidłowy, należy go uznać za nadmiarowy, ponieważ kontener domyślnie zachowuje się w taki właśnie sposób.

---

2 Które z wymienionych poniżej dyrektyw określają, że generowana odpowiedź protokołu HTTP będzie typu "image/svg"? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. `<%@ page type="image/svg" %>`
- ☐ B. `<%@ page mimeType="image/svg" %>`
- ☐ C. `<%@ page language="image/svg" %>`
- ☐ D. `<%@ page contentType="image/svg" %>`
- ☐ E. `<%@ page pageEncoding="image/svg" %>`

3 Mamy dany następujący kod JSP:

```
1. <%@ page import="java.util.*" %>
2. <html><body> Ludzie, którzy lubią
3. <%= request.getParameter("hobby") %>
4. to:

5. <% ArrayList al = (ArrayList) request.getAttribute("imiona"); %>
6. <% Iterator it = al.iterator();
7. while (it.hasNext()) { %>
8. <%= it.next() %>
9.

10. <% } %>
11. </body></html>
```

Które typy elementów zostały użyte w przedstawionym kodzie JSP?  
(Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. Wyrażenie EL.
- ☐ B. Dyrektywa.
- ☐ C. Wyrażenie.
- ☐ D. Tekst szablonu.
- ☐ E. Skryptlet.

4 Które zdania na temat metody **jspInit()** są prawdziwe?  
(Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. Metoda ta ma dostęp do obiektu **ServletConfig**.
- ☐ B. Metoda ta ma dostęp do obiektu **ServletContext**.
- ☐ C. Metoda jest wywoływana tylko raz.
- ☐ D. Metoda może być nadpisywana.

5 Które z wymienionych poniżej typów obiektów są dostępne dla metody **jspInit()**? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. **ServletConfig**
  - ☐ B. **ServletContext**
  - ☐ C. **JspServletConfig**
  - ☐ D. **JspServletContext**
  - ☐ E. **HttpServletRequest**
  - ☐ F. **HttpServletResponse**
- 

6 Mamy daną następującą dyrektywę JSP:

```
<%@ page isELIgnored="true" %>
```

Jaki jest efekt jej umieszczenia w kodzie strony? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. Żaden, taka dyrektywa **page** w ogóle NIE jest zdefiniowana.
  - ☐ B. Przedstawiona dyrektywa wyłącza w kontenerze JSP obsługę (wyznaczanie wartości) kodu języka EL dla wszystkich stron JSP należących do danej aplikacji.
  - ☐ C. Strona JSP, która zawiera tę dyrektywę, powinna być traktowana przez kontener JSP jak prawidłowo sformatowany plik XML.
  - ☐ D. Strona JSP z tą dyrektywą NIE powinna zawierać żadnego kodu języka EL, ponieważ odpowiednie wyrażenia nie zostaną właściwie zinterpretowane przez kontener JSP.
  - ☐ E. Przedstawiona dyrektywa **page** wyłączy obsługę wyrażeń języka EL pod warunkiem, że w deskrytorze wdrożenia zadeklarowano element **<el-ignored>true</el-ignored>** w ramach wzorca URL obejmującego daną stronę JSP.
- 

7 Które stwierdzenie odnośnie stron JSP jest prawdziwe? (Zaznacz tylko jedną odpowiedź).

- ☐ A. Tylko metoda **jspInit()** może być nadpisana.
- ☐ B. Tylko metoda **jspDestroy()** może być nadpisana.
- ☐ C. Tylko metoda **\_jspService()** może być nadpisana.
- ☐ D. Możemy nadpisać zarówno metodę **jspInit()**, jak i metodę **jspDestroy()**.
- ☐ E. Możemy nadpisać metody **jspInit()**, **jspDestroy()** i **\_jspService()**.

8 Który krok cyklu życia umieściliśmy w złym miejscu?

- ☐ A. Tłumaczenie kodu JSP na klasę serwletu.
- ☐ B. Kompilacja kodu źródłowego serwletu.
- ☐ C. Wywołanie metody `_jspService()`.
- ☐ D. Utworzenie egzemplarza klasy serwletu.
- ☐ E. Wywołanie metody `jspInit()`.
- ☐ F. Wywołanie metody `jspDestroy()`.

9 Które z wymienionych poniżej obiektów są poprawnymi, domyślnymi zmiennymi JSP?  
(Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. `stream`
- ☐ B. `context`
- ☐ C. `exception`
- ☐ D. `listener`
- ☐ E. `application`

10 Mamy dane żądanie z dwoma parametrami: pierwszym nazwanym "pierwszy", który reprezentuje imię użytkownika, oraz drugim nazwanym "ostatni" i reprezentującym jego nazwisko.

Który z zaprezentowanych poniżej skryptletów JSP prawidłowo wyświetli wartości tych parametrów?

- ☐ A. `<% out.println(request.getParameter("pierwszy"));  
out.println(request.getParameter("ostatni")); %>`
- ☐ B. `<% out.println(application.getInitParameter("pierwszy"));  
out.println(application.getInitParameter("ostatni")); %>`
- ☐ C. `<% println(request.getParameter("pierwszy"));  
println(request.getParameter("ostatni")); %>`
- ☐ D. `<% println(application.getInitParameter("pierwszy"));  
println(application.getInitParameter("ostatni")); %>`

**11** Mamy dany następujący kod JSP:

11. Witaj, `${uzytkownik.imie}!`

12. Twój numer: `<c:out value="${uzytkownik.telefon}"/>`

13. Twój adres: `<jsp:getProperty name="uzytkownik" property="adres" />`

14. `<% if (uzytkownik.isValid()) { %>Wszystko się zgadza!<% } %>`

Które zdania są prawdziwe? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. Wiersze 11. i 12. (i żaden inny) zawierają przykłady elementów języka EL.
  - ☐ B. Wiersz 14. jest przykładem kodu skryptletu.
  - ☐ C. Żaden z wierszy w zaprezentowanym przykładzie nie zawiera tekstu szablonu.
  - ☐ D. Wiersze 12. i 13. zawierają przykłady standardowych akcji JSP.
  - ☐ E. Wiersz 11. jest przykładem niewłaściwego użycia języka EL.
  - ☐ F. Wszystkie cztery wiersze tego przykładu stanowią poprawną zawartość kodu strony JSP.
- 

**12** Które z przedstawionych poniżej znaczników wyrażeń JSP prawidłowo wyświetli wartość parametru inicjalizacji kontekstu nazwanego "javax.sql.DataSource"?

- ☐ A. `<%= application.getAttribute("javax.sql.DataSource") %>`
  - ☐ B. `<%= application.getInitParameter("javax.sql.DataSource") %>`
  - ☐ C. `<%= request.getParameter("javax.sql.DataSource") %>`
  - ☐ D. `<%= contextParam.get("javax.sql.DataSource") %>`
- 

**13** Które zdania na temat wyłączania obsługi elementów skryptowych są prawdziwe? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. Nie możemy wyłączać obsługi elementów skryptowych z poziomu deskryptora wdrożenia.
- ☐ B. Wyłączenie obsługi elementów skryptowych jest możliwe jedynie na poziomie aplikacji.
- ☐ C. Możemy wyłączyć obsługę elementów skryptowych programowo za pomocą atrybutu `isScriptingEnabled` dyrektywy `page`.
- ☐ D. Możemy wyłączyć obsługę elementów skryptowych za pomocą elementu `<scripting-invalid>` deskryptora wdrożenia.

**14** Jakie są kolejne typy Javy następujących obiektów domyślnych JSP: **application**, **out**, **request**, **response**, **session**?

- ☐ A. `java.lang.Throwable`  
`java.lang.Object`  
`java.util.Map`  
`java.util.Set`  
`java.util.List`
- ☐ B. `javax.servlet.ServletConfig`  
`java.lang.Throwable`  
`java.lang.Object`  
`javax.servlet.jsp.PageContext`  
`java.util.Map`
- ☐ C. `javax.servlet.ServletContext`  
`javax.servlet.jsp.JspWriter`  
`javax.servlet.ServletException`  
`javax.servlet.ServletResponse`  
`javax.servlet.http.HttpSession`
- ☐ D. `javax.servlet.ServletContext`  
`java.io.PrintWriter`  
`javax.servlet.ServletConfig`  
`java.lang.Exception`  
`javax.servlet.RequestDispatcher`

**15** Który z poniższych wierszy jest przykładem składni stosowanej do importowania klasy w kodzie strony JSP?

- ☐ A. `<% page import="java.util.Date" %>`
- ☐ B. `<%@ page import="java.util.Date" @%>`
- ☐ C. `<%@ page import="java.util.Date" %>`
- ☐ D. `<% import java.util.Date; %>`
- ☐ E. `<%@ import file="java.util.Date" %>`

**16** Mamy dany następujący kod JSP:

1. `<%@ page isELIgnored="true" %>`
2. `<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>`
3. `<c:set var="awesomeBand" value="LIMOZEEN" />`
4. `${awesomeBand}`

Jaki wynik otrzymamy?

- ☐ A. `${awesomeBand}`
- ☐ B. `LIMOZEEN`
- ☐ C. Nie otrzymamy żadnego wyniku.
- ☐ D. Zostanie wygenerowany wyjątek, ponieważ wszystkie dyrektywy **taglib** muszą poprzedzać choć jedną dyrektywę **page**.



## Odpowiedzi do rozdziału 7.

1 Mamy dany następujący element deskryptora wdrożenia:

(Specyfikacja JSP 2.0, str. 1 – 87).

47. `<jsp-property-group>`  
 48. `<url-pattern>*.jsp</url-pattern>`  
 49. `<el-ignored>true</el-ignored>`  
 50. `</jsp-property-group>`

Jaki jest cel stosowania tego typu elementów? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. Wszystkie pliki z określonym rozszerzeniem (w tym przypadku *.jsp*) powinny być traktowane przez kontener JSP jak prawidłowo sformatowane pliki XML.
- ☐ B. Wszystkie pliki z określonym rozszerzeniem (w tym przypadku *.jsp*) powinny zawierać jakikolwiek kod języka EL (ang. *Expression Language*), który będzie odpowiednio interpretowany przez kontener JSP.
- ☒ C. Żaden z plików z określonym rozszerzeniem (w tym przypadku *.jsp*) domyślnie NIE powinien zawierać żadnego kodu języka EL, który byłby interpretowany i przetwarzany przez kontener JSP jako wyrażenie.
- ☐ D. Żaden, ten znacznik jest NIEZROZUMIAŁY dla kontenera.
- ☐ E. Chociaż przedstawiony znacznik jest prawidłowy, należy go uznać za nadmiarowy, ponieważ kontener domyślnie zachowuje się w taki właśnie sposób.

— Odpowiedź C wyłącza obsługę i wyznaczanie wartości wyrażeń EL przez kontener JSP (zgodny ze specyfikacją 2.0); domyślnie obsługa tych wyrażeń przez kontener jest włączona.

2 Które z wymienionych poniżej dyrektyw określają, że generowana odpowiedź protokołu HTTP będzie typu "image/svg"? (Zaznacz wszystkie prawidłowe odpowiedzi).

(Specyfikacja JSP 2.0, punkt 1.10.1).

- ☐ A. `<%@ page type="image/svg" %>`
- ☐ B. `<%@ page mimeType="image/svg" %>`
- ☐ C. `<%@ page language="image/svg" %>`
- ☒ D. `<%@ page contentType="image/svg" %>`
- ☐ E. `<%@ page pageEncoding="image/svg" %>`

— Odpowiedź D reprezentuje prawidłową składnię dla tej dyrektywy.

3 Mamy dany następujący kod JSP:

(Specyfikacja JSP 2.0, rozdział 1.).

```
1. <%@ page import="java.util.*" %>
2. <html><body> Ludzie, którzy lubią
3. <%= request.getParameter("hobby") %>
4. to:

5. <% ArrayList al = (ArrayList) request.getAttribute("imiona"); %>
6. <% Iterator it = al.iterator();
7. while (it.hasNext()) { %>
8. <%= it.next() %>
9.

10. <% } %>
11. </body></html>
```

Które typy elementów zostały użyte w przedstawionym kodzie JSP?  
(Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. Wyrażenie EL.
- ☒ B. Dyrektywa.
- ☒ C. Wyrażenie.
- ☒ D. Tekst szablonu.
- ☒ E. Skryptlet.

— Przedstawiony kod JSP nie zawiera żadnych wyrażeń języka EL. Istnieje tam dyrektywa (w wierszu 1.), wyrażenia (w wierszach 3. i 8.), tekst szablonu (między innymi w wierszu 2.) i oczywiście elementy skryptowe.

4 Które zdania na temat metody **jspInit()** są prawdziwe?  
(Zaznacz wszystkie prawidłowe odpowiedzi).

(Specyfikacja JSP 2.0, punkt 11.2.1).

- ☒ A. Metoda ta ma dostęp do obiektu **ServletConfig**.
- ☒ B. Metoda ta ma dostęp do obiektu **ServletContext**.
- ☒ C. Metoda jest wywoływana tylko raz.
- ☒ D. Metoda może być nadpisywana.

- 5 Które z wymienionych poniżej typów obiektów są dostępne dla metody **jspInit()**? (Zaznacz wszystkie prawidłowe odpowiedzi). *(Specyfikacja JSP 2.0, punkt 11.2.1).*

- ☒ A. **ServletConfig**
- ☒ B. **ServletContext**
- ☐ C. **JspServletConfig**
- ☐ D. **JspServletContext**
- ☐ E. **HttpServletRequest**
- ☐ F. **HttpServletResponse**

— Zawartość stron JSP jest tłumaczona na tradycyjne klasy serwletów, zatem mają one dostęp do tradycyjnych obiektów **ServletConfig** oraz **ServletContext**... jednocześnie metoda **jspInit()** jest wykonywana zbyt wcześnie w cyklu życia wygenerowanego serwletu, aby możliwy był dostęp do żądań i odpowiedzi.

- 6 Mamy daną następującą dyrektywę JSP: *(Specyfikacja JSP 2.0, str. 1 – 49).*

`<%@ page isELIgnored="true" %>`

Jaki jest efekt jej umieszczenia w kodzie strony? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. Żaden, taka dyrektywa **page** w ogóle NIE jest zdefiniowana.
- ☐ B. Przedstawiona dyrektywa wyłącza w kontenerze JSP obsługę (wyznaczanie wartości) kodu języka EL dla wszystkich stron JSP należących do danej aplikacji.
- ☐ C. Strona JSP, która zawiera tę dyrektywę, powinna być traktowana przez kontener JSP jak prawidłowo sformatowany plik XML.
- ☒ D. Strona JSP z tą dyrektywą NIE powinna zawierać żadnego kodu języka EL, ponieważ odpowiednie wyrażenia nie zostaną właściwie zinterpretowane przez kontener JSP.
- ☐ E. Przedstawiona dyrektywa **page** wyłączy obsługę wyrażeń języka EL pod warunkiem, że w deskrytorze wdrożenia zadeklarowano element `<el-ignored>true</el-ignored>` w ramach wzorca URL obejmującego daną stronę JSP.

— Odpowiedź B jest niepoprawna, ponieważ przedstawiona dyrektywa może mieć wpływ wyłącznie na bieżącą stronę JSP.

- 7 Które stwierdzenie odnośnie stron JSP jest prawdziwe? (Zaznacz tylko jedną odpowiedź). *(Specyfikacja JSP 2.0, rozdział 11.).*

- ☐ A. Tylko metoda **jspInit()** może być nadpisana.
- ☐ B. Tylko metoda **jspDestroy()** może być nadpisana.
- ☐ C. Tylko metoda **\_jspService()** może być nadpisana.
- ☒ D. Możemy nadpisać zarówno metodę **jspInit()**, jak i metodę **jspDestroy()**.
- ☐ E. Możemy nadpisać metody **jspInit()**, **jspDestroy()** i **\_jspService()**.

— Pamiętaj, że znak podkreślenia może stanowić wskazówkę dotyczącą braku możliwości nadpisywania danej metody.

8 Który krok cyklu życia umieściliśmy w złym miejscu?

(Specyfikacja JSP 2.0, rozdział 11.)

- ☐ A. Tłumaczenie kodu JSP na klasę serwletu.
- ☐ B. Kompilacja kodu źródłowego serwletu.
- ☒ C. Wywołanie metody `_jspService()`.
- ☐ D. Utworzenie egzemplarza klasy serwletu.
- ☐ E. Wywołanie metody `jspInit()`.
- ☐ F. Wywołanie metody `jspDestroy()`.

— Metoda `_jspService()` nigdy nie może być wywoływana przed metodą `jspInit()`.

9 Które z wymienionych poniżej obiektów są poprawnymi, domyślnymi zmiennymi JSP? (Zaznacz wszystkie prawidłowe odpowiedzi).

(Specyfikacja JSP 2.0, punkt 1.8.3).

- ☐ A. `stream`
- ☐ B. `context`
- ☒ C. `exception`
- ☐ D. `listener`
- ☒ E. `application`

— Odpowiedzi A, B i D reprezentują zmienne, które nie istnieją w formie obiektów domyślnych tworzonych przez kontener dla stron JSP.

10 Mamy dane żądanie z dwoma parametrami: pierwszym nazwanym "pierwszy", który reprezentuje imię użytkownika, oraz drugim nazwanym "ostatni" i reprezentującym jego nazwisko.

(Specyfikacja JSP 2.0, str. 1 – 41).

Który z zaprezentowanych poniżej skryptletów JSP prawidłowo wyświetli wartości tych parametrów?

- ☒ A. `<% out.println(request.getParameter("pierwszy"));  
out.println(request.getParameter("ostatni")); %>`
- ☐ B. `<% out.println(application.getInitParameter("pierwszy"));  
out.println(application.getInitParameter("ostatni")); %>`
- ☐ C. `<% println(request.getParameter("pierwszy"));  
println(request.getParameter("ostatni")); %>`
- ☐ D. `<% println(application.getInitParameter("pierwszy"));  
println(application.getInitParameter("ostatni")); %>`

— Odpowiedź A wykorzystuje domyślny obiekt "out" oraz jego metodę `println()`.

— W odpowiedziach C i D brakuje obiektu domyślnego "out".

11 Mamy dany następujący kod JSP:

(Specyfikacja JSP 2.0,  
str. 1 – 10).

11. Witaj, \${uzytkownik.imie}!

12. Twój numer: <c:out value="\${uzytkownik.telefon}"/>

13. Twój adres: <jsp:getProperty name="uzytkownik" property="adres" />

14. <% if (uzytkownik.isValid()) { %>Wszystko się zgadza!<% } %>

Które zdania są prawdziwe? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☒ A. Wiersze 11. i 12. (i żaden inny) zawierają przykłady elementów języka EL.
- ☒ B. Wiersz 14. jest przykładem kodu skryptetu. — Odpowiedź C jest niepoprawna, ponieważ wszystkie cztery wiersze zawierają tekst szablonu.
- ☐ C. Żaden z wierszy w zaprezentowanym przykładzie nie zawiera tekstu szablonu.
- ☐ D. Wiersze 12. i 13. zawierają przykłady standardowych akcji JSP. — Odpowiedź D jest niepoprawna, ponieważ 12. wiersz w przedstawionym fragmencie nie zawiera standardowej akcji JSP.
- ☐ E. Wiersz 11. jest przykładem niewłaściwego użycia języka EL. — Odpowiedź E jest niepoprawna, ponieważ zawarte w wierszu 11. wyrażenie języka EL jest prawidłowe.
- ☒ F. Wszystkie cztery wiersze tego przykładu stanowią poprawną zawartość kodu strony JSP.

12 Które z przedstawionych poniżej znaczników wyrażeń JSP prawidłowo wyświetli wartość parametru inicjalizacji kontekstu nazwanego "javax.sql.DataSource"?

(Specyfikacja JSP 2.0,  
str. 1 – 41).

- ☐ A. <%= application.getAttribute("javax.sql.DataSource") %>
- ☒ B. <%= application.getInitParameter("javax.sql.DataSource") %> — Odpowiedź B jest przykładem prawidłowego zastosowania obiektu domyślnego application.
- ☐ C. <%= request.getParameter("javax.sql.DataSource") %>
- ☐ D. <%= contextParam.get("javax.sql.DataSource") %>

13 Które zdania na temat wyłączania obsługi elementów skryptowych są prawdziwe? (Zaznacz wszystkie prawidłowe odpowiedzi).

(Specyfikacja JSP 2.0,  
punkt 3.3.3).

- ☐ A. Nie możemy wyłączać obsługi elementów skryptowych z poziomu deskryptora wdrożenia.
- ☐ B. Wyłączenie obsługi elementów skryptowych jest możliwe jedynie na poziomie aplikacji.
- ☐ C. Możemy wyłączyć obsługę elementów skryptowych programowo za pomocą atrybutu **isScriptingEnabled** dyrektywy page.
- ☒ D. Możemy wyłączyć obsługę elementów skryptowych za pomocą elementu **<scripting-invalid>** deskryptora wdrożenia. — Obsługę elementów skryptowych można wyłączyć jedynie z poziomu deskryptora wdrożenia. Element <jsp-property-group> umożliwia nam wyłączanie obsługi skryptów w wybranych stronach JSP przez zdefiniowanie wzorców adresów URL, które podlegają takiemu wyłączeniu.

**14** Jakie są kolejne typy Javy następujących obiektów domyślnych JSP: **application, out, request, response, session**? (Specyfikacja JSP 2.0, str. 1 – 41).

- ☐ A. `java.lang.Throwable`  
`java.lang.Object`  
`java.util.Map`  
`java.util.Set`  
`java.util.List`
- ☐ B. `javax.servlet.ServletConfig`  
`java.lang.Throwable`  
`java.lang.Object`  
`javax.servlet.jsp.PageContext`  
`java.util.Map`
- ☒ C. `javax.servlet.ServletContext`  
`javax.servlet.jsp.JspWriter`  
`javax.servlet.ServletException`  
`javax.servlet.ServletResponse`  
`javax.servlet.http.HttpSession`
- ☐ D. `javax.servlet.ServletContext`  
`java.io.PrintWriter`  
`javax.servlet.ServletConfig`  
`java.lang.Exception`  
`javax.servlet.RequestDispatcher`

— Odpowiedź C reprezentuje właściwe typy Javy dla wszystkich wymienionych obiektów domyślnych.

**15** Który z poniższych wierszy jest przykładem składni stosowanej do importowania klasy w kodzie strony JSP? (Specyfikacja JSP 2.0, str. 1 – 44).

- ☐ A. `<% page import="java.util.Date" %>`
- ☐ B. `<%@ page import="java.util.Date" @%>`
- ☒ C. `<%@ page import="java.util.Date" %>`
- ☐ D. `<% import java.util.Date; %>`
- ☐ E. `<%@ import file="java.util.Date" %>`

— Odpowiedzi A i D są niepoprawne, ponieważ pomiędzy znacznikami `<% ... %>` można umieszczać wyłącznie instrukcje języka programowania Java.

— Odpowiedź C jest jedynym przykładem prawidłowej składni JSP.

— Odpowiedź E jest niepoprawna, ponieważ specyfikacja JSP nie przewiduje możliwości stosowania dyrektywy `import`.

**16** Mamy dany następujący kod JSP:

1. `<%@ page isELIgnored="true" %>`
2. `<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>`
3. `<c:set var="awesomeBand" value="LIMOZEEN" />`
4. `${awesomeBand}`

(Specyfikacja JSP 2.0, punkt 1.10.1).

Jaki wynik otrzymamy?

- ☒ A. `${awesomeBand}`
- ☐ B. `LIMOZEEN`
- ☐ C. Nie otrzymamy żadnego wyniku.
- ☐ D. Zostanie wygenerowany wyjątek, ponieważ wszystkie dyrektywy **taglib** muszą poprzedzać choć jedną dyrektywę **page**.

— Odpowiedź A jest prawidłowa, ponieważ użyte wyrażenie EL zostanie zignorowane i przekazane na wyjście w niezmienionej formie.



## 8. Bezskryptowe strony JSP

# Strony bezskryptowe

Wszystko w moim życiu uległo zmianie na lepsze od momentu, w którym przestałem stosować skryptlety. Jestem wyższy, zwiększyłem masę mięśniową o dwa kilogramy i znacznie poprawiłem technikę szydełkowania.

To wspaniale. Ale musisz wiedzieć, że wszystkie technologie mają także swoje wady... kiedyś nie byłeś taki tysi.



**Porzuć skrypty.** Czy współpracujący z Tobą projektanci stron internetowych naprawdę muszą znać Javę? Czy to byłoby wobec nich w porządku? Czy oni oczekują od programistów Javy, którzy opracowują rozwiązania pracujące po stronie serwera, aby byli jednocześnie np. grafikami? A jeśli przyjmiemy nawet, że *jesteś* jedynie członkiem większego zespołu, czy rzeczywiście chciałbyś umieszczać rozbudowane fragmenty kodu Javy w swoich stronach JSP? Czyż nie nasuwa Ci się określenie „koszmar konserwacji oprogramowania”? Pisanie stron bezskryptowych jest nie tylko *możliwe*, ale stało się znacznie *prostsze* i bardziej elastyczne w specyfikacji JSP 2.0 głównie dzięki nowemu językowi wyrażań (ang. *Expression Language* — *EL*). Po doświadczeniach z językami JavaScript i XPath większość projektantów stron WWW nie będzie miała najmniejszego problemu ze stosowaniem języka EL — tak samo powinno być w Twoim przypadku (przynajmniej od momentu, kiedy przyzwyczaisz się do nowego języka wyrażań). Istnieje jednak kilka pułapek... wyrażenia języka EL wyglądają jak instrukcje Javy, ale ich znaczenie jest nieco inne. Wyrażenia EL zachowują się czasem zupełnie inaczej niż instrukcje Javy zbudowane w oparciu o identyczną składnię, musisz więc zachować ostrożność!



### **Budowanie stron JSP w oparciu o język wyrażeń (EL) oraz akcje standardowe**

- 7.1.** Napisz w języku EL fragment kodu wykorzystującego zmienne wysokopoziomowe. Do tego typu zmiennych zaliczamy następujące obiekty domyślne: pageScope, requestScope, sessionScope i applicationScope; param i paramValues; header i headerValues; cookies oraz initParam.
- 7.2.** Napisz fragment kodu wykorzystujący następujące operatory języka EL: operator dostępu do właściwości (operator `.`) oraz operator dostępu do kolekcji (operator `[]`).
- 7.3.** Napisz fragment kodu wykorzystujący następujące operatory języka EL: operatory arytmetyczne, operatory relacyjne oraz operatory logiczne.
- 7.4.** Napisz fragment kodu wykorzystujący funkcję EL, zidentyfikuj lub stwórz strukturę pliku TLD wykorzystywaną do zadeklarowania funkcji EL oraz zidentyfikuj lub opracuj przykładowy kod definiujący funkcję EL.
- 8.1.** Mając dany cel projektowy, opracuj fragment kodu wykorzystujący następujące akcje standardowe: jsp:useBean (z atrybutami 'id', 'scope', 'type' oraz 'class'), jsp:getProperty oraz jsp:setProperty (ze wszystkimi możliwymi kombinacjami atrybutów).
- 8.2.** Mając dany cel projektowy, opracuj fragment kodu wykorzystujący następujące akcje standardowe: jsp:include, jsp:forward oraz jsp:param.
- 6.7.** Mając dany konkretny cel projektowy obejmujący dołączanie segmentu JSP w kodzie innej strony, napisz kod JSP wykorzystujący najwłaściwszy mechanizm takiego dołączania (dyrektywę include lub akcję standardową <jsp:include>).

### **Uwagi wyjaśniające:**

*Wszystkie wymienione obok cele zostaną dogłębnie omówione jeszcze w tym rozdziale. Niniejszy rozdział jest wyjątkowo obszerny, zatem przygotuj się na jego długotrwałe studiowanie — mamy do omówienia mnóstwo interesujących i istotnych szczegółów.*

*W rozdziale omówimy OBA mechanizmy dołączania: akcję standardową <jsp:include> z celu 8.2 oraz dyrektywę include z celu 6.7 (większość celów należących do 6. części egzaminu została omówiona w poprzednim rozdziale poświęconym technologii JSP).*

## Działanie naszej aplikacji MVC jest uzależnione od atrybutów

Zapewne pamiętasz, jak w naszej oryginalnej aplikacji MVC (generującej porady piwne) serwlet *kontrolera* komunikował się z *modelem* (klasą Javy z logiką biznesową), po czym ustawiał atrybut w zasięgu żądania przed jego przekazaniem dalej do *widoku* w postaci strony JSP.

Strona JSP musi *otrzymać* ten atrybut z zasięgu żądania i wykorzystać go do wizualizowania odpowiedzi odsyłanej do klienta (autora żądania). Poniżej przedstawiono krótkie, uproszczone fragmenty kodu obrazujące mechanizm przekazywania atrybutu z kontrolera do widoku (spróbuj sobie wyobrazić rozmowę pomiędzy serwletem a modelem):

### Kod serwletu (kontrolera)

```
public void doPost(HttpServletRequest request, HttpServletResponse response)
 throws IOException, ServletException {
```

```
String nazwa = request.getParameter("nazwaUzytkownika");
request.setAttribute("nazwa", nazwa);
```

Używamy odczytanego z formularza parametru żądania do ustawienia atrybutu zasięgu żądania, który zostanie przekazany wraz z obiektem żądania do strony JSP.

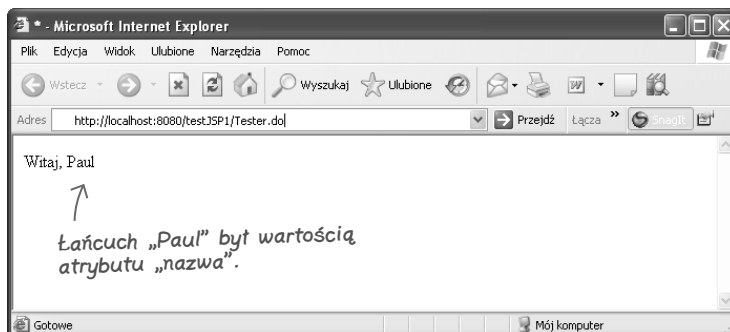
```
RequestDispatcher view = request.getRequestDispatcher("/wynik.jsp");
view.forward(request, response);
```

Przekazujemy żądanie do widoku.

### Kod strony JSP (widoku)

```
<html><body>
Witaj,
<%= request.getAttribute("nazwa") %>
</body></html>
```

Używamy wyrażenia skryptowego do odczytania wartości atrybutu i wyświetlenia jej w odpowiedzi. (Pamiętaj: wyrażenia skryptowe ZAWSZE są argumentami wywołania metody `out.print()`).



# A co będzie, jeśli przekazany w ten sposób atrybut nie będzie łańcuchem, tylko np. obiektem klasy Osoba?

I nie samym obiektem klasy Osoba, tylko obiektem z właściwością „imie”. Używamy pojęcia „właściwość” w nieco innym znaczeniu niż to przyjęte w społeczności programistów komponentów Enterprise JavaBeans\* — klasa Osoba zawiera parę metod `getImie()` i `setImie()`, które zgodnie ze specyfikacją technologii EJB wskazywałyby na fakt posiadania przez tę klasę własności nazwanej „imie”. Nie zapominaj, że uznanie pola „imie” za właściwość musi oznaczać zmianę wielkości pierwszej litery — „i”. Innymi słowy, nazwa właściwości jest tym, co otrzymujemy po odrzuceniu przedrostka „get” oraz „set” i zmianie wielkości pierwszej litery słowa bez tego przedrostka na małą. Oznacza to, że `getImie` (lub `setImie`) staje się **imie**.

## Kod serwletu

```
public void doPost(HttpServletRequest request, HttpServletResponse response)
 throws IOException, ServletException
{
 foo.Osoba p = new foo.Osoba();
 p.setImie("Evan");
 request.setAttribute("osoba", p);

 RequestDispatcher view = request.getRequestDispatcher("wynik.jsp");
 view.forward(request, response);
}
```

## Kod JSP

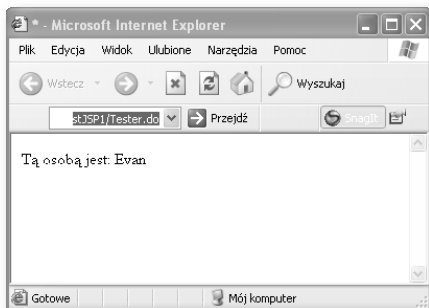
```
<html><body>
```

```
Tą osobą jest: <%= request.getAttribute("osoba") %>
```

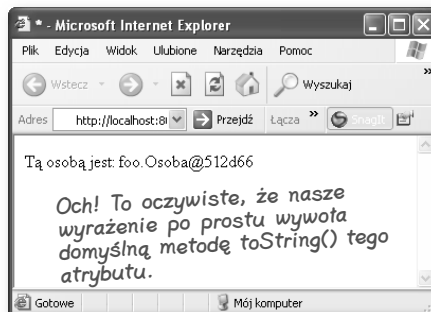
```
</body></html>
```

Co zwróci metoda `getAttribute()`?

## Co CHCEMY otrzymać:



## Co MAMY:



Prosty komponent  
JavaBean.

foo.Osoba
<pre>jpublic String getImie () public void setImie (String)</pre>

Na podstawie dostępnej pary metod zwracających i ustawiających możemy stwierdzić, że klasa Osoba zawiera właściwość nazwaną „imie” (zwróć uwagę na małą literę „i”).

\* Komponenty JavaBeans omówimy kilka stron dalej, na razie wystarczy, że będziesz wiedział, iż taki komponent jest zwykłą, tradycyjną klasą Javy z metodami zwracającymi i ustawiającymi zdefiniowanymi zgodnie z pewną konwencją nazewnictwa.

## Do odczytania właściwości imie obiektu Osoba potrzebujemy więcej kodu

Samo przesłanie wyniku metody `getAttribute()` do wywołania metody `write()` nie zapewni nam spodziewanego efektu — w ten sposób spowodujemy jedynie wywołanie metody `toString()` danego obiektu. A ponieważ klasa `Osoba` nie nadpisuje odziedziczonej metody `Object.toString()`, cóż, możemy łatwo przewidzieć, co się stanie. Nadal jednak chcemy wyświetlić właściwość *imie* obiektu `Osoba`.

### Kod JSP

```
<html><body>
```

```
<% foo.Osoba p = (foo.Osoba) request.getAttribute("osoba"); %>
```

```
Tą osobą jest: <%= p.getImie() %>
```

```
</body></html>
```

Wyświetla wynik wykonania metody `getImie()`.

### LUB wersja z pojedynczym wyrażeniem

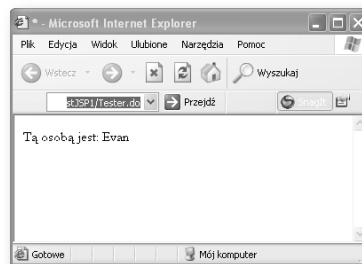
```
<html><body>
```

```
Tą osobą jest:
```

```
<%= ((foo.Osoba) request.getAttribute("osoba")).getImie() %>
```

```
</body></html>
```

### Co MAMY:



**Ale nagle przypominamy sobie pewną REGULĘ...**

**Tę, którą można streścić słowami  
„stosowanie elementów skryptowych to nasz koniec”.**

**Potrzebujemy zupełnie innego rozwiązania.**

# Osoba jest komponentem JavaBean, zatem użyjemy standardowych akcji dla tego typu komponentów

Za pomocą kilku akcji standardowych możemy całkowicie wyeliminować konieczność stosowania skryptów w kodzie stron JSP (pamiętaj: kod skryptowy obejmuje deklaracje, skryptlety oraz wyrażenia), nie tracąc przy tym możliwości wyświetlania wartości właściwości *imie* atrybutu *osoba*. Nie zapominaj, że *imie* nie jest atrybutem — jedynym atrybutem żądania jest w tym przypadku obiekt *osoba*. Właściwość *imie* jest po prostu tym, co zwraca metoda `getImie()` klasy *Osoba*.

## Bez standardowych akcji (z elementami skryptowymi)

```
<html><body>
```

*W ten sposób wyświetlaliśmy  
tę wartość do tej pory.*

```
<% foo.Osoba p = (foo.Osoba) request.getAttribute("osoba"); %>
```

```
Tą osobą jest: <%= p.getImie() %>
```

```
</body></html>
```

## Ze standardowymi akcjami (bez elementów skryptowych)

```
<html><body>
```

```
<jsp:useBean id="osoba" class="foo.Osoba" scope="request" />
```

```
Osobą utworzoną przez serwlet jest: <jsp:getProperty name="osoba" property="imie" />
```

```
</body></html>
```

*Żadnego kodu Javy! Nie ma żadnych  
elementów skryptowych, tylko dwie  
akcje standardowe.*

## Dekonstrukcja znaczników `<jsp:useBean>` i `<jsp:getProperty>`

Tym, czego naprawdę potrzebowaliśmy, była funkcjonalność standardowej akcji `<jsp:getProperty>`, ponieważ naszym celem było jedynie wyświetlenie wartości właściwości imię obiektu osoba. Skąd jednak kontener wie, co oznacza używane odwołanie "osoba"? Gdybyśmy w naszym kodzie użyli samego znacznika `<jsp:getProperty>`, nasze rozwiązanie przypominałoby trochę kod wykorzystujący niezadeklarowaną zmienną — nazwaną "osoba". Kontener przeważnie nie ma pojęcia, co mamy na myśli, chyba że WCZEŚNIEJ umieścimy w kodzie strony znacznik `<jsp:useBean>`. Użycie znacznika `<jsp:useBean>` jest jednym ze sposobów deklarowania i inicjalizowania faktycznego obiektu komponentu, którego użyjemy później w znaczniku standardowej akcji `<jsp:getProperty>`.

**Za pomocą znacznika `<jsp:useBean>` deklarujemy i inicjalizujemy atrybut komponentu**

```
<jsp:useBean id="osoba" class="foo.Osoba" scope="request" />
```

Identyfikuje standardową akcję.

Deklaruje identyfikator dla obiektu komponentu. Identyfikator odpowiada nazwie używanej w momencie, w którym kod naszego serwletu wywołuje metodę: `request.setAttribute("osoba", p);`

Deklaruje typ klasy (oczywiście kompletny) dla obiektu komponentu.

Identyfikuje zasięg atrybutu dla danego obiektu komponentu.

**Za pomocą znacznika `<jsp:getProperty>` odczytujemy wartość właściwości atrybutu komponentu**

```
<jsp:getProperty name="osoba" property="imie" />
```

Identyfikuje standardową akcję.

Identyfikuje rzeczywisty obiekt komponentu. Nazwa tego obiektu będzie odpowiadała wartości atrybutu "id" ze znacznika `<jsp:useBean>`.

Identyfikuje nazwę właściwości (innymi słowy, określa nazwę używaną w należącej do klasy komponentu metodzie zwracającej i ustawiającej daną właściwość).

## Także akcja <jsp:useBean> może TWORZYĆ komponenty!

Jeśli standardowa akcja <jsp:useBean> nie może znaleźć obiektu nazwanego „osoba”, może taki obiekt po prostu stworzyć! Działanie tej akcji przypomina trochę funkcjonowanie metody `request.getSession()` (lub `getSession(true)`) — najpierw podejmuje się próbę znalezienia istniejącego elementu, ale jeśli poszukiwania zakończą się niepowodzeniem, zostaje utworzony nowy element.

Przeanalizuj fragment kodu wygenerowanego serwletu, a przekonasz się, jakie jest faktyczne znaczenie standardowej akcji <jsp:useBean> — mamy tam instrukcję warunkową *if*! Odpowiedni fragment kodu weryfikuje istnienie komponentu w oparciu o wartości atrybutów *id* i *scope* znacznika, po czym — jeśli nie uda się znaleźć komponentu pasującego do tych wartości — tworzy nowy obiekt klasy określonej w atrybucie *class*, przypisuje nowoutworzony obiekt do zmiennej *id* i ustawia tę zmienną w roli atrybutu zdefiniowanego w naszym znaczniku zasięgu *scope*.

### Następujący znacznik

```
<jsp:useBean id="osoba" class="foo.Osoba" scope="request" />
```

### Jest tłumaczony na następujący kod w ramach metody `_jspService()`

```
foo.Osoba osoba = null;
synchronized (request) {
 osoba = (foo.Osoba)_jspx_page_context.getAttribute("osoba", PageContext.REQUEST_SCOPE);
 if (osoba == null) {
 osoba = new foo.Osoba();
 _jspx_page_context.setAttribute("osoba", osoba, PageContext.REQUEST_SCOPE);
 }
}
```

*Deklaruje zmienną w oparciu o wartość atrybutu id. Właśnie dzięki temu identyfikatorowi możliwe będzie odwoływanie się danej do zmiennej z poziomu pozostałych elementów naszej strony JSP (w tym także pozostałych znaczników komponentu).*

*Próbuje odczytać wartość atrybutu z zasięgu zdefiniowanego w znaczniku i przypisuje uzyskany wynik do zmiennej identyfikatora.*

*ALE jeśli w danym zasięgu NIE istnieje atrybut z taką nazwą...*

*Stwórz odpowiedni obiekt i przypisz go do zmiennej identyfikatora.*

*Na końcu musimy jeszcze ustawić nowy obiekt w formie atrybutu zdefiniowanego przez nas zasięgu.*

Byłoby bardzo niedobrze,  
gdyby tak się stało – NIE CHCĘ  
komponentu bez ustawionych  
właściwości! Jeśli kontener stworzy komponent  
w oparciu o ten znacznik, nowy komponent  
nie będzie zawierał odpowiednich wartości  
w swoich właściwościach...



## Możesz użyć znacznika `<jsp:setProperty>`

Wiesz już jednak, że wszędzie tam, gdzie istnieje mechanizm get, zwykle istnieje także odpowiedni mechanizm set. Znacznik `<jsp:setProperty>` jest trzecią i ostatnią standardową akcją komponentu. Jej stosowanie w kodzie JSP jest bardzo proste:

```
<jsp:useBean id="osoba" class="foo.Osoba" scope="request" />
<jsp:setProperty name="osoba" property="imie" value="Fred" />
```

Tym gorzej! Nasze  
NOWE rozwiązanie oznacza, że jeśli  
dany komponent już istnieje, moja strona  
JSP zmieni już istniejącą wartość jego  
właściwości! Chcę ustawiać tę właściwość  
tylko w NOWYCH komponentach...



# Znacznik <jsp:useBean> może mieć ciało!

Jeśli umieścimy nasz kod ustawiający (znacznik <jsp:setProperty>) w ciele znacznika <jsp:useBean>, **ustawienie właściwości będzie miało charakter warunkowy!** Innymi słowy, wartości tej właściwości będą ustawiane *tylko* wtedy, gdy będzie tworzony *nowy* komponent. Jeśli natomiast zostanie znaleziony już istniejący komponent z danym zasięgiem i danym identyfikatorem, ciało tego znacznika nigdy nie zostanie wykonane, zatem nasz kod JSP nie spowoduje ponownego ustawienia właściwości.

W ciele znacznika <jsp:useBean> możemy umieścić kod, który będzie wykonywany warunkowo... czyli TYLKO wtedy, gdy nie uda się znaleźć już istniejącego atrybutu komponentu i zostanie utworzony nowy komponent.

Nie ma ukośnika!

```
<jsp:useBean id="osoba" class="foo.Osoba" scope="page">
```

To jest wspominane ciało.

```
<jsp:setProperty name="osoba" property="imie" value="Fred" />
```

```
</jsp:useBean >
```

Na końcu musimy jeszcze zamknąć znacznik. Wszystko pomiędzy znacznikiem otwierającym a znacznikiem zamykającym stanowi wspomniane już ciało akcji <jsp:useBean>.

Każdy kod w ciele znacznika <jsp:useBean> jest WARUNKOWY. Taki kod jest wykonywany TYLKO wtedy, gdy dany komponent nie zostanie znaleziony i konieczne jest utworzenie nowego komponentu.

**P:** Dlaczego nie możemy po prostu zdefiniować argumentów konstruktora komponentu? Dlaczego musimy w każdym przypadku podejmować dodatkowy wysiłek związany z ustawianiem wartości?

**U:** Odpowiedź na to pytanie jest bardzo prosta: komponenty nie mogą MIEĆ konstruktorów otrzymujących argumenty! Cóż, każda klasa Javy może mieć taki konstruktor, ale kiedy już dany obiekt ma być traktowany jak komponent, Prawo Komponentów wyraźnie stwierdza, że będzie wywoływany TYLKO publiczny i bezargumentowy konstruktor tego komponentu. W praktyce, jeśli NIE zdefiniujemy publicznego, bezargumentowego konstruktora w klasie naszego komponentu, próba uruchomienia aplikacji zakończy się niepowodzeniem.

**P:** Czym u diabła jest Prawo Komponentów?

**U:** Jest to zbiór reguł wynikających z pradownej specyfikacji komponentów JavaBeans. Mówimy o komponentach JavaBeans — NIE Enterprise JavaBeans (EJB), które nie mają tutaj nic do rzeczy. (Możesz to sprawdzić sam). Zwykle, stara specyfikacja komponentów JavaBeans (nie w wersji Enterprise) dokładnie określa sytuację, w której klasa Javy staje się komponentem. Odpowiednie normy tej specyfikacji są co prawda zbyt skomplikowane, ale zanim zaczniesz stosować tego typu komponenty ze stronami JSP i serwetami, musisz znać tylko kilka niniejszych reguł (wspominamy tylko te reguły, które mają zastosowanie w naszych działaniach dotyczących serwetów i stron JSP):

- 1) MUSISZ zdefiniować publiczny, bezargumentowy konstruktor.
- 2) MUSISZ nazywać swoje publiczne metody zwracające i ustawiające zgodnie ze schematem: na początku słowa "get" (lub "is" w przypadku zwracania wartości logicznych) oraz "set", następnie nazwa właściwości rozpoczynająca się od wielkiej litery (np. getFoo(), setFoo()). Nazwę właściwości można odczytać przez odrzucenie przedrostka "get" lub "set" i zmianę pierwszej litery na małą.
- 3) Typ argumentu metody ustawiającej i typ wartości zwracanej przez metodę odczytującą MUSZĄ być identyczne. Poniższe deklaracje jednoznacznie definiują typ właściwości:  
`int getFoo() void setFoo(int foo)`
- 4) Nazwę i typ właściwości określa się na podstawie metod zwracających i ustawiających, a NIE w oparciu o konkretną składową klasy. Przykładowo, samo zdefiniowanie prywatnej zmiennej foo typu int wcale NIE musi oznaczać, że mamy do czynienia z właściwością. Możemy przecież zupełnie dowolnie nazywać swoje zmienne. Nazwa właściwości "foo" pochodzi z odpowiedniej pary metod. Innymi słowy, nasza klasa zawiera właściwość dlatego, że zdefiniowano w niej odpowiednią parę metod zwracających i ustawiających. To, jak te metody zaimplementujesz, zależy już tylko od Ciebie.
- 5) W przypadku wykorzystywania komponentów na stronach JSP typem właściwości POWINIEN być albo String, albo jeden z typów podstawowych. W przeciwnym przypadku nadal będziemy mieli do czynienia z prawidłowym komponentem, ale stracimy możliwość korzystania wyłącznie z akcji standardowych, a być może będziemy nawet zmuszeni do stosowania elementów skryptowych.

## Serwlet wygenerowany w sytuacji, w której znacznik `<jsp:useBean>` ma określone ciało

To proste. Kontener umieszcza dodatkowy kod ustawiający właściwość wewnątrz instrukcji warunkowej *if*.

### Kod wewnątrz metody `_jspService()` WRAZ Z ciałem znacznika `<jsp:useBean>`

```
foo.Osoba osoba = null;
osoba = (foo.Osoba) _jspx_page_context.getAttribute("osoba", PageContext.PAGE_SCOPE);

if (osoba == null) {
 osoba = new foo.Osoba();
 _jspx_page_context.setAttribute("osoba", osoba, PageContext.PAGE_SCOPE);
}
```

*Deklarujemy zmienną referencyjną.*

*Szukamy istniejącego atrybutu z nazwą i zasięgiem zgodnym z danymi użytymi w znaczniku.*

*Jeśli taki atrybut nie istnieje, tworzymy nowy egzemplarz.*

*Wiążemy nowy obiekt komponentu z określonym zasięgiem.*

*TO jest nowy fragment kodu. Takie wywołanie jest stosowane TYLKO wtedy, gdy znacznik useBean ma zdefiniowane ciało.*

```
org.apache.jasper.runtime.JspRuntimeLibrary.introspecthelper(
 _jspx_page_context.findAttribute("osoba"), "imie", "Fred", null, null, false);
```

```
}
osoba.setImie("Fred");
```

*Zapewne oczekiwałeś wywołania: `osoba.setImie("Fred");` ale właśnie takie jest znaczenie tego fragmentu kodu. Jedyną różnicą polega na wykorzystaniu uniwersalnej metody ustawiania właściwości, która otrzymuje w formie argumentów atrybut, właściwość oraz wartość. Efekt końcowy tego wywołania jest jednak taki sam: ostatecznie i tak zostanie wywołana metoda `setImie()` dla obiektu `Osoba`. (Pamiętaj, że nie musisz znać kodu implementacji kontenera Tomcat... powinien Cię interesować wyłącznie efekt końcowy).*

# Czy można tworzyć referencje polimorficzne do komponentów?

Kiedy w kodzie JSP zapisujemy znacznik `<jsp:useBean>`, atrybut `class` automatycznie determinuje klasę nowego obiektu (oczywiście, jeśli taki obiekt w ogóle jest tworzony). Atrybut ten określa także typ zmiennej *referencji*, która będzie wykorzystywana w wygenerowanym kodzie serwletu.

## W ten sposób OBECNIE zapisujemy kod JSP

```
<jsp:useBean id="osoba" class="foo.Osoba" scope="page" />
```

## Wygenerowany serwlet

```
foo.Osoba osoba = null;
// kod odczytujący atrybut osoba
if (osoba == null) {
 osoba = new foo.Osoba();
 ...
}
```

Użyty w tym znaczniku atrybut `class` reprezentuje zarówno typ referencji, JAK I typ obiektu.

Ale... co należałoby zrobić, gdybyśmy chcieli, aby typ referencji był *inny* niż rzeczywisty typ obiektu? Przekształćmy teraz klasę `Osoba` w klasę abstrakcyjną, po czym opracujemy konkretną podklasę `Pracownik`. Wyobraź sobie, że chcemy, aby typem *referencji* była klasa `Osoba`, natomiast nowym typem *obektu* była klasa `Pracownik`.

```
package foo;

public abstract class Osoba {
 private String imie;

 public void setImie(String imie) {
 this.imie=imie;
 }

 public String getImie() {
 return imie;
 }
}
```

```
package foo;

public class Pracownik extends Osoba {
 private int idPrac;

 public void setIdPrac(int idPrac) {
 this.idPrac = idPrac;
 }

 public int getIdPrac() {
 return idPrac;
 }
}
```

## Dodawanie atrybutu type do znacznika `<jsp:useBean>`

Po wprowadzeniu tej zmiany do klasy `Osoba` będziemy mieli poważny problem, jeśli nie uda się znaleźć odpowiedniego atrybutu:

### nasz oryginalny kod JSP

```
<jsp:useBean id="osoba" class="foo.Osoba" scope="page" />
```

### doprowadzi do następującego rezultatu:

```
java.lang.InstantiationException: foo.Osoba
```

### ponieważ kontener próbuje wykonać:

```
new foo.Osoba();
```

*Osoba jest teraz klasą abstrakcyjną! To oczywiste, że tworzenie obiektów tej klasy jest niemożliwe, ale — jak się okazuje — kontener nadal próbuje to zrobić w oparciu o atrybut class znacznika `<jsp:useBean>`.*

Musimy spowodować, że typem zmiennej *referencji* będzie klasa `Osoba`, natomiast *obiekt* będzie egzemplarzem klasy `Pracownik`. Takie rozwiązanie będzie możliwe, jeśli uzupełnimy znacznik `<jsp:useBean>` o atrybut `type`.

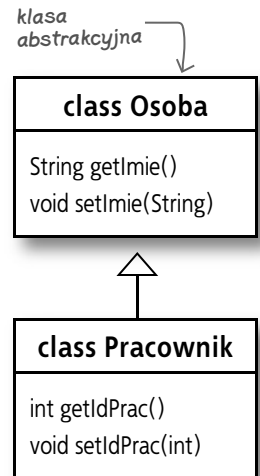
### Nasz nowy kod JSP z atrybutem `type`

```
<jsp:useBean id="osoba" type="foo.Osoba" class="foo.Pracownik" scope="page" />
```

### Wygenerowany serwlet

```
foo.Osoba osoba = null;
// kod odczytujący atrybut osoba
if (osoba == null) {
 osoba = new foo.Pracownik();
 ...
}
```

*Teraz typem referencji jest abstrakcyjna klasa `Osoba`, natomiast typem obiektu jest konkretna klasa `Pracownik`.*



Typ może być typem klasowym, typem abstrakcyjnym lub interfejsem — czymkolwiek, co może być wykorzystane w roli zadeklarowanego typu *referencji* do klasowego typu obiektu komponentu. Nie możesz rzecz jasna naruszyć reguł rządzących typami w języku Java. Jeśli typu *klasy* nie będzie można przypisać do typu referencji, będziemy mieli poważny problem. Nasza *klasa* musi więc być podklasą lub konkretną implementacją *typu*.

### Używanie atrybutu type bez atrybutu class

Co się stanie, jeśli w znaczniku `<jsp:useBean>` zadeklarujemy typ, ale nie zadeklarujemy klasy? Czy zadeklarowanie abstrakcyjnego lub konkretnego typu ma jakiegokolwiek znaczenie?

#### JSP

↙ bez klasy, tylko typ

```
<jsp:useBean id="osoba" type="foo.Osoba" scope="page" />
```

#### Wynik w sytuacji, gdy atrybut osoba już istnieje w zasięgu „page”

*Wszystko działa doskonale.*

#### Wynik w sytuacji, gdy atrybut osoba NIE istnieje w zasięgu „page”

NIE DZIAŁA!!

```
java.lang.InstantiationException: bean osoba not found within scope
```

**Jeśli atrybut `type` jest używany bez atrybutu `class`, dany komponent musi już istnieć.**

**Jeśli wykorzystujemy atrybut `class` (z lub bez atrybutu `type`), klasa NIE może być abstrakcyjna i musi zawierać publiczny, bezargumentowy konstruktor.**

**P:** W przedstawionym przykładzie „foo.Osoba” jest typem abstrakcyjnym, zatem z **OCZYWISTYCH** względów nie można go skonkretyzować. Co by się stało, gdybyśmy zmienili ten typ na „foo.Pracownik”? Czy klasa Pracownik będzie wówczas wykorzystywana zarówno w roli typu referencji, JAK I typu obiektu?

**U:** NIE! Takie rozwiązanie nigdy nie zadziała. Kontener odkryje, że dany komponent nie istnieje, i od razu zobaczy, że w znaczniku `<jsp:useBean>` użyto samego atrybutu `type`, bez atrybutu `class`. Będzie więc wiedział, że przekazaliśmy mu TYLKO połowę potrzebnych danych — sam typ referencji bez typu obiektu. Innymi słowy, nie wskazaliśmy kontenerowi klasy, której egzemplarz ma być utworzony!

Nie istnieje żadna rezerwowa reguła, która określałaby: „jeśli nie możesz znaleźć obiektu, kontynuuj pracę i użyj wskazanego typu ZARÓWNO dla referencji, jak i dla obiektu”. Nie, kontener działa na zupełnie INNYCH zasadach.

**Rozstrzygnięcie:** jeśli używasz atrybutu `type` bez atrybutu `class`, lepiej się UPEWNIJ, że dany komponent jest już składowany w postaci atrybutu (z dokładnie takim samym zasięgiem i identyfikatorem, jakiego użyłeś w znaczniku).

## Domyślną wartością atrybutu scope jest „page”

Jeśli nie określisz zasięgu ani w znaczniku `<jsp:useBean>`, ani w znaczniku `<jsp:getProperty>`, kontener użyje domyślnego zasięgu strony (wartości „page”).

### To

```
<jsp:useBean id="osoba" class="foo.Pracownik" scope="page" />
```

### Ma takie samo znaczenie jak to

```
<jsp:useBean id="osoba" class="foo.Pracownik" />
```



Oglądaj to!

### Nie myl atrybutu type z atrybutem class!

Przyjrzyj się poniższemu fragmentowi kodu:

```
<jsp:useBean id="osoba" type="foo.Pracownik" class="foo.Osoba" />
```

Musisz być przygotowany na to, że podobne rozwiązania NIGDY nie zadziałają! Otrzymasz za to wielki, tłusty wyjątek:

```
org.apache.jasper.JasperException: Unable to compile class for JSP
foo.Osoba is abstract; cannot be instantiated
 Osoba = new foo.Osoba();
```

KONIECZNIE musisz zapamiętać, że:

**type == typ referencji**  
**class == typ obiektu**

lub zapamiętać to w nieco innej formie:

**Atrybut type jest tym, co DEKLARUJESZ (może być abstrakcyjny)**  
**Atrybut class jest tym, co KONKRETYZUJESZ (musi być konkretny)**  
**type x = new class()**

Najprawdopodobniej myślisz teraz: „NATURALNIE — klasa zawsze jest klasą, natomiast typ wcale nie musi być klasą, może być np. interfejsem. W tej sytuacji jest OCZYWISTE, że atrybutu „class” używa się do reprezentowania elementów, które ZAWSZE muszą być klasami, natomiast atrybut „type” może równie dobrze reprezentować interfejsy”. Cóż, masz rację, ale zapewne masz też pewne wątpliwości: „Oczywiście nie WSZYSTKO w oficjalnej specyfikacji musi być takie intuicyjne i oczywiste, więc zawsze lepiej się upewnić”. Czasami (jak choćby w przypadku znacznika zabezpieczeń `<auth-constraint>`) nazwa elementu wskazuje na dokładnie odwrotne znaczenie niż w rzeczywistości. Jednak w tym przypadku klasa jest klasą, zaś typ jest... typem.



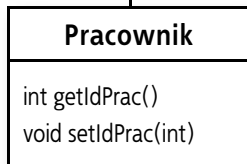
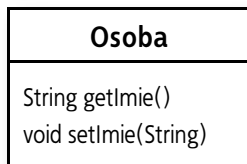
### BĄDŹ kontenerem

Przyjrzyj się następującej standardowej akcji:

```
<jsp:useBean id="osoba" class="foo.Pracownik" scope="request" >
 <jsp:setProperty name="osoba" property="imie" value="Fred" />
</jsp:useBean >
```

Imię: **<jsp:getProperty name="osoba" property="imie" />**

klasa abstrakcyjna



klasa konkretna

(Obie klasy należą do pakietu "foo").

Wyobraź sobie teraz, że serwlet przetwarza otrzymane dane i przekazuje swoje żądanie do strony JSP zawierającej powyższy fragment kodu.

Spróbuj określić, jaki będzie efekt wykonania naszego kodu JSP dla każdego z dwóch różnych przykładów kodu serwletu. (Rozwiązanie zadania znajdziesz na końcu rozdziału).

❶ Jaki będzie efekt wykonania następującego kodu serwletu:

```
foo.Osoba p = new foo.Pracownik();
p.setImie("Evan");
request.setAttribute("osoba", p);
```

❷ Jaki będzie efekt wykonania następującego kodu serwletu:

```
foo.Osoba p = new foo.Osoba();
p.setImie("Evan");
request.setAttribute("osoba", p);
```

Tak sobie myślę...  
przypuśćmy, że nie używamy  
serwletu kontrolera i że akcja  
formularza HTML trafia prosto do  
naszej strony JSP... czy można użyć  
parametrów żądania do właściwego  
ustawiania właściwości komponentu  
BEZ stosowania elementów  
skryptowych?



## Przekazywanie żądania wprost do strony JSP bez pośrednictwa serwletu...

Wyobraźmy sobie, że udostępniamy użytkownikowi następujący formularz HTML:

```
<html><body>
```

```
<form action="KomponentTestowy.jsp">
```

```
 imię: <input type="text" name="imieUzytkownika">
```

```
 ID: <input type="text" name="idUzytkownika">
```

```
 <input type="submit">
```

```
</form>
```

```
</body></html>
```

Żądanie trafia PROSTO  
do strony JSP.

Wiemy, że można to zadanie zrealizować, stosując kombinację standardowych akcji i elementów skryptowych:

```
<jsp:useBean id="osoba" type="foo.Osoba" class="foo.Pracownik" />
```

```
<% osoba.setNazwa(request.getParameter("imieUzytkownika")); %>
```

Możemy nawet umieścić elementy skryptowe WEWNĄTRZ akcji standardowej:

```
<jsp:useBean id="osoba" type="foo.Osoba" class="foo.Pracownik">
```

```
 <jsp:setProperty name="osoba" property="imie"
```

```
 value="<%= request.getParameter("imieUzytkownika") %>" />
```

```
</jsp:useBean>
```

Tak, WIDZISZ wyrażenie WEWNĄTRZ znacznika `<jsp:setProperty>` (który w tym przypadku został umieszczony w ciele znacznika `<jsp:useBean>`).

Tak, masz rację, FAKTYCZNIE nie wygląda to najlepiej.

# Rozwiązaniem jest użycie atrybutu param

To takie proste. Za pomocą atrybutu *param* możesz przekazywać parametry żądania prosto do komponentu, bez elementów skryptowych.

**Atrybut *param* umożliwia nam przypisywanie do właściwości komponentu wartości z parametru żądania. WYSTARCZY odpowiednio nazwać ten parametr!**

## W kodzie pliku KomponentTestowy.jsp

```
<jsp:useBean id="osoba" type="foo.Osoba" class="foo.Pracownik">
 <jsp:setProperty name="osoba" property="imie" param="imieUzytkownika" />
</jsp:useBean>
```

```
<html><body>

<form action="KomponentTestowy.jsp">
 imię: <input type="text" name="imieUzytkownika">
 ID: <input type="text" name="idUzytkownika">
 <input type="submit">
</form>

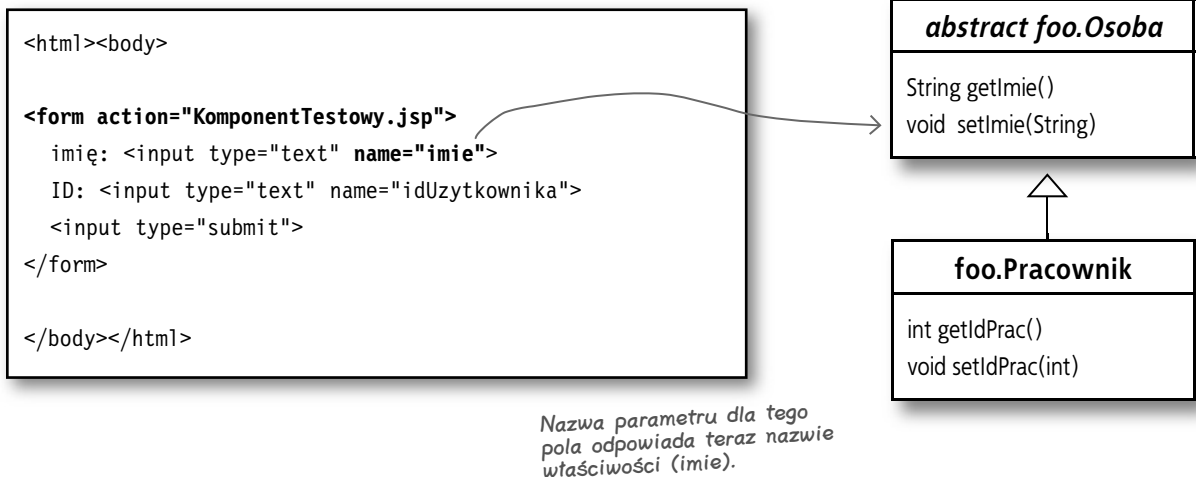
</body></html>
```

Wartość parametru „nazwaUzytkownika” pochodzi z atrybutu name pola input naszego formularza HTML.

## Zaczekaj! Mam jeszcze lepszy pomysł...

Musisz się tylko upewnić, że użyta w formularzu HTML *nazwa pola input* (która staje się nazwą parametru żądania) jest taka sama jak *nazwa właściwości* w Twoim komponencie. W takim przypadku nie będziesz już musiał definiować atrybutu *param* w znaczniku `<jsp:setProperty>`. Jeśli odpowiednio nazwiesz swoją *właściwość*, ale nie określisz ani atrybutu *value*, ani atrybutu *param*, wymusisz na kontenerze odczytanie odpowiedniej wartości z *parametru żądania* oznaczonego pasującą nazwą.

**Jeśli zmodyfikujemy kod HTML w taki sposób, że nazwa pola input będzie odpowiadała nazwie właściwości:**



## Musimy zrobić TO

```

<jsp:useBean id="osoba" type="foo.Osoba" class="foo.Pracownik">
 <jsp:setProperty name="osoba" property="imie" />
</jsp:useBean>

```

↑  
Nie zdefiniowaliśmy  
ŻADNEJ wartości!

**Jeśli nazwa parametru żądania pasuje do nazwy właściwości komponentu, nie musisz w znaczniku `<jsp:setProperty>` definiować wartości dla tej właściwości!**

### Jeśli możesz rozszerzyć ten mechanizm, całe rozwiązanie będzie jeszcze LEPSZE...

Zobacz, co się stanie, jeśli dostosujemy nazwy WSZYSTKICH parametrów żądania w taki sposób, aby odpowiadały nazwom odpowiednich właściwości komponentu. Komponent *osoba* (będący egzemplarzem klasy `foo.Pracownik`) w rzeczywistości zawiera dwie właściwości — imię i `idPrac`.

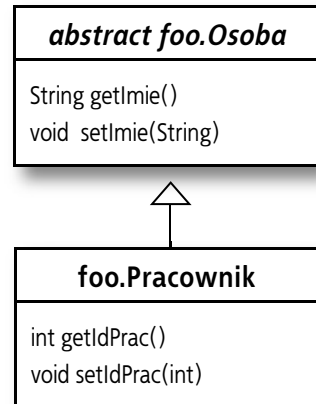
#### Jeśli ponownie zmienimy nasz kod HTML

```
<html><body>

<form action="KomponentTestowy.jsp">
 imię: <input type="text" name="imie">
 ID: <input type="text" name="idPrac">
 <input type="submit">
</form>

</body></html>
```

Teraz nazwy OBU parametrów pasują do nazw właściwości danego komponentu.



#### Otrzymamy to

```
<jsp:useBean id="osoba" type="foo.Osoba" class="foo.Pracownik">
 <jsp:setProperty name="osoba" property="*" />
</jsp:useBean>
```

Czyż to nie wspaniałe??

Chcę, żebyś kolejno przejrzał każdy z parametrów żądania, znalazł wszystkie te, które pasują do nazw właściwości tego komponentu, i tak ustawił WARTOŚCI pasujących właściwości, aby były równe wartościom odpowiednich parametrów żądania...



JSP

Jasne... znowu wszystko spada na MOJE barki. Muszę się przyjrzeć metodom zwracającym i ustawiającym klasy komponentu, aby określić jego właściwości. Następnie muszę jeszcze dopasować nazwy tych właściwości do nazw parametrów dołączonych do żądania...



Kontener

## Znaczniki komponentu automatycznie konwertują właściwości proste

Jeśli masz już jakieś doświadczenie w pracy z komponentami JavaBeans, ta własność nie powinna być dla Ciebie zaskoczeniem. Właściwości komponentów JavaBeans *mogą* być *czymkolwiek*, ale jeśli mają postać łańcuchów lub typów prostych, wszelkie działania związane z dopasowywaniem typów są podejmowane automatycznie.

To prawda — nie musisz poddawać analizie składniowej ani konwertować tych właściwości w swoim kodzie JSP.

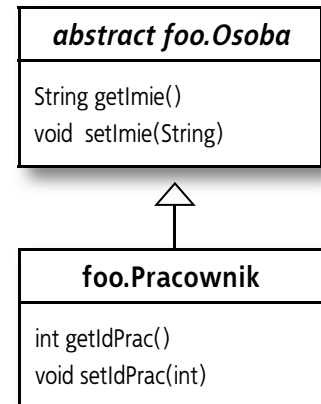
### Jeśli użyjemy typu *Pracownik* (zamiast typu *Osoba*)

```
<html><body>

<jsp:useBean id="osoba" type="foo.Pracownik" class="foo.Pracownik">
 <jsp:setProperty name="osoba" property="*" />
</jsp:useBean>

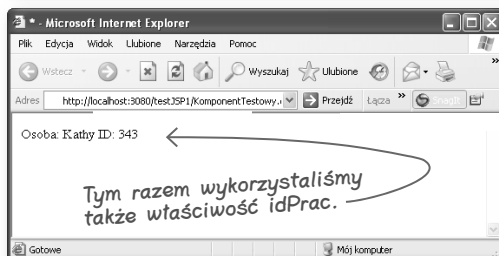
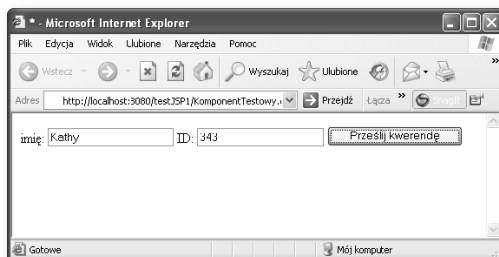
Osoba: <jsp:getProperty name="osoba" property="imie" />
ID: <jsp:getProperty name="osoba" property="idPrac" />

</body></html>
```



Teraz wygenerowany serwet będzie mówił:  
*Pracownik osoba = new Pracownik();* zamiast  
*Osoba osoba = new Pracownik();*

### Wszystko będzie działało prawidłowo



Akcja `<jsp:setProperty>` otrzymuje parametr żądania typu `String`, konwertuje go na liczbę całkowitą typu `int` i przekazuje tę liczbę metodzie ustawiającej daną właściwość w ramach komponentu `JavaBean`.

### Nie ma niemądrych pytań

**P:** No dobrze, myślę jednak, że kod kontenera wykonuje coś w rodzaju `Integer.parseInt(„343”)`. Czy w związku z tym nie otrzymamy wyjątku `NumberFormatException`, jeśli użytkownik wpisze coś, czego nie można przekształcić w wartość typu `int`? Co będzie, jeśli użytkownik wpisze np. „trzy” w polu identyfikatora pracownika?

**U:** Dobre spostrzeżenie. To prawda, jeśli konwersja parametru żądania dla właściwości `idPrac` na liczbę typu `int` nie będzie możliwa, z pewnością tego typu działania skończą się błędem. Musisz sprawdzać poprawność zawartości tego pola, aby zawsze mieć pewność, że zawiera ono wyłącznie znaki numeryczne. Mógłbyś np. przesłać dane z formularza do odpowiedniego serwletu, zamiast przekazywać je bezpośrednio do strony JSP. Jeśli jednak jesteś przywiązany do idei wysyłania żądań wprost do stron JSP i nie chcesz stosować elementów skryptowych, powinieneś w swoim formularzu HTML użyć odpowiednich funkcji języka JavaScript do weryfikacji tego pola jeszcze *przed* wysłaniem żądania. Jeśli nie masz żadnego doświadczenia w programowaniu w tym języku (który oczywiście nie ma NIC wspólnego z Javą), nie powinieneś się tym przejmować — to bardzo prosty język skryptowy, który przetwarza dane po stronie klienta (jego instrukcje są więc wykonywane przez przeglądarkę internetową). Wpisując wyrażenie `JavaScript validate input field` w wyszukiwarce Google, można błyskawicznie odnaleźć szereg gotowych skryptów, których możesz z powodzeniem użyć do powstrzymywania użytkowników przed wpisywaniem w polach wejściowych znaków innych niż cyfry.

**P:** Skoro właściwość komponentu nie musi być ani łańcuchem, ani typem prostym, JAK można ustawić tę właściwość bez stosowania elementów skryptowych? Przecież atrybut `value` znacznika zawsze jest łańcuchem, prawda?

**U:** Istnieje możliwość (choć wymaga dużej ilości dodatkowych nakładów) utworzenia specjalnej klasy określonej mianem niestandardowego edytora właściwości i zapewniającej dodatkowe wsparcie naszemu komponentowi. Taka klasa otrzymuje na wejściu Twoją wartość typu `String`, po czym określa, jak można ją przekonwertować na coś, co będzie można następnie wykorzystać do ustawienia jakiegoś bardziej skomplikowanego typu. Wspomniany mechanizm należy jednak do specyfikacji komponentów `JavaBeans`, a nie interesującej nas technologii JSP. Co więcej, jeśli atrybut `value` znacznika `<jsp:setProperty>` jest *wyrażeniem*, a nie stałą łańcuchową, i JEŚLI wartością tego wyrażenia jest obiekt zgodny z typem właściwości komponentu, próba wykonania takiego kodu prawdopodobnie nie spowoduje błędów. Jeśli prześlemy wyrażenie, którego wynikiem jest np. obiekt klasy `Pies`, zostanie wywołana metoda `setPies(Pies)` komponentu `Osoba`. Zastanów się nad tym — takie rozwiązanie oznacza, że odpowiedni obiekt klasy `Pies` musi w tym momencie istnieć. Tak czy inaczej byłoby lepiej, gdybyś NIE próbował konstruować nowych elementów w swoim kodzie JSP! Wszelkie próby konstruowania i ustawiania choćby najprostszych typów danych bez pomocy elementów skryptowych bywają bardzo trudne. (Co więcej, żadnego z tych zagadnień z pewnością nie spotkasz na egzaminie).



Oglądasz to!

**Automatyczna konwersja łańcuchów na typy proste NIE będzie prawidłowo funkcjonowała, jeśli użyjesz elementów skryptowych!! Próba wykonania takiego kodu zakończy się niepowodzeniem nawet wtedy, gdy odpowiednie wyrażenie znajdzie się WEWNĄTRZ znacznika `<jsp:setProperty>`.**

Jeśli używasz znacznika standardowej akcji `<jsp:setProperty>` z symbolem wieloznacznym w atrybucie `property` LUB z samą nazwą właściwości bez podania jej wartości lub atrybutu `param` (co oznacza, że nazwa właściwości pasuje do nazwy parametru żądania), LUB używasz atrybutu `param` do określenia parametru żądania, którego wartość powinna zostać przypisana do właściwości danego komponentu, LUB jeśli stosujesz wartość stałą, mechanizm automatycznej konwersji łańcucha na liczbę typu `int` zadziała. Każdy z poniższych przykładów spowoduje właśnie automatyczną konwersję typów:

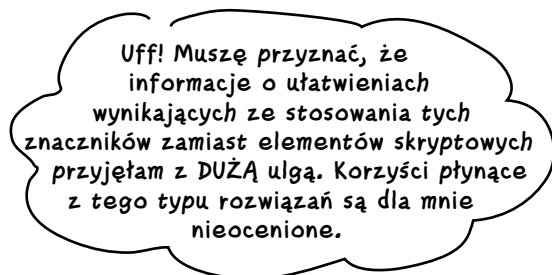
```
<jsp:setProperty name="osoba" property="*" />
<jsp:setProperty name="osoba" property="idPrac" />
<jsp:setProperty name="osoba" property="idPrac" value="343" />
<jsp:setProperty name="osoba" property="idPrac" param="idPrac" />
```

To wszystko  
działa!

ALE... jeśli użyjesz w swoim kodzie elementów skryptowych, automatyczna konwersja NIE zadziała:

To NIE działa!

```
<jsp:setProperty name="osoba" property="idPrac" value="<%= request.getParameter("idPrac") %>" />
```



**Znaczniki standardowych akcji komponentu są znacznie bardziej naturalne dla osób, które na co dzień nie programują.**

Także w tym przypadku okazuje się, że korzyści wynikające ze stosowania znaczników zamiast elementów skryptowych dotyczą w większym stopniu projektantów stron internetowych niż Ciebie (programistę Javy). Choć trzeba przyznać, że nawet programiści Javy widzą, że znaczniki są prostsze w utrzymaniu niż trwale zakodowane elementy skryptowe Javy. Projektant stron WWW, który wykorzystuje w swojej pracy znaczniki związane z komponentami, musi dysponować tylko podstawowymi informacjami identyfikującymi (nazwą atrybutu, zasięgiem i nazwą właściwości). To prawda, w takim przypadku konieczna jest znajomość pełnej nazwy klasy, ale dla wielu projektantów jest to po prostu kilka wyrazów oddzielonych kropkami. Projektant stron internetowych nie musi posiadać żadnej wiedzy na temat rzeczywistych mechanizmów kryjących się za tymi nazwami i może traktować komponenty jak zwykle rekordy z polami. Zadaniem programisty jest przekazanie projektantom informacji o tym rekordzie (nazwy klasy i identyfikatora) oraz potrzebnych polach (właściwościach).

Standardowe akcje komponentów nadal jednak nie wyglądają tak elegancko, jak byśmy oczekiwali.

*I dlatego właśnie to jeszcze nie koniec naszej historii o bezskryptowych stronach JSP. Czytaj dalej...*

## Ale co będzie, jeśli wykorzystywana właściwość NIE jest ani łańcuchem, ani typem prostym?

Wiemy już, jak proste jest wyświetlanie na stronie wartości *atrybutu* w sytuacji, gdy atrybut ten ma postać zwykłego łańcucha. Udało nam się także stworzyć i wykorzystać atrybut, który nie był obiektem klasy String, tylko egzemplarzem komponentu Osoba. Do tej pory nie próbowaliśmy jednak wyświetlić tego atrybutu (osoba) — ograniczyliśmy się tylko do wyświetlania jego właściwości (w naszym przykładzie było to imię i identyfikator pracownika, czyli odpowiednio imie i idPrac). Wszystko działało prawidłowo, ponieważ zastosowana akcja standardowa może obsługiwać zarówno właściwości typu String, *jak i* właściwości typów prostych. Wiemy więc, że standardowe akcje mogą obsługiwać atrybuty dowolnego typu tak długo, jak długo wszystkie ich *właściwości* są łańcuchami lub typami prostymi.

Ale co będzie, jeśli ten warunek nie zostanie spełniony? Co się stanie, jeśli komponent będzie zawierał właściwości *niebędące* ani łańcuchami, ani typami prostymi? Co, jeśli dana właściwość będzie jeszcze jednym obiektem? Typem obiekowym z *własnymi właściwościami*?

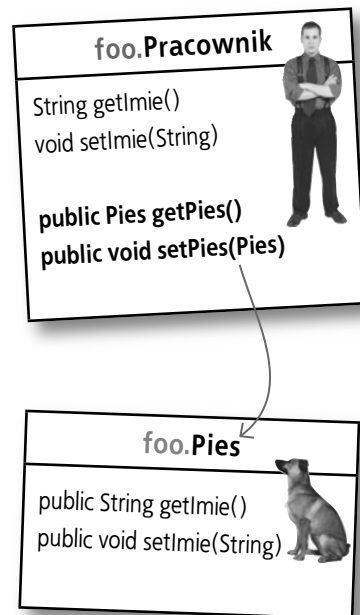
**Co będzie, jeśli zechcemy wyświetlić na stronie którąś z właściwości tej właściwości?**

**Osoba zawiera właściwość „imie” typu String.**

**Osoba zawiera właściwość „pies” typu Pies.**

**Pies zawiera właściwość „imie” typu String.**

Uwaga: W tym przykładzie Osoba jest konkretną klasą.



**Co powinniśmy zrobić, jeśli zaistnieje potrzeba wyświetlenia na stronie imienia psa danej Osoby?**

### Kod serwletu

```
public void doPost(HttpServletRequest request, HttpServletResponse response)
 throws IOException, ServletException {

 foo.Osoba o = new foo.Osoba();
 o.setImie("Evan");

 foo.Pies pies = new foo.Pies();
 pies.setImie("Spike");
 o.setPies(pies);
 request.setAttribute("osoba", o);

 RequestDispatcher view = request.getRequestDispatcher("wynik.jsp");
 view.forward(request, response);
}
```

Tym razem tworzymy obiekt klasy Pies, nadajemy mu imię i wywołujemy metodę setPies() klasy Osoba.

Teraz, kiedy obiekt klasy Osoba ma już obiekt klasy Pies w swojej właściwości „pies”, ustawiamy obiekt klasy Osoba (samą zmienną „o”) w roli atrybutu żądania.

## Próba wyświetlenia właściwości właściwości

Wiemy, że można zrealizować to zadanie za pomocą elementów skryptowych, ale czy można osiągnąć ten sam cel wyłącznie w oparciu o akcje standardowe? Co będzie, jeśli użyjemy nazwy „pies” w roli odczytywanej właściwości w ramach znacznika `<jsp:getProperty>`?

### Bez standardowych akcji (z wykorzystaniem elementów skryptowych)

```
<html><body>

<%= ((foo.Osoba) request.getAttribute("osoba")).getPies().getImie() %>

</body></html>
```

Wszystko działa doskonale...  
ale musieliśmy użyć elementów skryptowych.

### Z akcjami standardowymi (bez elementów skryptowych)

```
<html><body>

<jsp:useBean id="osoba" class="foo.Osoba" scope="request" />

Imię psa: <jsp:getProperty name="osoba" property="pies" />

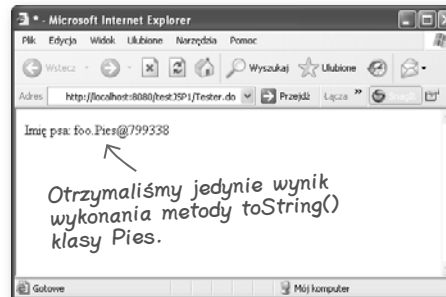
</body></html>
```

Ale co tak naprawdę jest  
wartością właściwości „pies”?

### Co **CHCEMY** otrzymać:



### Co **MAMY**:



### Nie możemy przecież powiedzieć: `property="pies.imię"`

Nie istnieje taka kombinacja standardowych akcji komponentu, która mogłaby prawidłowo funkcjonować dla oryginalnego kodu serwletu, ponieważ sam obiekt klasy `Pies` nie jest atrybutem! Jest tylko właściwością atrybutu, zatem możemy bez trudu wyświetlić obiekt typu `Pies`, ale nie możemy przejść do właściwości `imię` właściwości `Pies` atrybutu `Osoba`.

Standardowa akcja `<jsp:getProperty>` zapewnia nam dostęp tylko do właściwości atrybutu komponentowego. Nie mamy podobnych możliwości w przypadku właściwości zagnieżdżonych, a więc w sytuacji, gdy chcemy odczytać lub ustawić właściwość *właściwości* zamiast właściwości *atrybutu*.

## Uratował nas język wyrażeń (EL)

Tak, pomoc przyszła w samą porę; język wyrażeń JSP (ang. Expression Language — EL) został dodany do specyfikacji JSP 2.0, uwalniając nas od tyranii elementów skryptowych.

Zobacz, jak cudownie prosto może teraz wyglądać nasz kod JSP...

### Kod JSP bez elementów skryptowych, ale z wyrażeniem EL

```
<html><body>
```

```
 Imię psa: ${osoba.pies.imie}
```

```
</body></html>
```

*O to nam chodziło! Nie musieliśmy nawet deklarować, co oznacza słowo „osoba”... kontener po prostu to wie.*

**Język wyrażeń (EL) ułatwia wyświetlanie zagnieżdżonych właściwości... czyli po prostu właściwości właściwości!**

### Tym:

```
${osoba.pies.imie}
```

### Zastąpiliśmy to:

```
<%= ((foo.osoba) request.getAttribute("osoba")).getPies().getImie() %>
```



**Relax** Nie musisz wiedzieć WSZYSTKIEGO o języku EL

Na egzaminie nikt nie będzie od Ciebie oczekiwał kompletnej znajomości języka wyrażeń (EL). Wszystkie informacje, których będziesz potrzebował w swojej pracy i z których możesz być odpytywany na egzaminie, zostaną omówione na kilku kolejnych stronach. Jeśli więc chcesz studiować specyfikację tego języka, na razie się z tym wstrzymaj. Dla jasności — nikogo do opanowywania tej specyfikacji nie namawiamy.

## Dekonstrukcja języka wyrażeń JSP (EL)

Składnia i zakres języka wyrażeń EL są nieprzystojnie proste. Jedyną częścią tego języka, na którą należy zwrócić uwagę, są instrukcje przypominające Javę, ale zachowujące się w nieco inny sposób. Przekonasz się o tym już za moment, kiedy dojdziemy do operatora []. Będziesz miał do czynienia z konstrukcjami niedopuszczalnymi w Javie, ale akceptowanymi w języku EL, i odwrotnie. Jeśli nie będziesz próbował na siłę stosować reguł językowych i składniowych Javy w wyrażeniach EL, nauka nowego języka powinna pójść sprawnie. Studiując tę i kilka następnych stron, staraj się myśleć o języku EL jak o jednym ze środków umożliwiających dostęp do obiektów języka Java *bez stosowania samej Javy*.

**Wyrażenia języka EL należy ZAWSZE umieszczać wewnątrz nawiasów klamrowych i poprzedzać znakiem dolara**

`${osoba.imie}`

**Pierwsza użyta w wyrażeniu zmienna nazwana jest albo obiektem domyślnym, albo atrybutem.**

`${pierwszyElement.drugiElement}`

**OBIEKT DOMYŚLNY EL**

LUB

**ATRYBUT**

pageScope  
requestScope  
sessionScope  
applicationScope

param  
paramValues

header  
headerValues

cookie

initParam

pageContext

w zasięgu strony  
w zasięgu żądania  
w zasięgu sesji  
w zasięgu aplikacji

*Jeśli pierwszym elementem w wyrażeniu języka EL jest atrybut, jego nazwa może wskazywać na atrybut składowany w dowolnym z czterech dostępnych zasięgów.*

*Wszystkie wymienione zmienne są obiektami mapy.*

*Ze wszystkich wymienionych obiektów domyślnych tylko obiekt pageContext nie jest obiektem mapy. Jest to rzeczywista referencja do obiektu pageContext! (A sam obiekt pageContext jest egzemplarzem klasy JavaBean).*

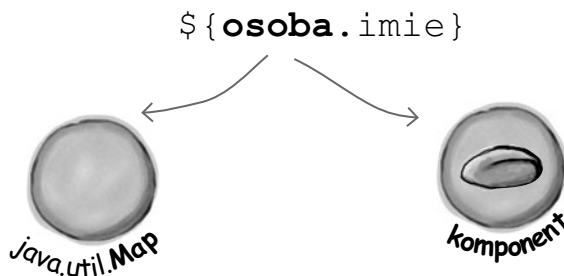
*Uwaga: Obiekty domyślne języka EL to nie to samo co obiekty domyślne udostępniane elementom skryptowym JSP, jedynym wyjątkiem jest właśnie obiekt pageContext.*

*(Przypomnienie dotyczące Javy: mapa jest zbiorem złożonym z par klucz-wartość, np. Hashtable oraz HashMap).*

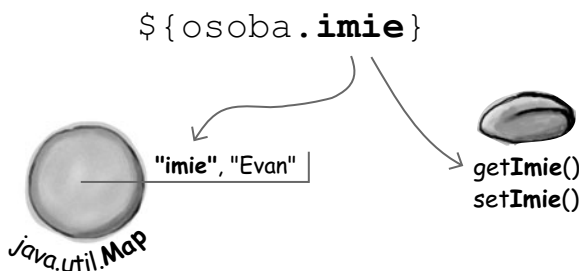
# Stosowanie operatora kropki (.) do uzyskiwania dostępu do właściwości oraz odwzorowywania wartości

Pierwsza zmienna jest albo obiektem domyślnym języka EL, albo atrybutem, natomiast element *na prawo* od kropki jest albo *wartością* mapy (jeśli pierwsza zmienna jest mapą), albo *właściwością* komponentu (jeśli pierwsza zmienna jest atrybutem będącym jednocześnie komponentem JavaBean).

- 1 **Jeśli wyrażenie zawiera zmienną znajdującą się bezpośrednio przed kropką, taka zmienna MUSI być albo mapą, albo komponentem.**



- 2 **Element na prawo od kropki MUSI być albo kluczem mapy, albo właściwością komponentu.**



- 3 **Element na prawo od kropki musi także być zgodny z normalnymi regułami Javy dotyczącymi nazewnictwa identyfikatorów.**

\${osoba.imie}

- \* Musi się rozpoczynać od litery, znaku podkreślenia ( \_ ) lub znaku dolara ( \$ ).
- \* Po pierwszym znaku mogą występować także cyfry.
- \* Nie może to być słowo kluczowe Javy.

Kiedy na lewo od kropki znajduje się zmienna, musi to być albo egzemplarz klasy Map (czyli coś z kluczami), albo komponent (czyli coś z właściwościami).

Powyższa reguła jest prawdziwa niezależnie od tego, czy dana zmienna jest obiektem domyślnym czy atrybutem.

Obiekt domyślny pageContext jest komponentem — udostępnia metody zwracające wartości swoich właściwości. Wszystkie pozostałe obiekty domyślne są mapami.

Jeśli dany obiekt jest komponentem i jeśli nie istnieje odpowiednia właściwość nazwana, zostanie wygenerowany wyjątek.

## Operator [] jest jak operator kropki, tyle że znacznie lepszy

Operator kropki można stosować tylko wtedy, gdy elementem po jego prawej stronie jest albo właściwość komponentu, albo klucz mapy (odpowiednio dla samego komponentu lub mapy na lewo od kropki). To wszystko. Okazuje się, że operator [] daje znacznie szersze możliwości i zapewnia większą elastyczność...

**To:**

```
${osoba.["imie"]}
```

**Ma takie samo znaczenie  
jak to:**

```
${osoba.imie}
```

**Prosta wersja operatora kropki działa,  
ponieważ *osoba* jest komponentem, zaś  
*imie* jest właściwością komponentu *osoba*.**

**Ale co powinniśmy zrobić, jeśli *osoba*  
będzie *tablicą*?**

**A jeśli *osoba* będzie *listą*  
(egzemplarzem klasy List)?**

**A jeśli *imie* będzie czymś, czego nie  
można wyrazić zgodnie z normalnymi  
regułami nazewnictwa Javy?**

Przecież to wcale nie  
wygląda lepiej. Przeciwnie,  
wygląda na to, że nowe  
rozwiązanie wymaga tylko  
dodatkowego dopisywania nawiasów  
i cudzysłowów...

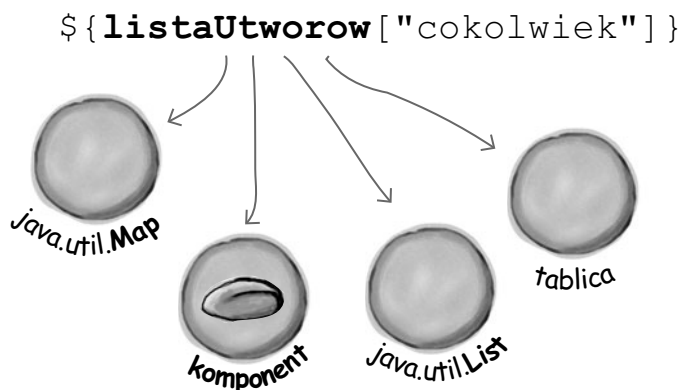


Operator `.` jest jednak lepszy

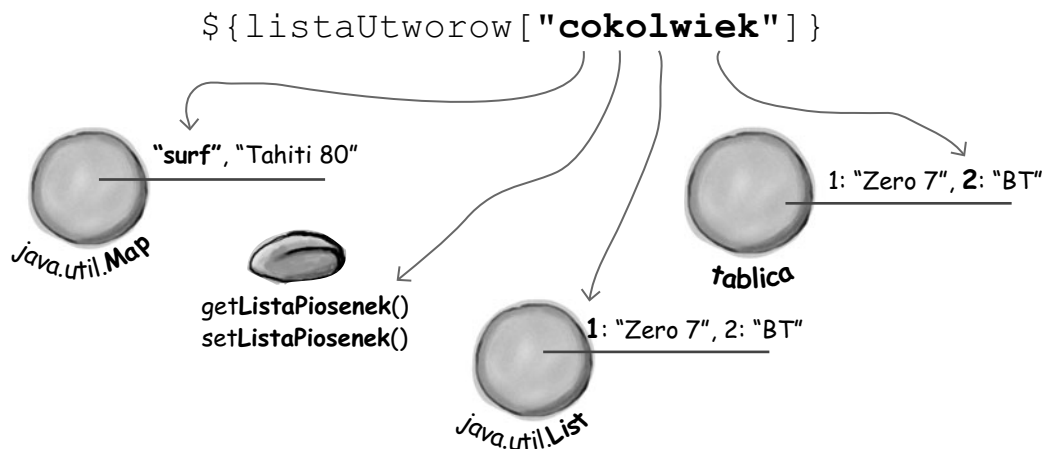
## Operator `[]` daje więcej możliwości...

Kiedy używasz operatora kropki, po lewej stronie możesz umieścić tylko egzemplarz klasy `Map` lub komponentu, natomiast prawa strona musi spełniać reguły nazewnictwa identyfikatorów w Javie. Nieco inaczej jest w przypadku operatora `[]`, gdzie element po lewej stronie może być także listą (obiektem klasy `List`) lub tablicą (dowolnego typu). To zaś oznacza, że prawa strona może być liczbą całkowitą, dowolnym wyrażeniem, którego wynikiem jest liczba całkowita, lub identyfikatorem, który nie musi spełniać stosowanych w Javie reguł nazewnictwa. Przykładowo, operator `[]` umożliwia użycie klucza obiektu `Map` w formie łańcucha zawierającego kropki (`"com.foo.trouble"`).

- ❶ **Jeśli dane wyrażenie zawiera zmienną poprzedzającą nawiasy kwadratowe `[]`, zmienna po lewej stronie może być egzemplarzem klasy `Map`, komponentem, egzemplarzem klasy `List` lub tablicą.**



- ❷ **Jeśli składnik wewnątrz nawiasów kwadratowych jest stałą łańcuchową (tj. ciągiem znaków w cudzysłowie), może to być klucz mapy lub właściwość komponentu; w przeciwnym przypadku jest to indeks elementu listy (egzemplarza klasy `List`) lub tablicy.**



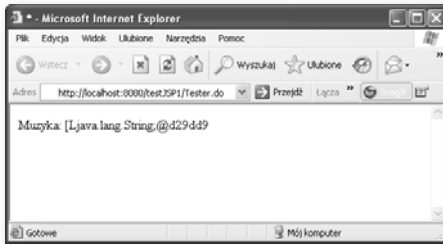
# Stosowanie operatora [] z tablicą

## W serwlecie

```
String[] ulubionaMuzyka = {"Zero 7", "Tahiti 80", "BT", "Frou Frou"};
request.setAttribute("listaUtworow", ulubionaMuzyka);
```

## W JSP

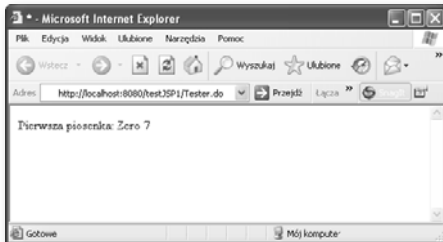
Muzyka: `${listaUtworow}`



To rozumiem... wywołuje metodę `toString()` dla tablicy.

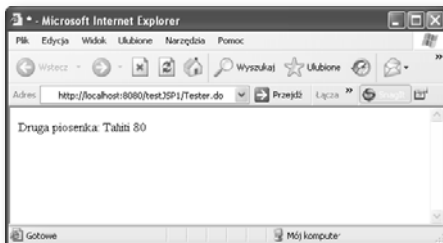
Pierwsza piosenka: `${listaUtworow[0]}`

Przecież to oczywiste...



To chyba jakiś żart, prawda? A może w tym drinku jest coś więcej niż sam poncz? Słuchaj stary, mógłbym PRZYSIĄC, że widziałem cudzystowy otaczające indeks tablicy, a to przecież niedorzeczne...

Druga piosenka: `${listaUtworow["1"]}` Co to ma znaczyć?



# Indeksy łańcuchowe tablic i list są automatycznie konwertowane na liczby całkowite

W języku EL wyrażenia uzyskujące dostęp do *tablicy* nie różnią się od wyrażen uzyskujących dostęp do *listy* (obiektu klasy `List`).

Pamiętajcie, to NIE jest Java. W języku wyrażen EL operator `[]` NIE jest operatorem dostępu do tablic. Nie, to po prostu operator `[]`. (Przysięgamy, zresztą możesz sam to sprawdzić w specyfikacji — operator `[]` nie ma nazwy! Ma tylko symbol `[]`. Tak jak Prince, przynajmniej w pewnym sensie). Gdyby ten operator MIAŁ nazwę, musielibyśmy mówić o operatorze dostępu do tablic/list/map/właściwości komponentów.

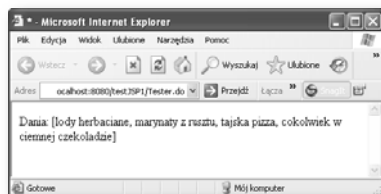
## W serwlecie

```
java.util.ArrayList ulubioneDania = new java.util.ArrayList();
ulubioneDania.add("lody herbaciane");
ulubioneDania.add("marynaty z rusztu");
ulubioneDania.add("tajska pizza");
ulubioneDania.add("cokolwiek w ciemnej czekoladzie");
request.setAttribute("ulubioneDania", ulubioneDania);
```

## W JSP

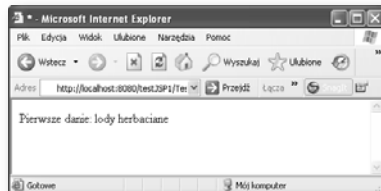
Dania: `${ulubioneDania}`

Klasa `ArrayList` zawiera oczywiście nadpisaną metodę `toString()`.



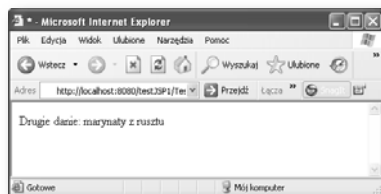
Pierwsze danie: `${ulubioneDania[0]}`

*dobrze*



Drugie danie: `${ulubioneDania["1"]}`

*Bardzo to dziwne, ale niech będzie... skoro takie wyrażenie jest poprawne i generuje właściwy wynik, będę musiał się do tego przyzwyczaić.*



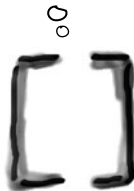
Jeśli składnik wyrażenia umieszczony na lewo od nawiasów jest tablicą lub listą, a użyty indeks jest stałą łańcuchową, indeks zostanie przekonwertowany na typ `int`.

Jednak takie wyrażenie NIE zadziała:

`${ulubioneDania[„jeden"]}`

Ponieważ łańcucha „jeden” nie da się przekształcić na liczbę całkowitą, to jeśli konwersja indeksu nie będzie możliwa, otrzymamy błąd.

Spójrzmy prawdzie w oczy, kropeczko. Jestem dużo lepszy od ciebie. Wiesz co o tobie piszą w specjalizacji? Nazwali cię „operatorem pomocniczym”. Moim zdaniem i tak potraktowali cię zbyt łagodnie.



Daj spokój, zastanów się lepiej, czy KTOKOLWIEK używa jeszcze tablic? Tablice i listy są takie... Stare. Liniowe. Nudne.



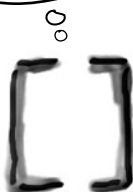
Jaaasne... tablice mieszające też pewnie nie są stosowane od czasów kamienia łupanego.



Już sam fakt, że wspominasz o tablicach mieszających, dowodzi, jaki JESTEŚ nowoczesny. Przecież takie tablice, podobnie jak inne struktury liniowe, są po prostu przestarzałe. Miałam na myśli raczej mapy i komponenty JavaBean. Każdy szanujący się programista korzysta obecnie tylko z tej pary struktur.



Sprawdzałeś może ostatnio datę opracowania specyfikacji komponentów JavaBean? Gdyby ta specyfikacja była kartonem z mlekiem, miałbyś już w środku wyhodowanego stworka podobnego do tych z Archiwum X...



Nie ma sensu z tobą gadać, nic nie rozumiesz.



# W przypadku komponentów i map można z powodzeniem stosować dowolny operator

Do uzyskiwania dostępu do komponentów JavaBean i map możesz używać zarówno operatora [], jak i pomocniczego operatora **kropki**. Wystarczy, że będziesz traktował klucze mapy w taki sam sposób jak nazwy właściwości należących do komponentu.

W odpowiedzi na żądanie klucza lub nazwy właściwości otrzymasz wartość odpowiedniego klucza lub właściwości.

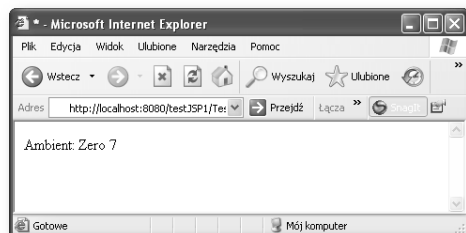
## W serwlecie

```
java.util.Map mapaMuzyki = new java.util.HashMap();
mapaMuzyki.put("Ambient", "Zero 7");
mapaMuzyki.put("Surf", "Tahiti 80");
mapaMuzyki.put("DJ", "BT");
mapaMuzyki.put("Indie", "Travis");
request.setAttribute("mapaMuzyki", mapaMuzyki);
```

*Stwórz mapę, umieść w niej kilka par tańcuchów kluczy i obiektów, po czym ustaw tę mapę w roli atrybutu żądania.*

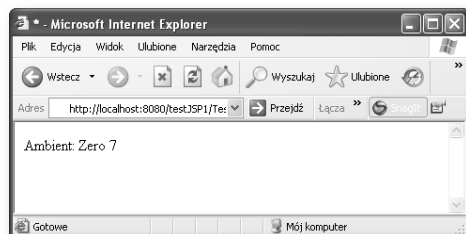
## W JSP

Ambient: \${mapaMuzyki.Ambient}



*Oba wyrażenia używają słowa Ambient w roli klucza danej mapy (ponieważ obiekt mapaMuzyki jest egzemplarzem klasy implementującej interfejs Map!).*

Ambient: \${mapaMuzyki["Ambient"]}



## Jeśli indeks NIE jest stałą łańcuchową, jego wartość jest wyznaczana automatycznie

Jeśli wewnątrz nawiasów kwadratowych nie ma cudzysłowów, kontener wyznacza wartość umieszczonego tam wyrażenia przez znalezienie atrybutu powiązanego z użytą nazwą i wstawia w jej miejsce odpowiednią wartość. (Jeśli istnieje obiekt domyślny oznaczony taką samą nazwą, zawsze zostanie użyty właśnie ten obiekt, a nie atrybut).

Muzyka: \${mapaMuzyki[Ambient]}

**Znajdź atrybut nazwany „Ambient”.  
Użyj WARTOŚCI tego atrybutu w roli klucza  
w danej mapie lub zwróć wartość null.**

Zastosowanie nazwy Ambient bez cudzysłowów NIE przyniesie oczekiwanego efektu!! Wynika to z faktu, że nie istnieje powiązany atrybut nazwany „Ambient”, zatem zostanie zwrócona wartość null.

### W serwlecie

```
java.util.Map mapaMuzyki = new java.util.HashMap();
mapaMuzyki.put("Ambient", "Zero 7");
mapaMuzyki.put("Surf", "Tahiti 80");
mapaMuzyki.put("DJ", "BT");
mapaMuzyki.put("Indie", "Travis");
```

```
request.setAttribute("mapaMuzyki", mapaMuzyki);
```

```
request.setAttribute("Rodzaj", "Ambient");
```

### PRAWIDŁOWE rozwiązanie w kodzie JSP

Muzyka: \${mapaMuzyki[Rodzaj]} ← jest przekształcane na → Muzyka: \${mapaMuzyki["Ambient"]}

ponieważ ISTNIEJE atrybut żądania nazwany „Rodzaj” z wartością „Ambient”, a łańcuch „Ambient” jest kluczem w obiekcie mapy mapaMuzyki.

### NIEPRAWIDŁOWE rozwiązanie w kodzie JSP (dla przedstawionego kodu serwletu)

Muzyka: \${mapaMuzyki["Rodzaj"]} ← nie ulega zmianie → Muzyka: \${mapaMuzyki["Rodzaj"]}

ponieważ w mapie mapaMuzyki NIE ISTNIEJE klucz nazwany „Rodzaj”. Z powodu użytych cudzysłowów kontener nawet nie próbuje wyznaczyć jego wartości — po prostu zakłada, że programista świadomie użył stałej nazwy klucza.

Jest to, co prawda, prawidłowe wyrażenie języka EL, ale nie przynosi oczekiwanego efektu.

# W nawiasach kwadratowych można umieszczać wyrażenia zagnieżdżone

Obsługa wyrażeń była głównym powodem stworzenia języka EL. Okazuje się, że język ten umożliwia zagnieżdżanie wyrażeń na dowolnym poziomie. Innymi słowy, możemy umieszczać skomplikowane wyrażenia wewnątrz skomplikowanych wyrażeń wewnątrz... (i tak dalej). Warto przy tym pamiętać, że wartości wyrażeń są wyznaczane od najgłębiej zagnieżdżonych do tych znajdujących się na najwyższym poziomie.

Prezentowane tutaj elementy języka EL powinny stanowić dla Ciebie najbardziej intuicyjną jego część, ponieważ mechanizm zagnieżdżania języka EL niczym się nie różni od mechanizmu stosowania nawiasów w kodzie Javy. Najtrudniejsze jest w tym przypadku uważne stosowanie (lub *pomijanie*) cudzysłówów.

## W serwlecie

```
java.util.Map mapaMuzyki = new java.util.HashMap();
mapaMuzyki.put("Ambient", "Zero 7");
mapaMuzyki.put("Surf", "Tahiti 80");
mapaMuzyki.put("DJ", "BT");
mapaMuzyki.put("Indie", "Travis");
request.setAttribute("mapaMuzyki", mapaMuzyki);

String[] rodzajeMuzyki = {"Ambient", "Surf", "DJ", "Indie"};
request.setAttribute("TypMuzyki", rodzajeMuzyki);
```

## PRAWIDŁOWE rozwiązanie w kodzie JSP

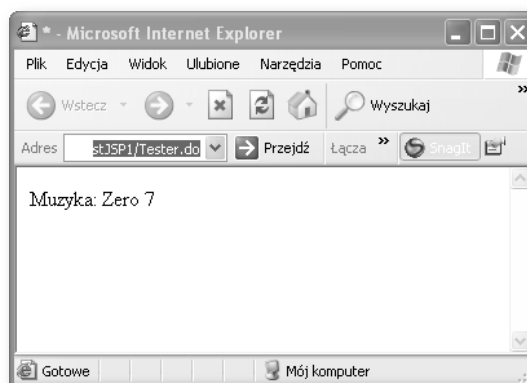
Muzyka: `${mapaMuzyki[TypMuzyki[0]]}`

↓ staje się

Muzyka: `${mapaMuzyki["Ambient"]}`

↓ staje się

Muzyka: **Zero 7**



## Nie możesz stosować wyrażen w postaci `${foo.1}`

Do uzyskiwania dostępu do obiektów komponentów i odwzorowań możesz używać operatora kropki, ale tylko wtedy, gdy składnik podawany przez Ciebie po kropce jest prawidłowym identyfikatorem Javy.

### To

```
${mapaMuzyki.Ambient} działa
```

### ma takie samo znaczenie jak to

```
${mapaMuzyki["Ambient"]} działa
```

### Ale tego

```
${listaUtworow["1"]}
```

### NIE można przekształcić w to

```
${listaUtworow.1} NIE! NIE! NIE!
```

**Jeśli jakiegos identyfikatora nie mógłbyś użyć jako nazwy zmiennej w swoim kodzie Javy, NIE umieszczaj go na prawo od operatora kropki.**

## Zaostrz ołówek



### Co zostanie wyświetlone?

Mając dany poniższy kod serwletu, spróbuj określić, co zostałoby wyświetlone na stronie (lub czy wystąpi jakiś błąd — w takim przypadku po prostu napisz „błąd”). Odpowiedzi znajdziesz na dole następnej strony.

```
java.util.ArrayList cyfry = new java.util.ArrayList();
cyfry.add("1");
cyfry.add("2");
cyfry.add("3");
request.setAttribute("liczby", cyfry);
String[] ulubionaMuzyka = {"Zero 7", "Tahiti 80", "BT", "Frou Frou"};
request.setAttribute("listaUtworow", ulubionaMuzyka);
```

❶ `${listaUtworow[liczby[0]]}`

❷ `${listaUtworow[liczby[0]+ 1]}`

❸ `${listaUtworow[liczby["2"]]}`

❹ `${listaUtworow[liczby[liczby[1]]]}`

(Więcej informacji na temat operatorów języka EL znajdziesz kilka stron dalej).



### Magnesiki z kodem

Jeśli spotkasz podobne ćwiczenie na egzaminie, wcale nie powinieneś się temu dziwić (z tą różnicą, że przykłady wykorzystywane podczas rzeczywistego egzaminu są... brzydsze).

Przeanalizuj zaprezentowaną na tej stronie trójkę klas, po czym uporządkuj magnesiki z kodem w taki sposób, aby utworzyły kod języka EL zwracający stronę odpowiedzi przedstawioną na zrzucie ekranu. (Odpowiedzi znajdziesz po odwróceniu strony, ale nie zaglądaj tam przed WYKONANIEM całego zadania, szczególnie jeśli masz zamiar przystąpić do prawdziwego egzaminu).

#### foo.Zabawka



```
package.foo;
public class Zabawka {
 private String nazwa;
 public void setNazwa(String
nazwa) {
 this.nazwa=nazwa;
 }
 public String getNazwa() {
 return nazwa;
 }
}
```

#### foo.Osoba



```
package.foo;
public class Osoba {
 private Pies pies;
 private String imie;
 public void setPies(Pies pies) {
 this.pies=pies;
 }
 public Pies getPies() {
 return pies;
 }
 public void setImie(String imie) {
 this.imie=imie;
 }
 public String getImie() {
 return imie;
 }
}
```

#### foo.Pies



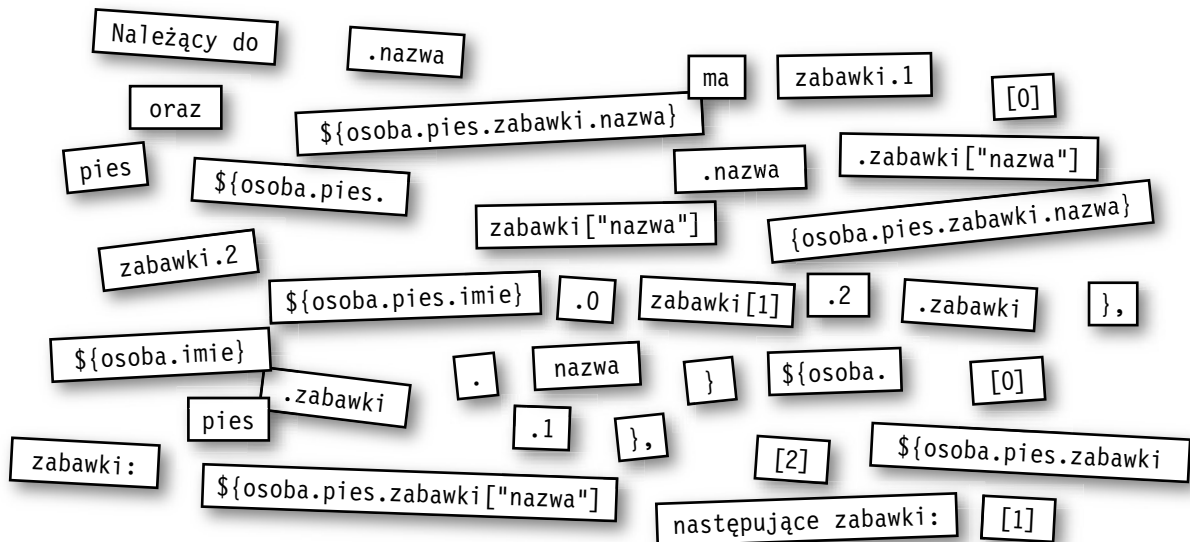
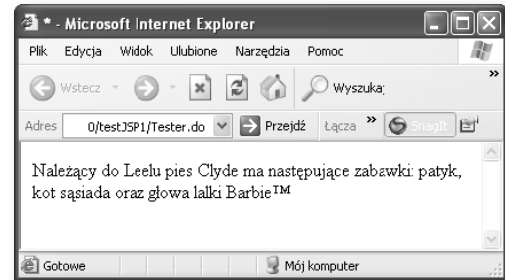
```
package.foo;
public class Pies {
 private String imie;
 private Zabawka[] zabawki;
 public void setImie(String imie) {
 this.imie=imie;
 }
 public String getImie() {
 return imie;
 }
 public void setZabawki(Zabawka[] zabawki) {
 this.zabawki=zabawki;
 }
 public Zabawka[] getZabawki() {
 return zabawki;
 }
}
```

Rozwiązanie ćwiczenia „Zaostrz ołówkę” z poprzedniej strony: 1) Tahiti 80, 2) BT, 3) Frou Frou, 4) Frou Frou.

## Kod serwletu

```
foo.Osoba o = new foo.Osoba();
o.setImie("Leelu");
foo.Pies p = new foo.Pies();
p.setImie("Clyde");
foo.Zabawka z1 = new foo.Zabawka();
z1.setNazwa("patyk");
foo.Zabawka z2 = new foo.Zabawka();
z2.setNazwa("kot sąsiada");
foo.Zabawka z3 = new foo.Zabawka();
z3.setNazwa("głowa lalki Barbie");
p.setZabawki(new foo.Zabawka[]{z1, z2, z3});
o.setPies(p);
request.setAttribute("osoba", p);
```

## Skonstruuj wyrażenie języka EL dla takich danych wyjściowych:





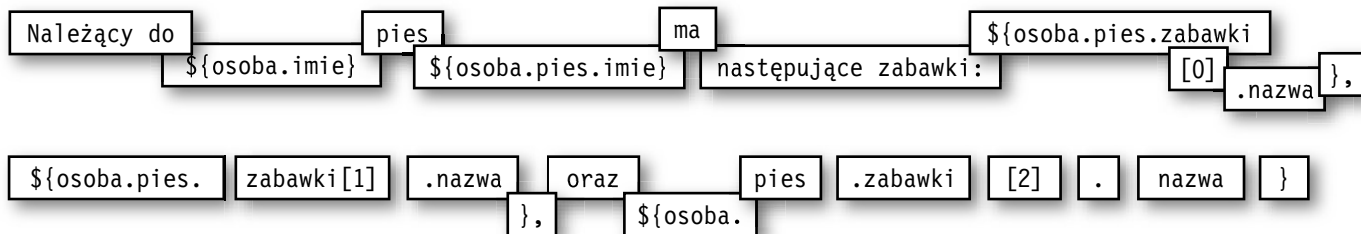
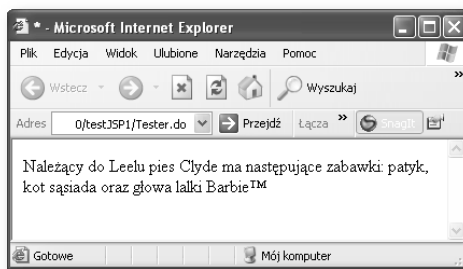
## Odpowiedzi do magnesików z kodem

Nie jest to **JEDYNY** sposób wygenerowania takich danych wyjściowych, ale inna realizacja tego zadania nie byłaby możliwa w oparciu o ten konkretny zbiór magnesików. Ćwiczenie dodatkowe: napisz nieco inne (zapomnij o magnesikach) wyrażenie języka EL, które wyświetli na stronie ten sam wynik.

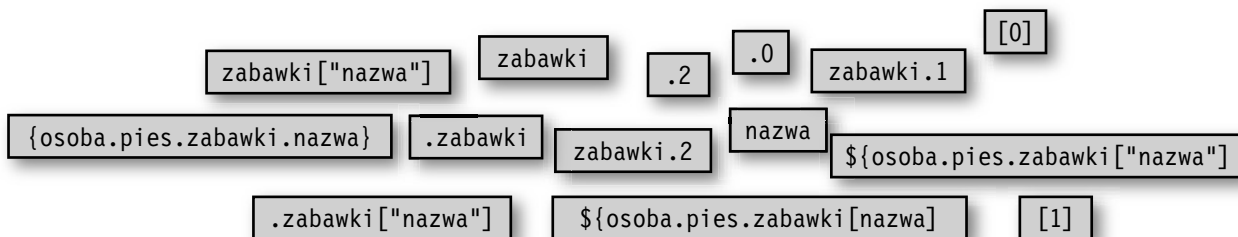
### Kod serwletu

```
foo.Osoba o = new foo.Osoba();
o.setImie("Leelu");
foo.Pies p = new foo.Pies();
p.setImie("Clyde");
foo.Zabawka z1 = new foo.Zabawka();
z1.setNazwa("patyk");
foo.Zabawka z2 = new foo.Zabawka();
z2.setNazwa("kot sąsiada");
foo.Zabawka z3 = new foo.Zabawka();
z3.setNazwa("głowa lalki Barbie™");
p.setZabawki(new foo.Zabawka[]{z1, z2, z3});
o.setPies(p);
request.setAttribute("osoba", p);
```

### Skonstruuj wyrażenie języka EL dla takich danych wyjściowych:



Należący do \${osoba.imie} pies \${osoba.pies.imie} ma następujące zabawki: \${osoba.pies.zabawki[0].nazwa}, \${osoba.pies.zabawki[1].nazwa} oraz \${osoba.pies.zabawki[2].nazwa}

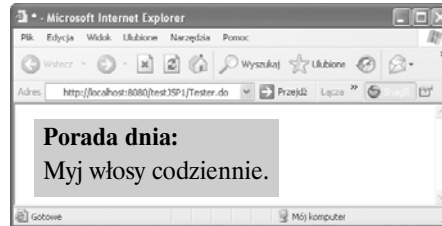


## Sprawa zaginionej treści

Zagadka na  
pięć minut



Specjaliści od obiegu dokumentów opracowali system zarządzania treścią wykorzystywany przede wszystkim do tworzenia podręczników dla użytkowników tradycyjnych aplikacji autonomicznych. Część tego systemu umożliwiała pracownikom odpowiedzialnym za przygotowywanie prezentowanych treści tworzenie fragmentów „Porada dnia” składowanych w atrybucie **currentTip** zasięgu żądania. Gdyby na przykład porada brzmiała „Myj włosy codziennie”, strona aplikacji zawierałaby następującą ramkę:

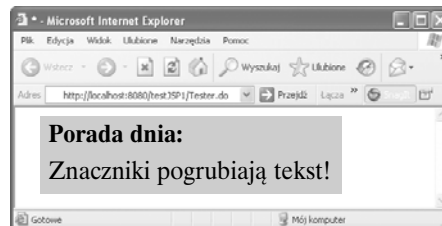


Kod JSP dla tej ramki ma następującą postać:

```
<div class='tipBox'>
 Porada dnia:

 ${pageContent.currentTip}
</div>
```

Przypuśćmy, że nowy klient próbuje za pomocą tego systemu opracować podręcznik użytkownika, ale nie radzi sobie z wyświetlaniem ramek porad dnia. Ku jego zdziwieniu, na przykład porada „Znaczniki **<b></b>** pogrubiają tekst!” jest wyświetlana w następujący sposób:



O co chodzi? Gdzie nasz klient popełnił błąd? Klient wezwał Towny’ego, naszego głównego programistę JSP, i wykrzyczał: „Gdzie się podziała część mojej porady? Dlaczego nie widać znaczników pogrubiających?”. Spostrzeżenia klienta natychmiast trafiły na listę błędów utrzymywaną przez twórców tego systemu.

**Co o tym myślisz? Czy znaczniki pogrubiające zostały przekazane do strumienia wyjściowego? Dlaczego nie zostały wyświetlone?**

# Wyrażenia EL wizualizują nieprzetworzony tekst, w tym kod HTML

Wszystko staje się jasne, kiedy zaglądamy do wygenerowanego kodu HTML.

```
<div class='tipBox'>
 Porada dnia:

 ${pageContent.currentTip}
</div>
```

## A oto wygenerowany kod HTML

```
<div class='tipBox'>
 Porada dnia:

 Znaczniki pogrubiają tekst!
</div>
```

Fragment „<b></b>” porady został co prawda wysłany w ramach strumienia wyjściowego, ale przeglądarka internetowa dokonała jego wizualizacji zupełnie tak, jakby miała do czynienia ze zwykłym kodem HTML, czyli pogrubiając pustą przestrzeń na stronie.

W tej sytuacji trudno się dziwić, że użytkownik nie widzi na ekranie znaczników „<b></b>”.

## To samo dotyczy znaczników wyrażeń JSP...

```
<div class='tipBox'>
 Porada dnia:

 <%= pageContent.getCurrentTip() %>
</div>
```

Niezależnie od wartości reprezentowanej przez to wyrażenie, ten fragment zostanie potraktowany jak zwykły kod HTML, zatem ewentualne znaczniki zostaną poddane standardowej wizualizacji (nie zostaną wyświetlone jako tekst).

## ... i standardowej akcji jsp:getProperty

```
<div class='tipBox'>
 Porada dnia:

 <jsp:getProperty name='pageContent' property='currentTip' %>
</div>
```

Z identyczną sytuacją mamy do czynienia także tym razem. Takie wyrażenia jak <i></i> czy <b></b> zostaną poddane wizualizacji, zatem przeglądarka nie wyświetli ich w oryginalnej formie tekstowej.



## WYTEŻ UMYŚŁ

No dobrze, zatem łańcuch porady jest przekazywany do strumienia wyjściowego, ale twórcy naszego systemu chcą konwertować znaki specjalne HTML-a na format umożliwiający prawidłową prezentację tych porad na ekranie. Aby użytkownik widział znak < w oknie przeglądarki, musimy użyć sekwencji „&lt;”. Aby wyświetlać znak >, musimy skorzystać z sekwencji „&gt;”.

### Jak to osiągnąć?

Pamiętaj,  
że akcja mojego formularza  
HTML trafia prosto do strony JSP.  
Czy istnieje wobec tego jakiś sposób  
przetwarzania parametrów żądania za  
pomocą samych wyrażeń  
języka EL?



## Obiekty domyślne języka EL

Nie zapominaj, że także język wyrażeń oferuje kilka własnych obiektów domyślnych. Obiekty te nie są jednak tożsame z obiektami niejawnymi JSP (z wyjątkiem jednego: `pageContext`). Poniżej przedstawiono krótką listę — część z tych obiektów omówimy bardziej szczegółowo na kilku następnych stronach. Przekonasz się, że wszystkie obiekty domyślne, poza jednym (znowu `pageContext`), są prostymi mapami, czyli parami nazwa-wartość.

`pageScope`

`requestScope`

*Mapy atrybutów  
zasięgów.*

`sessionScope`

`applicationScope`

`param`

*Mapy parametrów  
żądania.*

`paramValues`

`header`

`headerValues`

*Mapy nagłówków  
żądania.*

`cookie`

*Oooooch! To jest najcięższy przypadek...  
Czy to mapa... ciasteczek?*

`initParam`

*Mapy parametrów inicjalizacji kontekstu  
(NIE parametrów inicjalizacji serwletu!)*

`pageContext`

*Jedyny obiekt, który NIE jest mapą (obiektem implementującym interfejs `Map`). To wyjątkowy przypadek — rzeczywista referencja do obiektu `pageContext`, który możemy traktować jak komponent. Spójrz tylko na interfejs API dla metod zwracających obiekty typu `PageContext`.*

# Parametry żądania w języku EL

To proste. Obiekt domyślny param jest szczególnie przydatny wtedy, gdy wiesz, że dla konkretnej nazwy parametru istnieje dokładnie jeden parametr. Obiektu paramValues należy natomiast używać w sytuacji, gdy dla danej nazwy parametru może istnieć więcej niż jedna wartość.

## W formularzu HTML

```
<form action="KomponentTestowy.jsp">
 Nazwa użytkownika: <input type="text" name="nazwaUzytkownika">
 ID: <input type="text" name="idPrac">

 Pierwsze danie: <input type="text" name="jedzenie">
 Drugie danie: <input type="text" name="jedzenie">

 <input type="submit">
</form>
```

Parametry „nazwaUzytkownika” i „idPrac” mają po jednej wartości. Okazuje się jednak, że parametr „jedzenie” może mieć dwie wartości (jeśli przed kliknięciem przycisku *Prześlij kwerendę* użytkownik wypełni oba pola formularza).

## W JSP

Parametr żądania nazwaUzytkownika: `${param.nazwaUzytkownika}` <br>

Parametr żądania idPrac: `${param.idPrac}` <br>

Parametr żądania jedzenie: `${param.jedzenie}` <br>

Pierwszy parametr żądania jedzenie: `${paramValues.jedzenie[0]}` <br>

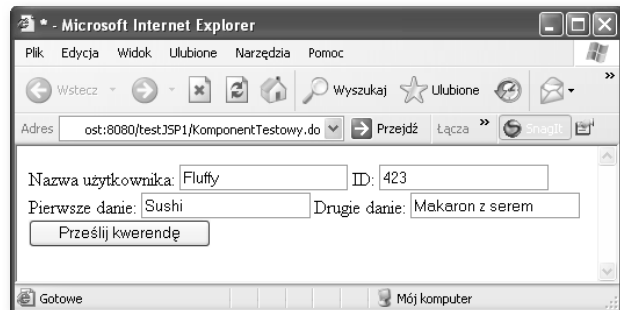
Drugi parametr żądania jedzenie: `${paramValues.jedzenie[1]}` <br>

Parametr żądania nazwaUzytkownika: `${paramValues.nazwaUzytkownika[0]}`

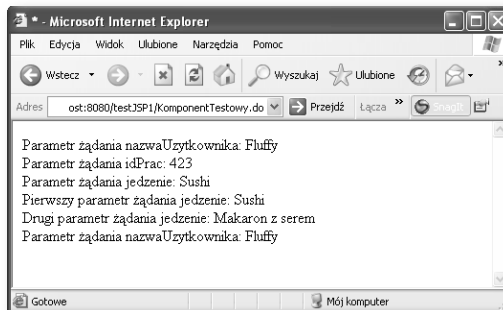
Pamiętaj, że obiekt domyślny param jest po prostu mapą (obiektem implementującym interfejs Map) nazw i wartości parametrów. Składniki na prawo od operatora kropki muszą być zgodne z nazwami zdefiniowanymi w polach wejściowych formularza HTML.

Chociaż parametr „jedzenie” może reprezentować więcej niż jedną wartość, nadal można używać obiektu domyślnego dla pojedynczego parametru, ale wówczas będzie odczytywana tylko pierwsza jego wartość.

## W przeglądarce klienta (klient wypełnia formularz i klika przycisk *Prześlij kwerendę*)



## Odpowiedź



## Co się stanie, jeśli będziemy potrzebowali więcej informacji z danego żądania?

Co powinienem zrobić, jeśli będę chciał uzyskać dostęp np. do informacji serwera zawartych w nagłówku "host" żądania HTTP? Jeśli przyjrzymy się API interfejsu `HttpServletRequest`, z pewnością zwrócisz uwagę na metodę `getHeader(String)`. Wiemy, że jeśli prześlemy łańcuch "host" na wejściu metody `getHeader()`, otrzymamy coś w stylu: "localhost:8080" (ponieważ właśnie tam działa nasz serwer WWW).

### Odczytywanie nagłówka „host”

Wiemy, jak można to zrobić za pomocą *elementów skryptowych*

```
Serwer: <%= request.getHeader("host") %>
```

Ale w języku wyrażeń EL mamy do dyspozycji obiekt domyślny header

```
Serwer: ${header["host"]}
```

```
Serwer: ${header.host}
```

Obiekt domyślny header reprezentuje mapę wszystkich nagłówków żądania. Do przekazania nazwy nagłówka i otrzymania jego wartości należy użyć jednego z dwóch operatorów dostępu. (Uwaga: istnieje także obiekt domyślny `headerValues` dla nagłówków wielowartościowych. Obiektu `headerValues` można używać w taki sam sposób jak omówionego wcześniej obiektu `paramValues`).

### Odczytywanie metody żądania HTTP

Hm. To będzie trochę trudniejsze... co prawda w API interfejsu `HttpServletRequest` istnieje metoda `getMethod()`, która zwraca GET, POST itp., ale jak uzyskać podobny efekt, stosując wyłącznie wyrażenia języka EL?

Wiemy, jak można to zrobić za pomocą *elementów skryptowych*

```
Metoda: <%= request.getMethod() %>
```

ale w języku EL *to NIE zadziała*

```
Metoda: ${request.method}
```

**NIE! NIE! NIE! NIE** istnieje obiekt domyślny request!

*To także NIE zadziała*

```
Metoda: ${requestScope.method}
```

**NIE! NIE! NIE!** Co prawda **ISTNIEJE** obiekt domyślny requestScope, ale **NIE** reprezentuje on samego żądania HTTP.

### Potrafisz znaleźć sposób realizacji tego zadania?

Wskazówka: przyjrzyj się raz jeszcze pozostałym obiektom domyślnym.

## Obiekt requestScope NIE jest obiektem żądania

Obiekt domyślny requestScope jest tylko mapą atrybutów zasięgu żądania, a nie obiektem samego żądania! To, co nas w tym momencie interesuje (metoda żądania HTTP), ma postać *właściwości* obiektu żądania, nie atrybutu zasięgu żądania. Innymi słowy, chcemy uzyskać dostęp do czegoś, co jest zwracane przez odpowiednią metodę obiektu żądania (jeśli przyjmiemy, że obiekt ten przypomina komponent).

Ale nie istnieje przecież obiekt domyślny request, a jedynie obiekt requestScope! Co robić?

Musimy szukać innego rozwiązania...

**Obiektu requestScope używaj do uzyskiwania  
ATRYBUTÓW żądania, a nie jego WŁAŚCIWOŚCI.  
W przypadku właściwości żądania należy korzystać  
z pośrednictwa obiektu domyślnego pageContext.**

**Używaj obiektu pageContext do odczytywania wszystkich  
pozostałych informacji...**

Metoda: `${pageContext.request.method}`

**Obiekt pageContext zawiera właściwość request.**

**Właściwość request zawiera właściwość method.**



**Oglądaj to!**

**Nie myl obiektów map poszczególnych zasięgów z obiektami,  
z którymi są wiązane atrybuty.**

To bardzo proste — wystarczy, że będziesz traktował np. obiekt applicationScope jak referencję do obiektu typu ServletContext, ponieważ właśnie tam znajdują się dowiązania atrybutów zasięgu aplikacji. Jednak podobnie jak w przypadku obiektów requestScope i request mapa zasięgu dla atrybutów zasięgu aplikacji jest tylko mapą atrybutów, niczym więcej. Nie możesz jej traktować jak kontekstu serwletu, a więc nie możesz oczekiwać, że obiekt domyślny applicationScope będzie Ci udostępniał właściwości należące do kontekstu serwletu (składowane w obiekcie ServletContext).

Skoro język EL i tak przeszukuje wszystkie zasięgi, po co w ogóle mam używać jednego z obiektów domyślnych reprezentujących zasięg? Jedyne, czego mógłbym się obawiać, to konflikt nazw, ale ciekawe, czy są jeszcze jakieś inne powody...



## Obiekty domyślne zasięgów mogą być wybawieniem

Jeśli musisz tylko wyświetlić imię danej osoby i nie interesuje Cię, w którym zasięgu jest składowany odpowiedni obiekt klasy Osoba (lub masz świadomość, że we wszystkich czterech zasięgach istnieje tylko jeden taki obiekt), możesz użyć następującego wyrażenia:

```
${osoba.imie}
```

Jeśli jednak obawiasz się potencjalnego konfliktu nazw, możesz wprost określić, który obiekt (z którego zasięgu) jest przedmiotem Twojego zainteresowania:

```
${requestScope.osoba.imie}
```

Ale czy istnieje jakiś inny powód, dla którego mielibyśmy poprzedzać atrybut nazwą domyślnego obiektu zasięgu? Inny niż kontrola... zasięgów?

Przemyśl następujący scenariusz: jeśli w nawiasach kwadratowych użyjesz nazwy bez cudzysłowów, będzie to oznaczało, że nazwa ta MUSI być zgodna z regułami nazewnictwa Javy, prawda? W tym przypadku wszystko jest w porządku, ponieważ osoba jest całkowicie poprawną nazwą zmiennej Javy. Warto jednak pamiętać, że ta poprawność wynika z faktu, że ktoś już kiedyś wywołał metodę:

```
request.setAttribute("osoba", o);
```

### Ale przecież nazwa atrybutu jest łańcuchem!

A same łańcuchy nie muszą być zgodne z regułami nazewnictwa obowiązującymi w Javie!

To zaś oznacza, że ktoś *mógł* równie dobrze wywołać metodę:

```
request.setAttribute("foo.osoba", o);
```

Wówczas będziemy mieli poważny problem, ponieważ TO nie zadziała:

```
${foo.osoba.imie}
```

Powinieneś teraz docenić rolę obiektów zasięgu, ponieważ właśnie dzięki tym obiektom możesz przejść na operator `[]` umożliwiający stosowanie nazw łańcuchowych, które z kolei nie muszą spełniać reguł nazewnictwa Javy.

```
${requestScope["foo.osoba"].imie}
```

NIE! Takie wyrażenie z pewnością jest poprawne, ale kontener zinterpretuje je w nieco inny sposób: pomyśli, że „foo” jest atrybutem jednego z zasięgów, a „osoba” jest właściwością tego atrybutu. Oczywiście kontener nigdy nie znajdzie atrybutu „foo”.

Doskonale! Zastosowanie obiektu requestScope umożliwiło nam umieszczenie nazwy atrybutu w cudzysłowach.

# Odczytywanie ciasteczek klienta i parametrów inicjalizacji

Przeanalizowaliśmy już wszystkie obiekty domyślne EL z wyjątkiem znaczników kontekstu klienta (tzw. ciasteczek) i parametrów inicjalizacji — oba obiekty omówimy poniżej. A tak! Zapomnieliśmy wspomnieć, że wszystkie obiekty niejawne mogą się pojawić na egzaminie.

## Wyświetlanie wartości ciasteczka „nazwaUzytkownika”

Wiemy, jak można to zrobić za pomocą *elementów skryptowych*

```
<% Cookie[] cookies = request.getCookies();

for (int i = 0; i < cookies.length; i++) {
 if ((cookies[i].getName()).equals("nazwaUzytkownika")) {
 out.println(cookies[i].getValue());
 }
} %>
```

*Przetwarzanie ciasteczek za pomocą elementów skryptowych jest kłopotliwe, ponieważ obiekt żądania NIE zawiera metody `getCookie(cookieName)`! Musimy odczytać całą tablicę ciasteczek i sami odnaleźć w niej interesujący nas obiekt.*

Ale w języku EL mamy do dyspozycji obiekt domyślny cookie

```
${cookie.nazwaUzytkownika.value}
```

*DUŻO prostsze rozwiązanie. Wystarczy przekazać samą nazwę znacznika kontekstu klienta, a otrzymamy wartość z obiektu mapy ciasteczek (przechowującego pary nazwa-wartość).*

## Wyświetlanie parametru inicjalizacji kontekstu

Musimy skonfigurować odpowiedni parametr w deskryptorze wdrożenia

```
<context-param>
 <param-name>emailGlowny</param-name>
 <param-value>cokolwiek@wickedlysmart.com</param-value>
</context-param>
```

*Pamiętaj, że w ten sposób można konfigurować parametry inicjalizacji kontekstu (o zasięgu ogólnoplikacyjnym). To NIE to samo co parametry inicjalizacji serwletu.*

Wiemy, jak można to zrobić za pomocą elementów skryptowych

```
email: <%= application.getInitParameter("emailGlowny") %>
```

Okazuje się, że w języku EL wyświetlenie parametru jest jeszcze prostsze

```
email: ${initParam.emailGlowny}
```



Oglądaj to!

**Stosowany w języku EL obiekt domyślny `initParam` NIE służy do odczytywania parametrów skonfigurowanych za pomocą znacznika `<init-param>`!**

*Podobne nazwy mogą być źródłem nieporozumień: parametry inicjalizacji serwletu są konfigurowane za pomocą znaczników `<init-param>`, natomiast parametry inicjalizacji **kontekstu** definiuje się w znacznikach `<context-param>`; obiekt domyślny `"initParam"` języka EL udostępnia parametry kontekstu! Gdyby nas poproszono o radę, zasugerowalibyśmy autorom specyfikacji rozważenie nadania tej zmiennej nazwy `"contextParam"`... ale znowu zapomnieli nas spytać o zdanie.*

EL jest cudowny... ale czasem potrzebuję funkcjonalności, a nie tylko wartości atrybutów i właściwości. Gdyby tylko istniał sposób zmuszenia wyrażenia EL do wywołania metody Javy, która zwróciłaby jakąś wartość... byłabym szczęśliwa.



## Ona nie wie jeszcze o funkcjach EL

Kiedy potrzebujesz trochę silniejszego wsparcia ze strony, powiedzmy, metody Javy, ale nie chcesz stosować w kodzie swojej strony JSP elementów skryptowych, możesz użyć funkcji języka EL. Jest to najprostszy sposób napisania prostego wyrażenia EL, które wywoła statyczną metodę przygotowanej wcześniej tradycyjnej klasy Javy. Wszystko, co zostanie przez tę metodę zwrócone będzie wykorzystane w danym wyrażeniu. Konfiguracja takiego rozwiązania wymaga co prawda trochę więcej pracy, ale funkcje zapewniają znacznie więcej... *funkcjonalności*.

## Wyobraź sobie, że chcesz, aby Twoja strona JSP rzucała kostką

Zdecydowałeś, że byłoby cudownie, gdyby udało się stworzyć internetową usługę rzucającą kostkami do gry. Taka usługa pozwoliłaby uniknąć użytkownikom kłopotliwego szukania *prawdziwych* kostek za biurkiem lub w załamaniach kanapy — wystarczyłoby wejść na Twoją stronę internetową, kliknąć wirtualną kostkę i poczekać na wynik! Rzucanie kostkami do gry nigdy nie było równie proste! (Oczywiście nie masz pojęcia, że wyszukiwanie podobnych stron w *Google* prawdopodobnie zwróci ze 4 tysiące witryn, które robią dokładnie to samo).

### ① Napisz klasę Javy z publiczną metodą statyczną.

Jest to po prostu zwykła, tradycyjna klasa Javy. Utworzona w tej klasie metoda MUSI być publiczna i statyczna, może też otrzymywać na wejściu argumenty. Metoda powinna (choć nie jest to wymagane) mieć zdefiniowany typ zwracanych wartości inny niż `void`. Naszym celem jest przecież wywołanie jej z poziomu strony JSP, aby otrzymać z powrotem coś, co będziemy mogli wykorzystać jako część wyrażenia lub przynajmniej wyświetlić.

Umieść plik klasy w strukturze katalogów `/WEB-INF/classes` (oczywiście po odpowiednim dostosowaniu struktury katalogów pakietów Javy, tak jak w przypadku wszystkich innych klas aplikacji internetowej).

### ② Przygotuj plik deskryptora biblioteki znaczników (TLD).

W przypadku funkcji języka wyrażeń (EL) deskryptor *TLD* (od ang. *Tag Library Descriptor*) zapewnia niezbędne odwzorowanie pomiędzy klasą Javy, która *definiuje* tę funkcję, a stroną JSP, która tę funkcję *wywołuje*. Dzięki temu nazwa funkcji i rzeczywista nazwa metody nie muszą być takie same. Przykładowo, korzystanie z klasy udostępniającej metodę z najgłupszą nazwą, jaką można sobie wyobrazić, mogłoby być kłopotliwe; odwzorowanie deskryptora TLD umożliwia stworzenie bardziej oczywistej i (lub) intuicyjnej nazwy dla korzystających z języka EL projektantów stron internetowych. To żaden problem — wystarczy, że deskryptor TLD powie: „To jest klasa Javy, to jest sygnatura metody dla danej funkcji (włącznie ze zwracanym typem), a to jest *nazwa*, która będzie stosowana w wyrażeniach języka EL”. Innymi słowy, *nazwa* wykorzystywana w wyrażeniach EL nie musi być taka sama jak rzeczywista nazwa metody, a za odpowiednie odwzorowanie nazw odpowiada deskryptor biblioteki znaczników (TLD).

Umieść plik TLD w katalogu `/WEB-INF`. W nazwie pliku deskryptora koniecznie użyj rozszerzenia `.tld`. (Istnieją także inne miejsca, w których można umieszczać pliki deskryptorów TLD; będziemy o tym mówili w dwóch kolejnych rozdziałach).

### ③ Umieść odpowiednią dyrektywę taglib w swoim kodzie JSP.

Dyrektywa `taglib` wysyła do kontenera sygnał: „Mam zamiar użyć tego deskryptora TLD, a kiedy w kodzie JSP będzie potrzebna funkcja z tego deskryptora, mam zamiar ją poprzedzić tą nazwą...”. Innymi słowy, dyrektywa `taglib` umożliwia nam definiowanie przestrzeni nazw. Możemy używać funkcji zadeklarowanych w więcej niż jednym pliku deskryptora biblioteki znaczników; co więcej, nawet istnienie wielu tak samo nazwanych funkcji nie stanowi problemu. Dyrektywę taglib można traktować jak narzędzie do nadawania naszym funkcjom pełnych nazw. Tego typu funkcję wywołujemy przez określenie zarówno jej nazwy, JAK I przedrostka TLD. Przedrostek zdefiniowany w deskrytorze może mieć dowolną postać, na jaką się zdecydujemy.

### ④ Użyj języka EL do wywołania funkcji.

To jest w tym wszystkim najprostsze. Funkcje możemy wywoływać z wyrażenia, stosując zapis `${przedrostek.nazwa()}`.

## Klasa funkcji, plik deskryptora TLD oraz strona JSP

Metoda funkcji *MUSI* być publiczna i statyczna.

### Klasa z daną funkcją

```
package foo;

public class RzucanieKostka {
 public static int rzutKostka() {
 return (int) ((Math.random() * 6) + 1);
 }
}
```

### Plik deskryptora biblioteki znaczników (TLD)

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<taglib xmlns="http://java.sun.com/xml/ns/j2ee" xmlns:xsi="http://www.
w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/
xml/ns/j2ee/web-jsptaglibrary_2_0.xsd" version="2.0">

 <tlib-version>1.2</tlib-version>
 <uri>FunkcjeKostki</uri>
 <function>
 <name>rzut</name>
 <function-class>foo.RzucanieKostka</function-class>
 <function-signature>
 int rzutKostka()
 </function-signature>
 </function>

</taglib>
```

NIE martw się o całą tę zawartość znacznika <taglib>.

Deskryptory TLD omówimy dokładniej w dwóch kolejnych rozdziałach.

Atrybut uri w dyrektywie taglib określa na potrzeby kontenera nazwę deskryptora TLD (która wcale NIE musi być taka sama jak nazwa PLIKU deskryptora!), na podstawie której kontener określa metodę wykonywaną w momencie wywołania funkcji EL w kodzie strony JSP.

### Kod strony JSP

```
<%@ taglib prefix="moje" uri="FunkcjeKostki" %>

<html><body>

 ${moje:rzut()}

</body></html>
```

Przedrostek "moje" jest po prostu przydomkiem, którego będziemy używać w kodzie TEJ strony — stosowanie tego typu przydomków umożliwia odróżnianie deskryptorów (w sytuacji, gdy MAMY ich kilka).

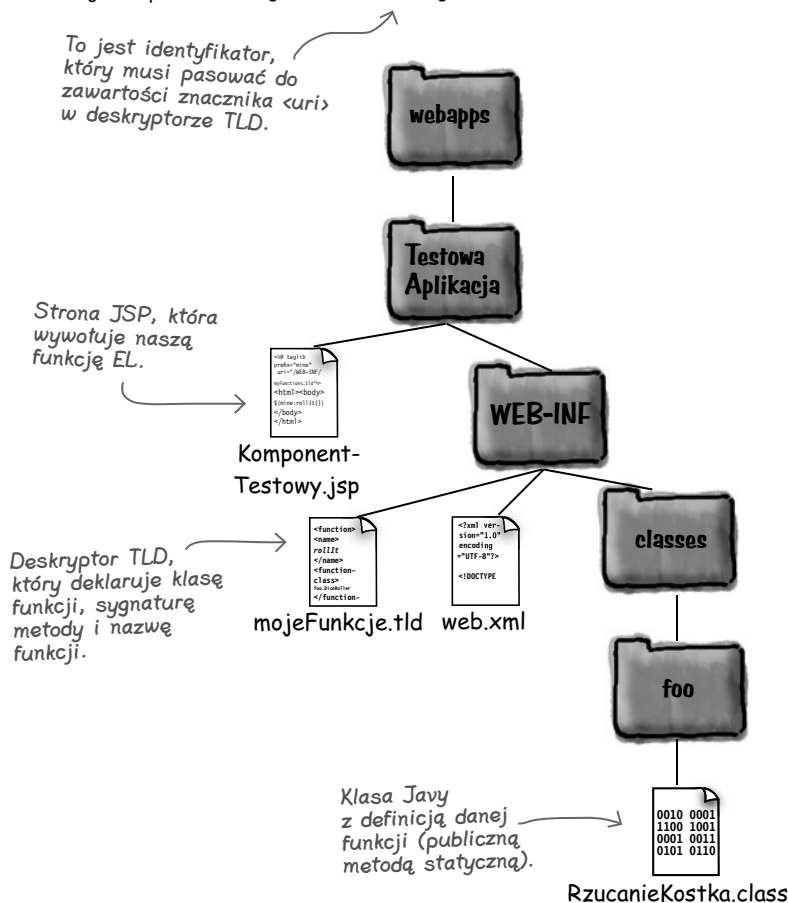
Nazwa funkcji rzut() pochodzi ze znacznika <name> w pliku deskryptora TLD, a nie z którejkolwiek z istniejących klas Javy.

# Wdrażanie aplikacji internetowej z funkcjami statycznymi

Jedyną nowością jest w tym przypadku plik *mojeFunkcje.tld*. Plik ten musi się znajdować gdzieś w katalogu *WEB-INF* lub jednym z jego podkatalogów (chyba że wdrażana aplikacja ma postać pliku JAR, ale taki przypadek szczegółowo omówimy w dalszej części tej książki). Ponieważ wdrażana przez nas aplikacja jest bardzo prosta, deskryptor wdrożenia (*web.xml*) i deskryptor biblioteki znaczników (*mojeFunkcje.tld*) będą przechowywane na najwyższym poziomie katalogu *WEB-INF*, ale równie dobrze moglibyśmy je umieścić w hierarchii podkatalogów.

Kluczowe znaczenie ma w tym przypadku takie rozmieszczenie plików, aby nasza klasa z funkcją statyczną BYŁA dostępna dla pozostałych składników aplikacji, zatem... na razie wiemy, że jej umieszczenie w katalogu *WEB-INF/classes* przyniesie pożądany efekt. Nie zapominaj, że w dyrektywie taglib w kodzie strony JSP określiliśmy identyfikator URI pasujący do URI zadeklarowanego w deskrytorze TLD. Na tym etapie możesz traktować tajemniczy skrót URI jak **dowolne wyrażenie, które posłużyło nam za nazwę w pliku TLD**. To tylko nazwa. W następnym rozdziale, poświęconym stosowaniu znaczników niestandardowych, omówimy wszystkie szczegóły związane z deskryptorami TLD i identyfikatorami URI.

```
<%@ taglib prefix="moje" uri="FunkcjeKostki" %>
```



Klasa udostępniająca daną funkcję (publiczną metodę statyczną) musi być dostępna dla aplikacji internetowej tak jak serwet, komponent czy klasy nasłuchujące. Oznacza to, że plik takiej klasy musi znajdować się gdzieś w katalogu *WEB-INF/classes....*

Plik TLD należy umieścić gdzieś pod katalogiem *WEB-INF*, więc musisz się upewnić, że dyrektywa taglib w kodzie JSP zawiera atrybut `uri` pasujący do elementu `<uri>` w wykorzystywanym deskrytorze TLD.

## Nie ma niemądrych pytań

**P:** Zwykłe wyrażenie skryptletu **MUSI** coś zwracać. Jeśli powiemy `<%= foo.getFoo() %>`, metoda `getFoo()` **NIE** może zwracać typu `void`. (Przynajmniej zgodnie z tym, co mówiłeś wcześniej). Tak sobie myślę, że to samo dotyczy funkcji języka EL, mam rację?

**U:** Nie! Z funkcjami EL jest nieco INACZEJ, choć w praktyce dla wielu programistów ta różnica jest... zaskakująca. Przemyśl to — jeśli wywołujesz funkcję języka EL, która niczego nie zwraca, tak naprawdę wywołujesz ją *wyłącznie z myślą o jakimś efekcie ubocznym jej wykonywania!* Skoro wiemy, że jednym z celów opracowania języka EL było ograniczenie ilości kodu w stronach JSP (strony JSP mają przecież stanowić WIDOK aplikacji!), możemy przyjąć, że wywoływanie funkcji EL tylko dla jej efektu ubocznego nie jest dobrym rozwiązaniem.

**P:** Jak kontener odnajduje plik TLD? Identyfikator URI nie określa ścieżki ani nazwy pliku deskryptora. Czy mamy do czynienia z cudem?

**U:** Mieliśmy nadzieję, że ktoś wreszcie zada to pytanie. Tak, masz rację — nigdy *nie mówiliśmy* kontenerowi, gdzie dokładnie należy szukać rzeczywistego pliku TLD. Kiedy aplikacja jest wdrażana, kontener automatycznie przegląda katalog `WEB-INF` i wszystkie jego podkatalogi (lub pliki JAR w katalogu `WEB-INF/lib`) w poszukiwaniu plików `.tld`. Kiedy taki plik zostanie znaleziony, kontener odczytuje z niego identyfikator URI i tworzy odwzorowanie, które mówi: „Deskryptor TLD z *tym* identyfikatorem URI w rzeczywistości znajduje się w *tym* pliku w *tym* katalogu...”. Ta historia ma swój ciąg dalszy, który omówimy w następnym rozdziale.

**P:** Czy funkcja EL może otrzymywać argumenty?

**U:** Oczywiście. Musisz tylko pamiętać, aby w pliku TLD określić pełną nazwę klasy (chyba że są to typy proste) dla każdego z tych argumentów. Przykładowo, dla funkcji otrzymującej na wejściu argument typu `Map` taka deklaracja miałaby postać:

```
<function-signature>
 int rzutKostka(java.util.Map)
</function-signature>
```

A jej wywołanie wymaga użycia wyrażenia `${moje:rzutKostka(jakiśAtrybutMapy)}`.



Oglądaj to!

**Nazwa METODY to nie to samo co nazwa FUNKCJI!**

Musisz dobrze zapamiętać relacje występujące pomiędzy klasą, deskryptorem TLD oraz stroną JSP. Najważniejsze, abyś nie zapominał, że nazwa METODY NIE musi pasować do nazwy FUNKCJI. Identyfikatory używane przez Ciebie w wyrażeniu EL do wywoływania funkcji muszą pasować jedynie do elementów `<name>` w deklaracjach `<function>` z deskryptora biblioteki znaczników (TLD). Wykorzystywany tam element `<function-signature>` ma na celu jedynie określenie na potrzeby kontenera, która metoda ma być wywoływana, kiedy JSP użyje danej nazwy (`<name>`). I jedynym miejscem, w którym występuje nazwa klasy (oczywiście poza samą deklaracją klasy), jest element `<function-class>`. Ach, skoro już tutaj jesteśmy... czy zauważyłeś, że nazwy wszystkich elementów w ramach znacznika `<function>` POZA znacznikiem `<name>` zawierają słowo `<function>`? Musisz w związku z tym zachować ostrożność, aby nie popełnić takiego błędu:

```
<function>
 <function-name>rzut</function-name>
 <function-class>
 foo.RzucanieKostka</function-class>
 <function-signature>
 int rzutKostka()
 </function-signature>
</function>
```

Prawidłowym znacznikiem definiującym nazwę funkcji jest `<name>`!

```
<function>
 <name>rzut</name>
 <function-class>
 foo.RzucanieKostka</function-class>
 <function-signature>
 int rzutKostka()
 </function-signature>
</function>
```

# I jeszcze kilka operatorów EL...

Prawdopodobnie nie będziesz (a przynajmniej *nie powinienes*) wykonywać obliczeń i działań związanych z logiką biznesową na poziomie wyrażen języka EL. Pamiętaj, że strona JSP jest widokiem, a zadaniem widoku jest jedynie wizualizacja odpowiedzi, a nie podejmowanie ważnych decyzji ani wielkie przetwarzanie. Jeśli potrzebujesz rzeczywistej funkcjonalności, w normalnych warunkach za jej zapewnienie odpowiada kontroler i model. Przygotowanie pomniejszej funkcjonalności jest możliwe w oparciu o znaczniki niestandardowe (włącznie ze znacznikami biblioteki JSTL) oraz funkcje EL.

Ale... w przypadku drobnych operacji mających niekiedy postać prostych działań arytmetycznych lub testów logicznych stosowanie tego typu elementów może być bardzo wygodne. W tej sytuacji warto się przyjrzeć najbardziej przydatnym operatorom arytmetycznym, relacyjnym i logicznym języka EL.

## Arytmetyczne (5)

Dodawanie:	+	
Odejmowanie:	-	
Mnożenie:	*	
Dzielenie:	/ oraz <b>div</b>	Nawiasem mówiąc... w języku EL MOŻEMY dzielić przez zero — zamiast błędu otrzymamy wówczas NIESKONCZONOŚĆ.
Reszta z dzielenia:	% oraz <b>mod</b>	NIE możesz jednak używać operatora reszty z dzielenia przez zero — w takim przypadku zostanie zwrócony wyjątek.

## Logiczne (3)

I:	<b>&amp;&amp;</b> oraz <b>and</b>
LUB:	<b>  </b> oraz <b>or</b>
NIE:	<b>!</b> oraz <b>not</b>

## Relacyjne (6)

Równe:	<b>==</b> oraz <b>eq</b>
Różne:	<b>!=</b> oraz <b>ne</b>
Mniejsze:	<b>&lt;</b> oraz <b>lt</b>
Większe:	<b>&gt;</b> oraz <b>gt</b>
Mniejsze lub równe:	<b>&lt;=</b> oraz <b>le</b>
Większe lub równe:	<b>&gt;=</b> oraz <b>ge</b>



Oglądaj to!

**Nigdy nie używaj słów zastrzeżonych jako identyfikatorów!**

Z jedenastoma takimi słowami miałeś okazję się zapoznać na tej stronie — są to „słowa” alternatywne dla operatorów relacyjnych logicznych i arytmetycznych. Jednak język EL zawiera jeszcze kilka słów zastrzeżonych:

<b>true</b>	stała logiczna
<b>false</b>	INNA stała logiczna
<b>null</b>	oznacza... null
<b>instanceof</b>	(słowo zastrzeżone „na przyszłość”)
<b>empty</b>	operator sprawdzania, czy coś jest równe null, czy jest puste; przykładowo, wyrażenie <code>\${empty A}</code> zwraca wartość true, jeśli A jest równe null lub jest puste (praktyczne zastosowanie tego słowa zademonstrujemy w dalszej części tego rozdziału).



## Zaostrz ołówek

Spójrz na poniższy fragment kodu, po czym spróbuj określić, co zostanie wyświetlone na stronie dla każdego z przedstawionych wyrażeń języka EL. W kilku przypadkach będziesz musiał się oprzeć na swojej intuicji, ponieważ nie omówiliśmy jeszcze wszystkich możliwych reguł. Sumienne wykonanie tego ćwiczenia ułatwi Ci w przyszłości przewidywanie zachowań wyrażeń języka EL. Wskazówka: język EL jest elastyczny i wybacza błędy.

### Mając dany następujący kod serwletu:

```
String num = "2";
request.setAttribute("num", num);
Integer i = new Integer(3);
request.setAttribute("integer", i);
java.util.ArrayList list = new java.util.ArrayList();
list.add("true");
list.add("false");
list.add("2");
list.add("10");
request.setAttribute("list", list);
```

*Przyjmij, że masz dostęp do klasy komponentu Pies.*

### spróbuj określić, co zostanie wyświetlone przez każde z tych wyrażeń:

\_\_\_\_\_ `${num > 3}`

\_\_\_\_\_ `${integer le 12}`

\_\_\_\_\_ `${requestScope(integer) ne 4 and 6 le num || false}`

\_\_\_\_\_ `${list[0] || list["1"] and true}`

\_\_\_\_\_ `${num > integer}`

\_\_\_\_\_ `${num == integer-1}`

```
<jsp:useBean clean="foo.Pies" id="mojPies" >
 <jsp:setProperty name="mojPies" property="imie" value="${list[1]}" />
</jsp:useBean>
```

\_\_\_\_\_ `${mojPies.imie and true}`

\_\_\_\_\_ `${42 div 0}`



### Mając dany następujący kod serwletu:

```
String num = "2";
request.setAttribute("num", num);
Integer i = new Integer(3);
request.setAttribute("integer", i);
java.util.ArrayList list = new java.util.ArrayList();
list.add("true");
list.add("false");
list.add("2");
list.add("10");
request.setAttribute("list", list);
```

### spróbuj określić, co zostanie wyświetlone przez każde z tych wyrażeń:

<u>false</u>	<code>\${num &gt; 3}</code>	Atrybut "num" został znaleziony, a jego wartość "2" jest automatycznie konwertowana na typ int.
<u>true</u>	<code>\${integer le 12}</code>	To jeszcze ciekawsze! Wartość typu Integer została automatycznie przekonwertowana na postać odpowiedniego typu prostego (int), dopiero potem wartości po obu stronach operatora zostały ze sobą porównane.
<u>false</u>	<code>\${requestScope(integer) ne 4 and 6 le num    false}</code>	Zobacz, czy jesteś w stanie określić reguły kolejności działań w sytuacji, gdy wyrażenie nie zawiera nawiasów. Reguły te są bardzo intuicyjne (od lewej do prawej), zatem podczas właściwego egzaminu NIE powinieneś mieć żadnych problemów z wyznaczaniem pierwszeństwa działań.
<u>true</u>	<code>\${list[0]    list["1"] and true}</code>	
<u>false</u>	<code>\${num &gt; integer}</code>	
<u>true</u>	<code>\${num == integer-1}</code>	

Zwróć uwagę na użycie operatora `==` zamiast operatora `=`. W języku wyrażeń (EL) NIE ma operatora `=`.

```
<jsp:useBean clean="foo.Pies" id="mojPies" >
 <jsp:setProperty name="mojPies" property="imie" value="${list[1]}" />
</jsp:useBean>
```

false `${mojPies.imie and true}`

infinity `${42 div 0}`

Tak, można używać wyrażeń języka EL wewnątrz znaczników!

# Język EL obsługuje wartości null z wdziękiem

Kluczową decyzją projektową, przed którą stają programiści wyrazów języka EL, jest zapewnienie właściwej obsługi wartości null bez zwracania wyjątków. Dlaczego? Dlatego, że jedno z założeń tego języka mówi: „lepiej wyświetlić częściową, niekompletną stronę, niż zaprezentować użytkownikowi stronę informującą o błędzie”.

Przyjmij, że *nie* istnieje atrybut nazwany "foo", ale ISTNIEJE atrybut nazwany "bar", który jednak nie zawiera właściwości ani klucza nazwanego "foo".

## Wyrażenie EL Co wyświetli

<hr/>		
<code>\${foo}</code>		
<code>\${foo[bar]}</code>		<p>← W tym przypadku nic nie zostanie wyświetlone. Jeśli powiemy w wyrażeniu: „wartość wynosi: <code>\${foo}</code>”, to tak, jakbyśmy powiedzieli: „wartość wynosi:”.</p>
<code>\${bar[foo]}</code>		
<code>\${foo.bar}</code>		
<hr/>		
<code>\${7 + foo}</code>	7	<p>W wyrażeniach arytmetycznych języka EL nieznana zmienna jest traktowana jak „zero”.</p>
<code>\${7 / foo}</code>	infinity	
<code>\${7 - foo}</code>	7	
<code>\${7 % foo}</code>	Zostanie rzucony wyjątek	
<hr/>		
<code>\${7 &lt; foo}</code>	false	<p>W wyrażeniach logicznych języka EL nieznana zmienna jest traktowana jak „fałsz”.</p>
<code>\${7 == foo}</code>	false	
<code>\${foo == foo}</code>	true	
<code>\${7 != foo}</code>	true	
<code>\${true and foo}</code>	false	
<code>\${true or foo}</code>	true	
<code>\${not foo}</code>	true	

Język EL jest przyjazny dla wartości null. Język ten obsługuje zmienne nieznane i zmienne równe null w taki sposób, aby nie przekreślać szansy na wyświetlenie strony, nawet jeśli nie ma możliwości odnalezienia atrybutu, właściwości lub klucza oznaczonego nazwą użytą w wyrażeniu.

W arytmetyce język EL traktuje wartość null jak „zero”.

W wyrażeniach logicznych język EL traktuje wartość null jak „fałsz”.

## Przegląd języka wyrażeń JSP (EL)

KLUCZOWE  
ZAGADNIENIA

- Wyrażenia języka EL zawsze są umieszczane wewnątrz nawiasów klamrowych i poprzedzane znakiem dolara (\$): `${wyrażenie}`.
- Pierwsza nazwana zmienna w danym wyrażeniu jest albo obiektem niejawnym (domyślnym), albo atrybutem należącym do jednego z czterech zasięgów (strony, żądania, sesji lub aplikacji).
- Operator kropki umożliwia nam uzyskiwanie dostępu do wartości za pomocą klucza mapy lub nazwy właściwości komponentu; na przykład wyrażenie `${foo.bar}` zwraca wartość `bar`, gdzie `bar` jest albo nazwą klucza mapy `foo`, albo właściwością komponentu `foo`. Cokolwiek umieścimy na prawo od operatora kropki, musi to być nazwa spełniająca normalne reguły nazewnictwa identyfikatorów Javy! (Innymi słowy, musi się rozpoczynać od litery, znaku podkreślenia lub znaku dolara, natomiast po pierwszym znaku może zawierać także cyfry, żadne inne znaki nie są dopuszczalne).
- Na prawo od operatora kropki NIGDY nie można umieszczać nazw, które nie są prawidłowymi identyfikatorami Javy. Przykładowo w kodzie swojej strony JSP nie możesz używać wyrażeń w postaci: `${foo.1}`.
- Operator `[]` zapewnia znacznie większe możliwości od operatora kropki, ponieważ pozwala na dostęp do tablic i list (egzemplarzy klasy `List`), a także na umieszczanie w nawiasach kwadratowych innych wyrażeń (włącznie z nazwanymi zmiennymi) oraz na zagnieżdżanie ich na dowolnym poziomie.
- Przykładowo, jeśli obiekt `listaUtworow` jest obiektem klasy `ArrayList`, możemy uzyskać dostęp do pierwszej wartości listy za pomocą wyrażenia `${listaUtworow[0]}` LUB `${listaUtworow["0"]}`. W języku EL nie ma znaczenia, czy indeks listy znajduje się w cudzysłowie, czy nie.
- Jeśli to, co umieściliśmy w nawiasach kwadratowych, nie jest otoczone cudzysłowem, kontener musi wyznaczyć wartość tego wyrażenia. Jeśli natomiast użyjesz cudzysłowu i nie będzie to prawidłowy indeks tabeli lub listy, kontener będzie wiedział, że ma do czynienia ze stałą nazwą właściwości lub klucza.
- Wszystkie obiekty domyślne języka EL poza jednym są mapami. Za pośrednictwem map obiektów domyślnych możemy odczytywać atrybuty należące do wszystkich

czterech zasięgów, wartości parametrów żądania, wartości nagłówków, wartości znaczników kontekstu klienta (ciasteczek) oraz parametry inicjalizacji kontekstu. Jedynym obiektem domyślnym niebędącym mapą jest `pageContext`, który jest referencją do... obiektu `PageContext`.

- Nie myl obiektów domyślnych języka wyrażeń (EL) reprezentujących zasięgi (mapy atrybutów należących do tych zasięgów) z obiektami, z którymi te atrybuty są związane. Innymi słowy, nie należy mylić obiektu domyślnego ***requestScope*** z dostępnym w JSP obiektem domyślnym ***request***. Jedynym sposobem uzyskania dostępu do obiektu `request` jest wykorzystanie w roli pośrednika obiektu domyślnego `pageContext`. (Chociaż niektóre dane, których w normalnych warunkach szukamy w obiekcie reprezentującym żądanie, są udostępniane także przez inne obiekty domyślne EL, włącznie z obiektami *param* i *paramValues*, *header* i *headerValues* oraz *cookie*).
- Funkcje języka EL umożliwiają nam wywoływanie publicznych metod statycznych zdefiniowanych w tradycyjnych klasach Javy. Nazwa funkcji nie musi odpowiadać nazwie rzeczywistej metody! Przykładowo, użycie w kodzie JSP wyrażenia `${foo.rzut()}` wcale nie oznacza, że musi istnieć metoda nazwana `rzut()` w klasie zawierającej definicję odpowiedniej funkcji.
- Nazwa funkcji (np. `rzut()`) jest odwzorowywana na rzeczywistą metodę statyczną w oparciu o odpowiednie zapisy w pliku deskryptora biblioteki znaczników (TLD). Musimy zadeklarować funkcję za pomocą elementu `<function>` obejmującego nazwę funkcji (`<name>`, w tym przypadku `rzut()`), pełną nazwę klasy funkcji (`<function-class>`) oraz sygnaturę funkcji (`<function-signature>`), która obejmuje nie tylko zwracany typ, ale także nazwę i listę argumentów metody.
- Aby użyć funkcji języka EL w kodzie strony JSP, musimy zadeklarować odpowiednią przestrzeń nazw za pomocą dyrektywy `taglib`. Umieszczamy atrybut `prefix` w dyrektywie `taglib`, aby wskazać kontenerowi deskryptor TLD, w którym znajduje się deklaracja wywoływanej przez nas funkcji. Oto przykład:

```
<%@ taglib prefix="moje"
uri="/WEB-INF/foo.tld"%>
```



OCZYWIŚCIE omówimy szablony układu. Jeśli KTOŚ zna komponenty wielokrotnego użytku, z pewnością musi to być programista Javy.



## Szablony wielokrotnego użytku

Na wszystkich stronach swojej witryny internetowej używasz nagłówków. Wszystkie te nagłówki są takie same. Na każdej stronie tej witryny stosujesz także tę samą stopkę. Czyż kodowanie tych samych znaczników nagłówka i stopki we wszystkich stronach JSP Twojej aplikacji internetowej nie byłoby głupotą?

Jeśli myślisz jak programista Javy (którym rzecz jasna jesteś), wiesz, że takie działanie byłoby zaprzeczeniem idei programowania obiektowego. Prawdopodobnie już na samą myśl o powielaniu kodu we wszystkich stronach JSP robi Ci się niedobrze. Co będzie, jeśli projektant Twojej witryny internetowej wprowadzi drobną *zmianę* w nagłówku lub stopce?

Będziesz musiał wprowadzić odpowiednie modyfikacje we wszystkich swoich stronach.

Spokojnie. Technologia JSP oferuje mechanizm obsługi tego typu sytuacji — nosi nazwę **include**. Swoje strony JSP piszesz tak jak do tej pory, z tą różnicą, że zamiast umieszczania wielokrotnie wykorzystywanych elementów wprost w kodzie JSP nakazujesz tylko kontenerowi *dołączenie* do istniejącej strony innego pliku (przechowywanego we wskazanym przez Ciebie katalogu). To tak, jakbyś stwierdzał:

```
<html><body>
```

```
<!-- tutaj wstaw plik nagłówka -->
```

Witamy na naszej witrynie...

bla, bla, bla, więcej kodu...

```
<!-- tutaj wstaw plik stopki -->
```

```
</body></html>
```

W tej części rozdziału przyjrzymy się dwóm różnym mechanizmom dołączania kodu: *dyrektywie include* oraz *standardowej akcji* `<jsp:include/>`.

## Dyrektywa include

Dyrektywa include mówi kontenerowi jedną rzecz: skopiuj całą zawartość dołączanego pliku i wklej ją w tym pliku, właśnie **tutaj**...

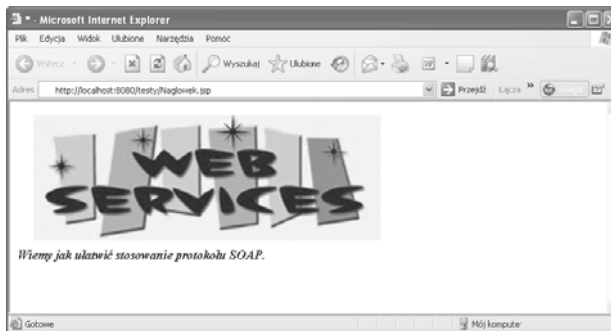
### Plik standardowego nagłówka („Naglowek.jsp”)

```
<html><body>

Wiemy jak ułatwić stosowanie protokołu SOAP.

</body></html>
```

To jest nasz kod HTML, który chcemy umieścić w każdej stronie danej aplikacji internetowej.



### Strona JSP z aplikacji internetowej („Kontakt.jsp”)

```
<html><body>

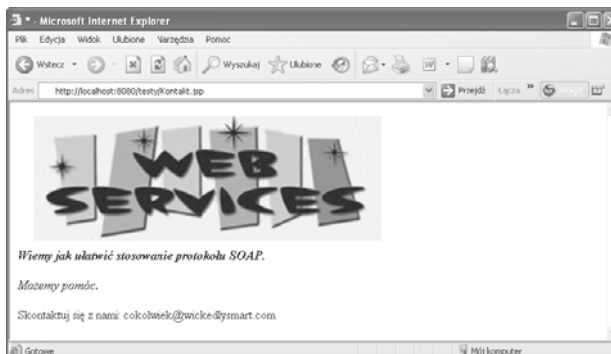
<%@ include file="Naglowek.jsp"%>

Możemy pomóc.

Skontaktuj się z nami: ${initParam.glownyEmail}

</body></html>
```

W tym miejscu mówimy: „Wstaw TUTAJ, NA TEJ stronie całą zawartość pliku Naglowek.jsp, po czym kontynuuj przetwarzanie reszty tej strony JSP...”.



# Standardowa akcja <jsp:include>

Wygłąda na to, że standardowa akcja <jsp:include> ma identyczne znaczenie jak dyrektywa include.

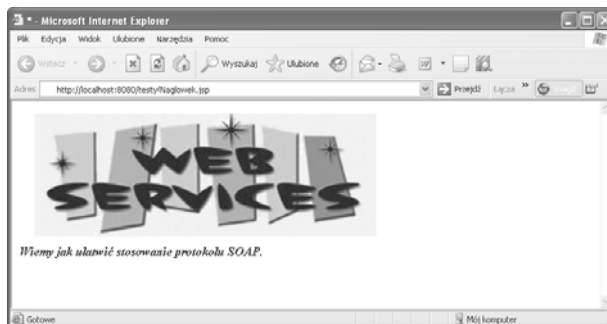
## Plik standardowego nagłówka („Naglowek.jsp”)

```
<html><body>

Wiemy jak ułatwić stosowanie protokołu SOAP.

</body></html>
```

Chcemy, aby ten element HTML był wyświetlany na **KAZDEJ** stronie naszej aplikacji.



## Strona JSP z aplikacji internetowej („Kontakt.jsp”)

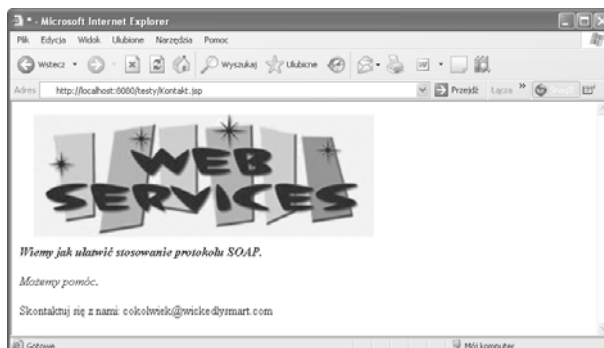
```
<html><body>

<jsp:include file="Naglowek.jsp"%>

Możemy pomóc.

Skontaktuj się z nami: ${initParam.glownyEmail}
</body></html>
```

W tym miejscu mówimy: „Wstaw TUTAJ, NA TEJ stronie całą zawartość pliku Naglowek.jsp, po czym kontynuuj przetwarzanie reszty tej strony JSP...”



## W rzeczywistości znaczenie tych akcji NIE jest identyczne...

Standardowa akcja `<jsp:include />` i dyrektywa `include` wyglądają bardzo podobnie i często prowadzą do identycznych zachowań, ale warto się przyjrzeć wygenerowanym na ich podstawie fragmentom kodu serwletów. Zaprezentowane poniżej przykłady skopiowaliśmy bezpośrednio z metod `_jspService()` wygenerowanych przez kontener Tomcat.

### Kod serwletu wygenerowany dla pliku nagłówka

```
out.write("\r<html>\r<body>\r

\rWiemy jak ułatwić stosowanie protokołu SOAP.
\r\r
</body>\r</html>\r");
```

*To proste... tekst strony jest po prostu przekazywany na wyjście.*

### Kod serwletu wygenerowany dla naszej strony JSP w oparciu o dyrektywę `include`

```
out.write("<html><body>\r");
```

*Fragment wyróżniony pogrubieniem jest DOKŁADNIE taki sam jak odpowiedni fragment wygenerowany przez stronę Naglowek.jsp.*

```
out.write("\r<html>\r<body>\r

\rWiemy jak ułatwić stosowanie protokołu SOAP.
\r\r
</body>\r</html>\r");
```

```
out.write("\r
\r\r\r<rm>Możemy pomóc.

\r\rSkontaktuj się z nami: ");
out.write((java.lang.String) org.apache.jasper.runtime.PageContextImpl.
 proprietaryEvaluate("${initParam.glownyEmail}", java.lang.String.class,
 (PageContext)_jspx_page_context, null, false));
```

```
out.write("\r\r\r</body></html>");
```

*Dyrektywa `include` po prostu pobiera zawartość pliku „Naglowek.jsp” i umieszcza ją w kodzie strony „Kontakt.jsp” jeszcze PRZED przystąpieniem do procesu tłumaczenia.*

### Kod serwletu wygenerowany dla naszej strony JSP w oparciu o standardową akcję `<jsp:include />`

```
out.write("<html><body>\r");
```

*To zupełnie inne rozwiązanie! Oryginalny plik Naglowek.jsp NIE znajduje się wewnątrz wygenerowanego serwletu, ma jedynie postać odpowiedniego wywołania realizowanego w czasie wykonywania serwletu...*

```
org.apache.jasper.runtime.JspRuntimeLibrary.include(request, response, "Naglowek.jsp", out, false);
```

```
out.write("\r
\r\r\r<rm>Możemy pomóc.

\r\rSkontaktuj się z nami: ");
out.write((java.lang.String) org.apache.jasper.runtime.PageContextImpl.
 proprietaryEvaluate("${initParam.glownyEmail}", java.lang.String.class,
 (PageContext)_jspx_page_context, null, false));
```

```
out.write("\r\r\r</body></html>");
```

# Dyrektywa include jest uwzględniana w czasie tłumaczenia; akcja standardowa <jsp:include> jest uwzględniana w czasie działania aplikacji

Skutki użycia **dyrektywy** `include` NIE różnią się od otwarcia kodu strony JSP i wklejenia tam zawartości pliku *Naglowek.jsp*. Innymi słowy, zastosowanie tej dyrektywy daje taki sam efekt jak powielenie kodu z pliku nagłówka w kodzie strony JSP. Jedyna różnica polega na tym, że to kontener wstawia odpowiedni kod w procesie jego tłumaczenia na serwet, dzięki czemu sami nie musimy tego kodu powielać we wszystkich naszych stronach. Jeśli użyjesz dyrektywy `include` we wszystkich swoich stronach, kontener automatycznie przejmie odpowiedzialność za skopiowanie kodu nagłówka do każdej z tych stron JSP jeszcze przed przystąpieniem do ich tłumaczenia i kompilacji wygenerowanych serwetów.

Sytuacja wygląda zupełnie inaczej w przypadku standardowej akcji `<jsp:include>`. Zamiast kopiować kod źródłowy z pliku *Naglowek.jsp*, standardowa akcja `include` wstawia *odpowiedź* tej strony w czasie wykonywania serwetu. Kluczowym aspektem decydującym o funkcjonowaniu standardowej akcji `<jsp:include>` jest fakt tworzenia przez kontener obiektu `RequestDispatcher` na podstawie atrybutu `page` i stosowania metody `include()`. Przydzielona (dołączona) strona JSP jest wykonywana nie tylko dla tych samych obiektów żądania i odpowiedzi, ale także w ramach tego samego wątku.

**Dyrektywa include wstawia KOD ŹRÓDŁOWY strony „Naglowek.jsp” w czasie tłumaczenia,  
ale standardowa akcja <jsp:include /> wstawia ODPOWIEDŹ strony „Naglowek.jsp” w czasie wykonywania serwetu.**

**P:** Dlaczego w tej sytuacji nie stosujemy zawsze standardowej akcji `<jsp:include>`? W ten sposób moglibyśmy przecież zagwarantować, że strona zawsze będzie wyświetlała najnowszą wersję wielokrotnie wykorzystywanego elementu.

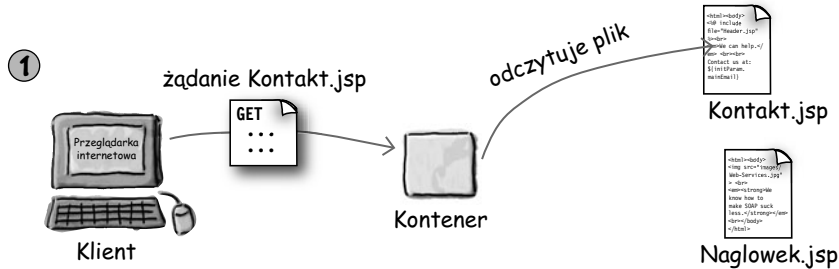
**U:** Dobrze się zastanów. Każde użycie standardowej akcji `<jsp:include>` oznacza dodatkowe obciążenie kontenera i opóźnienie wysłania odpowiedzi. Z drugiej strony, zastosowanie dyrektywy `include` powoduje jednorazowe wykonanie operacji dołączenia — w czasie tłumaczenia strony dołączającej. Jeśli więc jesteś pewien, że raz zatwierdzony plik do dołączania nie będzie w przyszłości zmieniany, dyrektywa `include` może być lepszym rozwiązaniem. Oczywiście istnieje jeszcze jeden argument przeciw stosowaniu tej dyrektywy — wygenerowana klasa serwetu jest nieco większa w przypadku zastosowania dyrektywy niż w przypadku użycia odpowiedniej akcji standardowej.

**P:** Wypróbowałem to w kontenerze Tomcat — przygotowałem statyczną stronę HTML i dołączyłem ją do strony JSP za pomocą dyrektywy `include`. Następnie zmieniłem ten plik HTML bez żadnego ponownego wdrażania i okazało się, że zwrócone przez stronę JSP dane wyjściowe uwzględniały wprowadzoną modyfikację! Skoro kontener potrafi zaktualizować takie dołączenie, po co w ogóle miałbym używać standardowej akcji `<jsp:include>`?

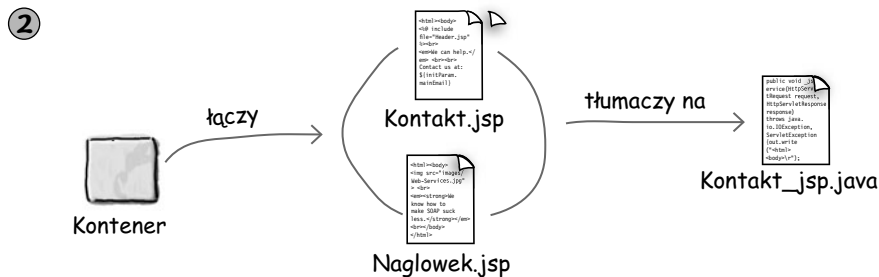
**U:** Ach! Po prostu korzystasz z bardzo przyjaznego kontenera (takim kontenerem jest np. Tomcat 5). Tak, większość nowoczesnych kontenerów wykorzystuje mechanizmy wykrywania zmian w dołączanych plikach i w razie konieczności ponownie te pliki tłumaczą. Problem w tym, że tego typu zachowań NIE GWARANTUJE SPECYFIKACJA JSP! Jeśli więc podczas pisania swojego kodu będziesz się opierał na takich rozwiązaniach, być może uniemożliwisz przyszłe przenoszenie tej aplikacji do innych kontenerów.

## Dyrektywa include przy pierwszym żądaniu

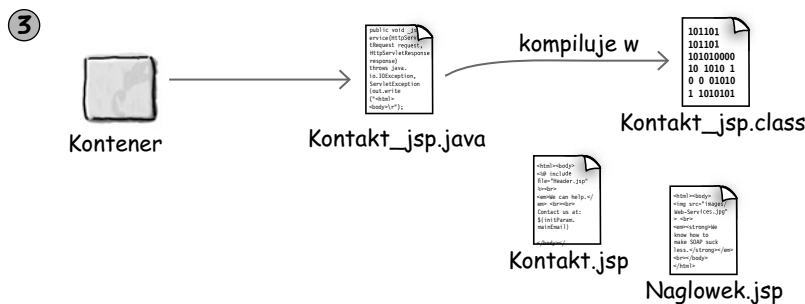
Zastosowanie **dyrektywy** include oznacza, że kontener będzie miał mnóstwo dodatkowej pracy, ale *tylko* podczas obsługi pierwszego żądania. Obsługa drugiego żądania nie wymaga już żadnych dodatkowych nakładów w czasie wykonywania.



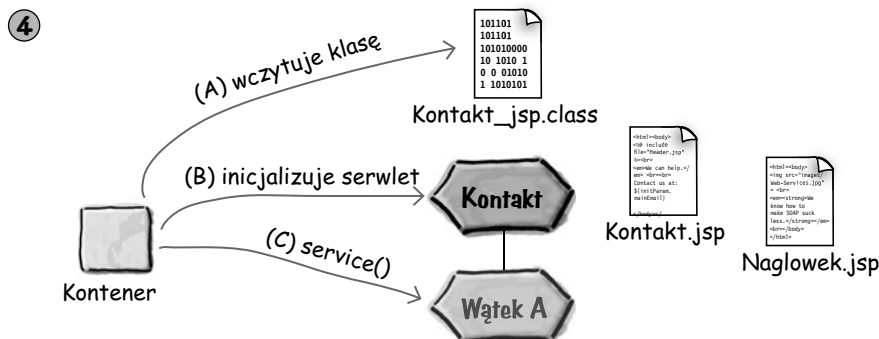
Klient przysyła żądanie strony *Kontakt.jsp*, która nie została jeszcze przetłumaczona na serwet. Kontener odczytuje kod strony *Kontakt.jsp* i rozpoczyna proces tłumaczenia.



Kontener znajduje w kodzie JSP dyrektywę include i łączy kod źródłowy pliku *Naglowek.jsp* z kodem strony *Kontakt.jsp*, po czym tłumaczy połączone strony na plik z kodem źródłowym odpowiedniego serwletu Javy.



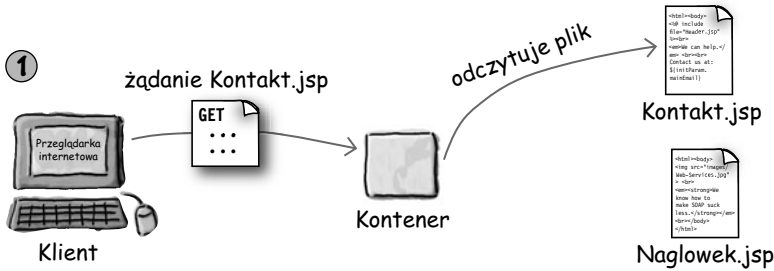
Kontener kompiluje przetłumaczony plik źródłowy w klasę serwletu. Na tym etapie plik z kodem źródłowym jest traktowany jak każdy inny kod serwletu, a poprzedni krok nigdy więcej nie będzie wykonywany, przynajmniej do czasu wprowadzenia zmian w pliku *Kontakt.jsp* (lub do momentu, w którym kontener wykorzysta mechanizm wykrywający ewentualne modyfikacje w dołączonym pliku *Naglowek.jsp*).



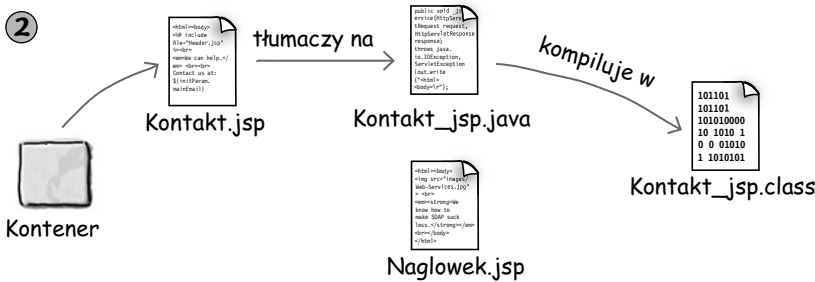
Aby zakończyć realizację żądania, kontener wczytuje właśnie skompilowaną klasę, inicjalizuje serwet (tworzy jego egzemplarz i wywołuje metodę `init()` dla nowego obiektu), przydziela wątek dla żądania i wywołuje metodę `_jspService()`. W odpowiedzi na drugie i każde kolejne żądanie kontener będzie wykonywał tylko krok (C), czyli będzie przydzielał wątek i wywoływał metodę `_jspService()`.

# Standardowa akcja <jsp:include> przy pierwszym żądaniu

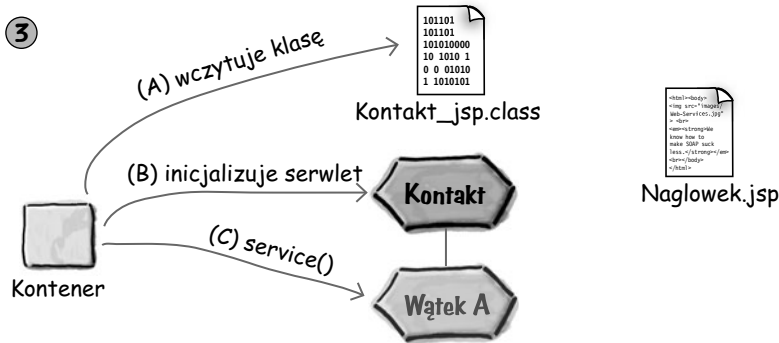
W przypadku zastosowania standardowej akcji <jsp:include> kontener będzie miał mniej pracy w czasie tłumaczenia, ale będzie odpowiedzialny za realizację większej liczby zadań podczas przetwarzania każdego żądania, w szczególności jeśli dołączany plik jest stroną JSP.



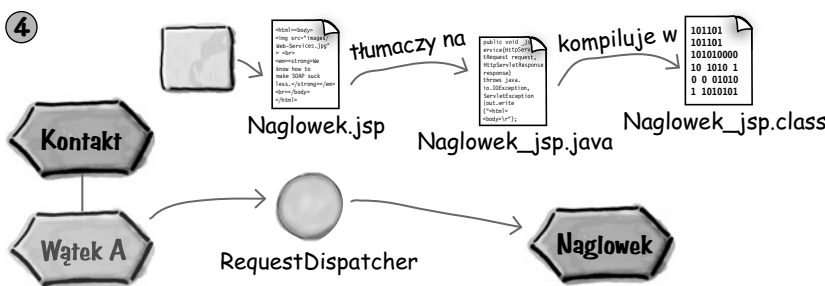
Klient przysyła żądanie strony *Kontakt.jsp*, która nie została jeszcze przetłumaczona na serwlet. Kontener odczytuje kod strony *Kontakt.jsp* i rozpoczyna proces tłumaczenia.



Kontener znajduje w kodzie JSP standardową akcję <jsp:include> i wykorzystuje ją do wstawienia w generowanym kodzie serwletu wywołania metody, która — w czasie wykonywania — dynamicznie połączy odpowiedź strony *Nagłówek.jsp* z odpowiedzią strony *Kontakt.jsp*. Kontener generuje serwlety dla obu plików JSP. (Ten element nie jest jasno sprecyzowany w specyfikacji, zatem przedstawiamy tylko jedno z możliwych rozwiązań).



Kontener kompiluje przetłumaczony plik źródłowy w klasę serwletu. Na tym etapie plik z kodem źródłowym jest traktowany jak każdy inny kod serwletu. Wygenerowany plik klasy serwletu jest wczytywany do wirtualnej maszyny Javy kontenera, w której jest inicjalizowany. Kontener przydziela następnie wątek dla danego żądania i wywołuje metodę `_jspService()`.



Serwlet *Kontakt* wywołuje metodę odpowiedzialną za dynamiczne dołączanie stron — w efekcie wykonywane są jakieś operacje opracowane przez producenta kontenera! Interesuje nas wyłącznie to, czy odpowiedź wygenerowana przez serwlet *Nagłówek* zostanie prawidłowo (we właściwym miejscu) połączona z odpowiedzią serwletu *Kontakt*. (NIE PRZEDSTAWIONO: w pewnym momencie strona *Nagłówek.jsp* jest tłumaczona i kompilowana, następnie gotowa już klasa serwletu jest wczytywana i inicjalizowana).



Oglądaj to!

Nazwy atrybutów są inne w dyrektywie `include` i inne w standardowej akcji `<jsp:include>`.

Zapamiętaj to! Spójrz na atrybuty stosowane w obu mechanizmach dołączania. Co je różni?

```
<%@ include file="Naglowek.jsp"%>
```

```
<jsp:include page="Naglowek.jsp" />
```

Tak. Atrybutem dyrektywy jest **file**, natomiast atrybutem standardowej akcji jest **page**! Być może łatwiej będzie Ci to zapamiętać, jeśli zauważysz, że dyrektywa dołączania `<%@ include file="foo.jsp"%>` jest wykorzystywana wyłącznie w czasie **tłumaczenia** (tak jak wszystkie inne dyrektywy). A podczas takiego tłumaczenia kontener operuje wyłącznie na **plikach** — pliki `.jsp` tłumaczy na pliki `.java`, a pliki `.java` kompiluje do postaci plików `.class`. Z drugiej strony, standardowa akcja `<jsp:include page="foo.jsp">` — jak wszystkie akcje standardowe — jest wykonywana w czasie obsługi **żądania**, kiedy kontener przetwarza **strony** żądane przez klienta.

**P:** Czy dołączane strony JSP mogą zawierać własną dynamiczną treść? Przecież użyta w zaprezentowanych przykładach strona **Naglowek.jsp** równie dobrze mogłaby być statyczną stroną **Naglowek.html**.

**U:** Skoro użyliśmy strony JSP, odpowiedź brzmi tak: można dołączać strony dynamiczne (ale masz rację — w tym przypadku moglibyśmy użyć dla nagłówka statycznej strony HTML i nasze rozwiązanie działałoby w identyczny sposób). Istnieje jednak kilka istotnych ograniczeń: dołączona strona **NIE MOŻE** zmieniać kodu stanu odpowiedzi ani ustawiać nagłówków (co oznacza, że w kodzie dołączanej strony JSP nie możemy wywoływać np. metody `addCookies()`). W przypadku podjęcia takiej próby w dołączonej stronie JSP nie otrzymasz informacji o błędzie — po prostu nie uzyskasz efektu, którego oczekiwałeś.

**P:** A jeśli za pomocą statycznej dyrektywy `include` dołączymy stronę dynamiczną, czy będzie to oznaczało, że dynamiczna zawartość strony zostanie wygenerowana tylko raz?

**U:** Przypuśćmy, że dołączasz stronę JSP zawierającą wyrażenie EL, które wywołuje funkcję `rzut()` generującą liczbę losową. Pamiętaj, że w przypadku zastosowania dyrektywy `include` takie wyrażenie EL zostanie po prostu skopiowane do kodu dołączającej strony JSP. Zatem za każdym razem, gdy klient przyśle żądanie do takiej strony, zostanie wykonane wyrażenie EL i serwet wygeneruje nową liczbę losową. Wbij sobie do głowy jedno: **w przypadku użycia dyrektywy `include` kod źródłowy dołączanej strony staje się CZĘŚCIĄ strony z dyrektywą `include`.**



Oglądaj to!

Dyrektywa `include` jest wrażliwa na umiejscowienie!

I jest to **JEDYNA** dyrektywa, której położenie w kodzie JSP odgrywa jakąś rolę. Inaczej jest na przykład w przypadku dyrektywy `page`, którą możemy umieścić w dowolnym miejscu strony, chociaż zgodnie z konwencją większość programistów i projektantów umiejscawia wszystkie swoje dyrektywy `page` na samym początku strony.

Jednak dyrektywa `include` dokładnie wskazuje kontenerowi, GDZIE należy wstawić kod źródłowy pochodzący z dołączanego pliku! Przykładowo, jeśli dołączymy do naszej strony zarówno nagłówek, jak i stopkę, opracowany w ten sposób kod JSP może mieć następującą postać:

```
<html><body>
```

```
<%@ include file="Naglowek.html"%>

```

```
Możemy pomóc.


```

```
Skontaktuj się z nami: ${initParam.
glownyEmail}

```

```
<%@ include file="Stopka.html"%>
```

```
</body></html>
```

Ta dyrektywa musi znajdować się na samym końcu kodu Twojej strony JSP (przed znacznikami zamykającymi), przynajmniej jeśli chcesz, aby właśnie w tym miejscu była wyświetlana zawartość strony **Stopka.html**. Pamiętaj, że cała strona JSP plus zawartość dwóch dołączonych plików zostanie połączona w ramach jednej wielkiej strony — **KOLEJNOŚĆ MA ZNACZENIE!**

Tak, także standardowa akcja `<jsp:include>` jest rzecz jasna wrażliwa na umiejscowienie, ale nie wspominaliśmy o tym, ponieważ w tym przypadku jest to bardziej oczywiste niż w przypadku dyrektywy `include`.

WITAM! Czy naprawdę PRZYJRZAŁEŚ SIĘ wygenerowanemu kodowi serwletu i przeanalizowałeś zawartą tam dyrektywę include? Przecież widać tam wyraźnie zagnieżdżone znaczniki HTML i BODY! Takie rozwiązanie jest złe i głupie.



### Och! Ona ma rację...

Zastanów się, co tak naprawdę zrobiliśmy. Stworzyliśmy stronę nagłówka (*Naglowek.jsp*). Była to przyjemna dla oka i samodzielna strona JSP — była kompletna, ponieważ zawierała otwierające i zamykające znaczniki HTML i BODY. Następnie opracowaliśmy stronę *Kontakt.jsp* i także w jej kodzie umieściliśmy odpowiednie znaczniki otwierające i zamykające. No cóż, czyż sami nie mówiliśmy, że *cała* zawartość dołączanego pliku jest (wirtualnie) wklejana do strony zawierającej dyrektywę include? **Jak cała, to cała.**

Przedstawiony poniżej fragment kodu, który pochodzi z wygenerowanego serwletu, NIE będzie działał we wszystkich przeglądarkach. W naszej przeglądarce zadziałał tylko dlatego, że mieliśmy szczęście.

```
out.write("<html><body>\r");
```

```
out.write("\r<html>\r<body>\r
```

```

```

```

\rWiemy jak ułatwić stosowanie protokołu SOAP.
```

```

\r\r
```

```
</body>\r</html>\r");
```

Faktycznie!!

```
out.write("\r
\r\r\r<rm>Możemy pomóc.

\r\rSkontaktuj się
z nami: ");
```

```
out.write((java.lang.String) org.apache.jasper.runtime.PageContextImpl.
 proprietaryEvaluate("${initParam.glownyEmail}", java.lang.String.class,
 (PageContext)_jspx_page_context, null, false));
```

```
out.write("\r\r\r</body></html>");
```

**NIE umieszczaj otwierających i zamykających znaczników HTML i BODY wewnątrz dołączanych elementów wielokrotnego użytku!**

**Projektuj i napisz swoje elementy szablonu układu (nagłówki, paski nawigacji itp.) z założeniem, że będą one dołączane do kodu INNYCH stron internetowych.**

Nie oczekuj  
ode MNIE odrzucania  
wszystkich nadmiarowych  
znaczników otwierających  
i zamykających.



## POWINNIŚMY to zrobić w ten sposób

W tym miejscu usuniemy z dołączanych plików (nagłówka i stopki) znaczniki otwierające i zamykające. Oznacza to, że dołączane pliki nie będą już mogły samodzielnie generować poprawnych stron HTML; będą uzależnione od operacji dołączania w ramach większych struktur. Struktur zawierających znaczniki `<html><body>` oraz `</body></html>`. Taki jest właśnie cel naszego zabiegu — projektujemy te fragmenty wielokrotnego użytku po to, aby stworzyć kompletne układy złożone z mniejszych elementów bez konieczności ręcznego powielania kodu. Tego rodzaju fragmenty w *zalozeniu* nie mają funkcjonować jako samodzielne strony internetowe.

### ① Plik nagłówka („Naglowek.jsp”)

```


Wiemy jak ułatwić stosowanie protokołu SOAP.

```

### ② Kontakt.jsp

```
<html><body>

<%@ include file="Naglowek.jsp"%>

Możemy pomóc.

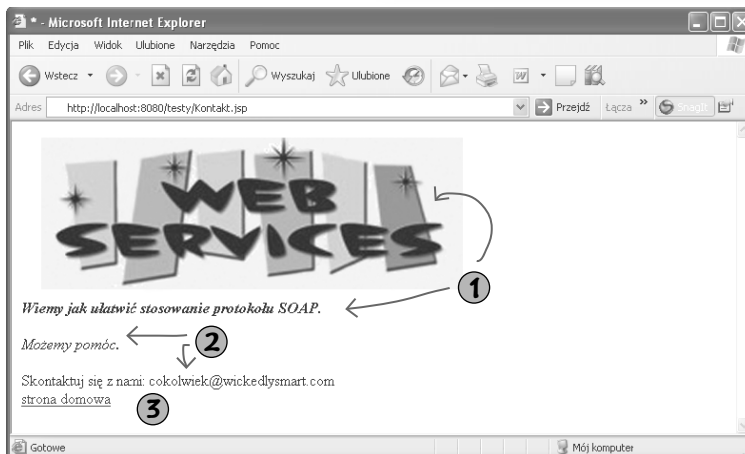
Skontaktuj się z nami: ${initParam.glownyEmail}

<%@ include file="Stopka.html"%>
</body></html>
```

Zwróć uwagę na fakt  
usunięcia z dołączanych  
plików wszystkich  
znaczników HTML  
i BODY.

### ③ Plik stopki ("Stopka.jsp")

```
strona domowa
```



Uwaga: przedstawiona idea odrzucania otwierających i zamykających znaczników dotyczy OBU mechanizmów dołączania — standardowej akcji `<jsp:include>` oraz dyrektywy `include`.

# Dostosowywanie dołączanej treści za pomocą standardowej akcji <jsp:param>

No dobrze, zatem dysponujemy nagłówkiem, który w założeniu ma być wyświetlany w taki sam sposób na każdej stronie naszej aplikacji internetowej. Ale co będzie, jeśli zechcemy zmodyfikować określoną część tego nagłówka? Co będzie, jeśli będziemy chcieli zastosować zależny od strony podtytuł będący częścią tego nagłówka i jeśli ten podtytuł będzie z oczywistych względów zależny od danej strony?

Mamy kilka możliwości.

**Rozwiązanie nierozważne:** umieszczamy informacje dotyczące podtytułu na głównej stronie, na przykład w formie pierwszego elementu za dyrektywą bądź standardową akcją dołączającą nagłówek.

**Rozwiązanie nieco lepsze:** przekazujemy do dołączanej strony informacje dotyczące podtytułu w postaci nowego parametru żądania.

**Dlaczego to rozwiązanie jest dobre:** jeśli przekazane informacje dotyczące podtytułu mają być częścią nagłówka, ale z drugiej strony są tym elementem, który różni się na poszczególnych stronach, nadal chcemy, aby właśnie część nagłówkowa szablonu odpowiadała za decyzje dotyczące sposobu wyświetlania i treści podtytułu ostatecznej wersji strony. Innymi słowy, chcemy pozostawić osobie, która zaprojektowała nagłówek, decyzję odnośnie sposobu wizualizacji podtytułu!

## Strona JSP, która wyświetla dołączony nagłówek

```
<html><body>
```

Spójrz tutaj... brakuje  
ukośnika zamykającego!

```
<jsp:include page="Naglowek.jsp" >
```

```
<jsp:param name="podTytuł" value="Wyciągamy żądło z protokołu SOAP." />
```

```
</jsp:include>
```

```


```

```
Grupa wsparcia dla usług internetowych.


```

```
Skontaktuj się z nami: ${initParam.glownyEmail}

```

```
</body></html>
```

Standardowa akcja <jsp:include> może mieć CIAŁO, zatem możemy dodawać (lub zastępować) parametry żądania, które będą wykorzystywane już w ramach dołączanej struktury.

## Dołączany nagłówek, który WYKORZYSTUJE nowy parametr („Naglowek.jsp”)

```


```

```
${param.podTytuł}

```



Na poziomie dołączanego pliku parametr ustawiony za pomocą standardowej akcji <jsp:param> jest traktowany tak jak każdy INNY parametr żądania. W tym przypadku wykorzystano wyrażenie języka EL do odczytania wartości tego parametru.

Uwaga: Przedstawione rozwiązanie oparte na parametrach nie miałoby oczywiście najmniejszego sensu w przypadku użycia dyrektywy include (która nie jest dynamiczna), zatem proponowana idea ma zastosowanie WYŁĄCZNIE w przypadku standardowej akcji <jsp:include>.

Jedno nie daje mi spokoju... skoro mogę dołączyć jedną stronę JSP do innej, co będzie, jeśli zechcę przekazać żądanie z jednej strony JSP do innej? Jeśli klient uzyskał dostęp do mojej strony i nie zalogował się w systemie, chcę go odesłać na inną stronę...



## Akcja standardowa <jsp:forward>

MOŻESZ przekazać żądanie z jednej strony JSP do innej. Możesz także przekazać żądanie ze strony JSP do serwletu lub z jednej strony JSP do dowolnego innego zasobu w ramach tej samej aplikacji internetowej.

Oczywiście w większości przypadków nie będziemy *chcieli* tego robić, ponieważ zgodnie z modelem MVC widok ma pozostawać tylko widokiem i jako taki nie ma żadnego interesu w wykonywaniu logiki kontroli! Innymi słowy, określanie, czy użytkownik jest zalogowany czy nie, nie powinno należeć do widoku — odpowiednią decyzję powinien podejmować ktoś *inny* (konkretnie kontroler) jeszcze przed ewentualnym przekazaniem żądania do widoku.

Na razie jednak spróbujmy zapomnieć o (skądinąd słusznych) regułach modelu MVC i przekonajmy się, jak moglibyśmy to zadanie zrealizować, gdybyśmy *musieli* przekazać żądanie dalej ze strony JSP do jakiegoś innego składnika aplikacji.

Po co mielibyśmy sobie utrudniać życie, skoro w praktyce nigdy nie będziemy zmuszeni do tego typu działań? Cóż, pewnego dnia *możesz* się znaleźć w sytuacji, w której właśnie standardowa akcja <jsp:forward> będzie korzystnym rozwiązaniem. Co więcej, tak jak w przypadku większości zagadnień poruszanych w tej książce (i testowanych na egzaminie), możliwości jakie daje akcja <jsp:forward> po prostu *warto znać*. Wśród niezliczonych stron JSP pewnego dnia możesz stanąć przed koniecznością konserwacji (lub w idealnej sytuacji — *udoskonalenia*) tak skonstruowanej strony.

### Warunkowe przekazywanie żądań...

Wyobraź sobie, że jesteś stroną JSP i zakładasz, że będziesz wywoływany przez żądania zawierające parametr *nazwaUzytkownika*. Ponieważ w swoich działaniach liczysz na obecność tego parametru, w pierwszej kolejności chcesz sprawdzić, czy parametr *nazwaUzytkownika* nie jest równy null. Jeśli nie jest, nie ma oczywiście problemu — możesz zakończyć generowanie odpowiedzi. Jeśli jednak parametr *nazwaUzytkownika* jest równy null, chcesz w tym miejscu przerwać swoje działanie i przekazać całe to żądanie gdzieś *indziej* — np. do innej strony JSP, która poprosi użytkownika o podanie potrzebnego parametru.

Wiemy już, jak można to zadanie zrealizować w oparciu o elementy skryptowe:

#### Strona JSP z warunkowym przekazywaniem żądań (Witaj.jsp)

```
<html><body>
Witamy na naszej stronie!

<% if (request.getParameter("nazwaUzytkownika") == null) { %>

 <jsp:forward page="ObsluzTo.jsp" />

<% } %>

Witaj, ${param.nazwaUzytkownika}

</body></html>
```

Testujemy parametr żądania.

Jeśli parametr żądania jest równy null, przekazujemy dane żądanie (dokładnie tak jak z wykorzystaniem obiektu *RequestDispatcher*) do strony wskazanej w atrybucie standardowej akcji *<jsp:forward>*.

Skoro doszliśmy aż tutaj, parametr *nazwaUzytkownika* musiał zawierać jakąś wartość! Jeśli żądanie zostanie przekazane do innego składnika aplikacji internetowej, ta strona **NICZEGO** nie wyświetli.

#### Strona JSP, do której będziemy przekazywać żądania (ObsluzTo.jsp)

```
<html><body>
Przykro nam... musisz się ponownie zalogować.

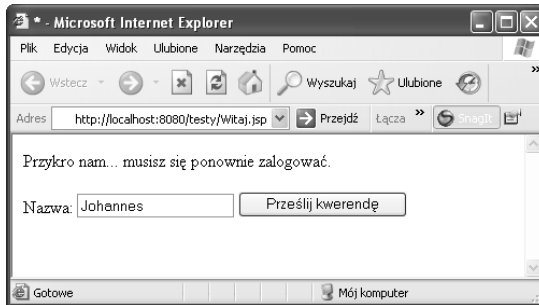
<form action="Witaj.jsp" method="get">
Nazwa: <input name="nazwaUzytkownika" type="text">
<input name="Submit" type="submit">
</form>
</body></html>
```

Mamy tutaj zwykłą, tradycyjną stronę, która pobierze od użytkownika niezbędny parametr żądania i odeśle żądanie do strony JSP, którą właśnie opuściliśmy... Witaj.jsp.

## Jak to działa?

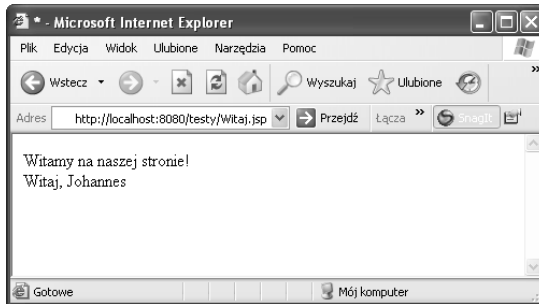
Kiedy po raz *pierwszy* zażadasz strony *Witaj.jsp*, zostanie wykonany zakodowany w JSP warunkowy test, który odkryje, że parametr żądania nazwaUzytkownika nie ma wartości, i przekaże żądanie dalej do strony *ObsluzTo.jsp*. Zakładając, że użytkownik poda nazwę w odpowiednim polu tekstowym formularza, *drugie* żądanie nie spowoduje kolejnego wywołania strony *ObsluzTo.jsp*, ponieważ parametr żądania nazwaUzytkownika będzie zawierał wartość różną od null.

### Pierwsze żądanie dla strony Witaj.jsp



Zaczekaj chwilę... gdzie się podziały słowa „Witamy na naszej stronie!”? Przecież ten tekst znajduje się w kodzie strony Witaj.jsp przed wyrażeniem przekazania żądania... dlaczego więc nie został wyświetlony w ramach pierwszej odpowiedzi?

### Drugie żądanie dla strony Witaj.jsp



Jak to się stało, że tekst „Witamy na naszej stronie!” nie został wyświetlony za pierwszym razem?

## W przypadku użycia akcji standardowej <jsp:forward> bufor jest zerowany PRZED przekazaniem żądania

Kiedy następuje przekazanie żądania, zasób, do którego to żądanie trafia, rozpoczyna swoją pracę z czystym (pustym) buforem odpowiedzi! Inaczej mówiąc, wszystkie dane zapisane w odpowiedzi przed przekazaniem żądania są usuwane.

Nie ma  
niemądrych pytań

**P:** Takie rozwiązanie jest oczywiście uzasadnione, jeśli strona jest buforowana... ponieważ wszystko, co zapisujemy na wyjściu, jest umieszczane w buforze, a kontener po prostu ten bufor opróżnia. Ale co będzie, jeśli zatwierdzimy odpowiedź jeszcze PRZED przekazaniem żądania? Co będzie, jeśli np. zapiszemy coś w odpowiedzi i wywołamy metodę flush() dla obiektu out?

**U:** No dobrze, mamy świadomość, że zadajesz to pytanie wyłącznie w celu zaspokojenia swojej intelektualnej ciekawości, ponieważ powszechnie wiadomo, że wywołanie metody flush() byłoby wyjątkowo głupie i bezcelowe. Zapewne i Ty zdajesz sobie z tego sprawę. Ale wiesz także, że równie niesamowite pomysły mogą mieć twórcy egzaminu, ponieważ Twoi leniwi i nieskorzy do nauki współpracownicy mogą bezmyślnie wprowadzić tego typu rozwiązania do swojego kodu, a wówczas lepiej będzie, jeśli będziesz znał skutki podobnych działań. Prawdopodobnie domyślasz się, jaka będzie odpowiedź. Warto jednak przeanalizować konkretny przykład:

```
<html><body>
```

Witamy na naszej stronie!

```
<% out.flush(); %>
```

```
<% if (request.getParameter("nazwaUzytkownika") == null) { %>
```

```
 <jsp:forward page="ObsluzTo.jsp" /> <% } %>
```

```
Witaj, ${param.nazwaUzytkownika}
```

```
</body></html>
```

Kontener posłusznie zatwierdza (wysyła) tekst „Witamy na naszej stronie!” w formie odpowiedzi i dopiero potem odnajduje w kodzie strony wyrażenie przekazujące żądanie. Och! **Za późno.** Niestety, zostanie zwrócony wyjątek `IllegalStateException`.

Problem w tym, że nikt tego wyjątku nawet nie zobaczy! Przeglądarka klienta wyświetli jedynie słowa „Witamy na naszej stronie!”... i nic więcej. Wyrażenie przekazujące żądanie rzuci co prawda wyjątek, ale dla kontenera będzie już za późno, aby cofnąć raz wysłaną odpowiedź, zatem klient będzie miał możliwość zapoznania się tylko z tekstem już zatwierdzonym. Żądanie nie zostanie przekazane dalej, bieżąca strona nie zostanie przetworzona do końca. To już koniec historii tej strony. Zapamiętaj: **niegdy nie zatwierdzaj odpowiedzi przed przekazaniem żądania!**

W przypadku przekazania żądania NIC z tego, co zapisałeś przed tym przekazaniem, nie zostanie wyświetlone.

Nie rozumiem, dlaczego nasze ostateczne rozwiązanie opiera się na skryptcie. Przecież MÓWIONO mi, że w tym rozdziale nie będziemy już używali skryptletów. Gdyby tylko istniał jakiś sposób wykonania testu warunkowego bez konieczności wracania do elementów skryptowych...



## Ona nie wie jeszcze o znacznikach JSTL

Kiedy w kodzie swojego serwletu potrzebujesz więcej funkcjonalności, czegoś wykraczającego poza możliwości standardowych akcji JSP i wyrażeń języka EL, wcale nie musisz się uciekać do elementów skryptowych. W następnym rozdziale przedstawimy sposób wykorzystywania standardowej biblioteki znaczników JSP w wersji 1.1 (*JSP Standard Tag Library 1.1* — JSTL 1.1) do zapewnienia wszystkich potrzebnych mechanizmów — wyłącznie w oparciu o odpowiednią kombinację znaczników i wyrażeń języka EL. Oto krótkie spojrzenie na rozwiązanie problemu warunkowego przekazywania żądań *bez stosowania skryptów*.

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
```

```
<html><body>
```

```
Witamy na naszej stronie!
```

```
<c:if test="${empty param.nazwaUzytkownika}" >
```

```
 <jsp:forward page="ObsluzTo.jsp" />
```

```
</c:if>
```

```
Witaj, ${param.nazwaUzytkownika}
```

```
</body></html>
```

W ten sposób zastępujemy stosowany dotychczas test warunkowy w postaci skryptletu.

Deklarujemy dyrektywę taglib nazywającą bibliotekę, która zawiera odpowiednie znaczniki.

Nawiasem mówiąc... prawdopodobnie nie będziesz mógł tak od razu uruchomić tej strony JSP, ponieważ nie masz w swojej aplikacji internetowej biblioteki JSTL. Omówimy ten problem w kolejnym rozdziale.

# Przegląd standardowych akcji związanych z komponentami

### KLUCZOWE ZAGADNIENIA



- Standardowa akcja `<jsp:useBean>` definiuje zmienną, która przechowuje referencję albo do *istniejącego* atrybutu komponentu, albo — jeśli taki komponent jeszcze nie istnieje — do *nowego* komponentu.
- Standardowa akcja `<jsp:useBean>` MUSI zawierać atrybut „id” deklarujący nazwę zmiennej, która będzie wykorzystywana w kodzie bieżącej strony JSP do odwoływania się do danego komponentu.
- Jeśli w standardowej akcji `<jsp:useBean>` nie zdefiniujemy atrybutu „scope”, zostanie zastosowany domyślny zasięg strony („page”).
- Atrybut „class” jest opcjonalny i deklaruje typ klasy, która zostanie użyta w razie konieczności utworzenia nowego komponentu. Zadeklarowany w ten sposób typ musi być publiczny, nieabstrakcyjny i udostępniać publiczny, bezargumentowy konstruktor.
- Jeśli w standardowej akcji `<jsp:useBean>` umieścimy atrybut „type”, musi to być typ, do którego dany komponent może być rzutowany.
- Jeśli w standardowej akcji `<jsp:useBean>` użyjemy atrybutu „type”, ale NIE zadeklarujemy atrybutu „class”, dany komponent będzie musiał już istnieć, ponieważ nie będzie znany typ klasy, której egzemplarz należałoby utworzyć dla nowego komponentu.
- Znacznik `<jsp:useBean>` może mieć ciało, którego zawartość będzie przetwarzana TYLKO wtedy, gdy w wyniku standardowej akcji `<jsp:useBean>` zostanie utworzony nowy komponent (a więc wtedy, gdy w określonym (lub domyślnym) zasięgu nie udało się znaleźć komponentu z podanym identyfikatorem).
- Głównym celem definiowania ciała znacznika `<jsp:useBean>` jest ustawianie właściwości nowego komponentu za pomocą innej akcji standardowej: `<jsp:setProperty>`.
- Akcja standardowa `<jsp:setProperty>` musi zawierać atrybut „name” (który z kolei musi pasować do atrybutu „id” znacznika `<jsp:useBean>`) oraz atrybut „property”. Wartość atrybutu „property” musi być albo nazwą rzeczywistej właściwości, albo symbolem wieloznacznym „\*“.
- Jeśli nie dołączymy do standardowej akcji `<jsp:setProperty>` atrybutu „value”, kontener ustawi wartość odpowiedniej właściwości tylko wtedy, gdy będzie istniał parametr żądania z nazwą odpowiadającą nazwie tej właściwości. Jeśli natomiast w wartości atrybutu „property” użyjemy symbolu wieloznacznego (\*), kontener ustawi wartości wszystkich właściwości, których nazwy będą pasowały do nazwy parametru żądania (w takim przypadku pozostałe właściwości nie będą modyfikowane).
- Jeśli nazwa parametru żądania różni się od nazwy właściwości, a mimo to chcemy, aby wartość tej właściwości była równa wartości naszego parametru żądania, w znaczniku `<jsp:setProperty>` możemy użyć atrybutu „param”.
- Akcja `<jsp:setProperty>` wykorzystuje mechanizm introspekcji do dopasowania wskazanej właściwości do odpowiedniej metody ustawiającej komponentu `JavaBean`. Jeśli zamiast konkretnej właściwości użyto symbolu „\*”, JSP iteracyjnie przeszuka wszystkie parametry żądania celem ustawienia właściwości komponentu.
- Wartościami właściwości mogą być łańcuchy lub typy proste — standardowa akcja `<jsp:setProperty>` automatycznie wykona wszelkie niezbędne konwersje.

# Przegląd mechanizmów dołączania

## KLUCZOWE ZAGADNIENIA



- Możemy budować strony z komponentami wielokrotnego użytku w oparciu o jeden z dwóch mechanizmów dołączania — *dyrektywę include* oraz *standardową akcję <jsp:include>*.
- *Dyrektywa include* wykonuje operację dołączenia już w czasie tłumaczenia i tylko raz. Dyrektywa ta jest więc uważana za właściwy mechanizm w sytuacjach, gdy jest mało prawdopodobne, aby dołączana zawartość była modyfikowana już po wdrożeniu aplikacji.
- W praktyce *dyrektywa include* kopiuje całą zawartość dołączanego pliku i wkleja ją do strony, która tę dyrektywę zawiera. Kontener scala wszystkie dołączone pliki i kompiluje tylko jeden plik dla wygenerowanego serwletu. W czasie wykonywania aplikacji strona z dyrektywą *include* działa dokładnie tak, jakbyśmy samodzielnie zapisali cały kod źródłowy w jednym pliku.
- Akcja standardowa *<jsp:include>* jedynie dodaje odpowiedź dołączonej strony do odpowiedzi oryginalnej (dołączającej) strony, a cała operacja jest wykonywana w czasie wykonywania aplikacji. Akcja standardowa *<jsp:include>* jest więc uważana za właściwy mechanizm w sytuacjach, gdy dołączana treść może być aktualizowana po wdrożeniu aplikacji, a więc stanowi pod tym względem przeciwieństwo dyrektywy *include*.
- Oba mechanizmy umożliwiają dołączanie zarówno elementów dynamicznych (jak choćby kodu JSP z wyrażeniami języka EL), jak i statycznych stron HTML.
- Dyrektywa *include* jest jedyną dyrektywą JSP wrażliwą na umiejscowienie; dołączana zawartość jest dodawana do kodu strony dokładnie w tym miejscu, w którym umieszczono tę dyrektywę.
- Nazwy atrybutów dyrektywy *include* i standardowej akcji *<jsp:include>* są niespójne — dyrektywa wykorzystuje atrybut "file", natomiast standardowa akcja wykorzystuje atrybut "page".
- Musisz się upewnić, że Twoje komponenty wielokrotnego użytku nie zawierają znaczników otwierających i zamykających. W przeciwnym przypadku wygenerowane dane wyjściowe będą zawierały zagnieżdżone znaczniki otwierające i zamykające, które nie przez wszystkie przeglądarki są obsługiwane. Swoje fragmenty wielokrotnego użytku powinieneś projektować i konstruować przy założeniu, że będą one dołączane (wstawiane) do innych, większych struktur.
- Możesz dostosowywać dołączony plik przez ustawianie (lub zastępowanie) parametru żądania za pomocą standardowej akcji *<jsp:param>* umieszczonej w ciele standardowej akcji *<jsp:include>*.
- W tym rozdziale nie przedstawiliśmy dopuszczalnego rozwiązania, w którym standardowa akcja *<jsp:param>* jest wykorzystywana w ciele znacznika *<jsp:forward>*.
- JEDYNYMI miejscami, w których stosowanie standardowej akcji *<jsp:param>* jest uzasadnione, są standardowe akcje *<jsp:include>* oraz właśnie *<jsp:forward>*.
- Jeśli dla użytej w akcji standardowej *<jsp:param>* nazwy istnieje już parametr żądania z jakąś wartością, nowa wartość zastąpi wartość dotychczasową. W przeciwnym przypadku do żądania zostanie dodany nowy parametr.
- Istnieją pewne ograniczenia w funkcjonowaniu dołączonego zasobu: zasób ten nie może zmienić kodu stanu odpowiedzi ani ustawiać nagłówków.
- Akcja standardowa *<jsp:forward>* przekazuje dane żądanie (podobnie jak obiekt *RequestDispatcher*) do innego zasobu należącego do tej samej aplikacji internetowej.
- Podczas przekazywania żądania w pierwszej kolejności czyszczony jest bufor odpowiedzi! Zasób, do którego dane żądanie jest przekazywane, otrzymuje na starcie pusty bufor danych wyjściowych. Oznacza to, że wszystkie dane zapisane w odpowiedzi przed przekazaniem żądania zostaną utracone.
- Jeśli zatwierdzisz odpowiedź przed przekazaniem żądania dalej (np. przez wywołanie metody *out.flush()*), klient otrzyma jedynie zatwierdzone dane wyjściowe i nic więcej. W takim przypadku samo przekazanie żądania nie zostanie zrealizowane, nie zostanie też przetworzona reszta oryginalnej strony.



## BĄDŹ kontenerem ODPOWIEDZI

Uwaga: w tym znaczniku zdefiniowano typ, nie klasę.

Przyjrzyj się następującej standardowej akcji:

```
<jsp:useBean id="osoba" type="foo.Pracownik" scope="request" >
 <jsp:setProperty name="osoba" property="imie" value="Fred" />
</jsp:useBean >
```

Imię: `<jsp:getProperty name="osoba" property="imie" />`

Ciało **NIGDY** nie zostanie wykonane! Umieszczanie ciała wewnątrz znacznika `<jsp:useBean>` jest bezcelowe, jeśli użyto atrybutu type bez atrybutu class! Pamiętaj, że ciało znacznika jest wykonywane TYLKO wtedy, gdy tworzony jest nowy komponent, co nie może mieć miejsca w sytuacji, gdy w znaczniku tym zadeklarowano sam atrybut type (bez atrybutu class).

Jeśli dojdziemy aż tutaj, strona JSP wyświetli imię „Evan”.

- 1 Jaki będzie efekt wykonania następującego kodu serwletu?

```
foo.Osoba p = new foo.Pracownik();
p.setImie("Evan");
request.setAttribute("osoba", p);
```

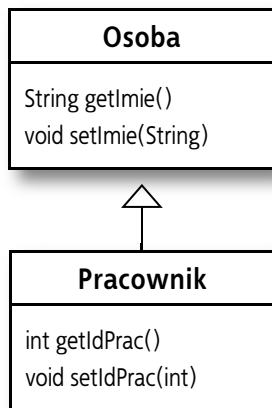
Zakończy się **BŁĘDEM** w czasie obsługi żądania! Atrybut „osoba” jest składowany w zakresie żądania, zatem znacznik `<jsp:useBean>` nie będzie działał, ponieważ określono w nim tylko typ. Kontener WIE, że jeśli określiliśmy w znaczniku `<jsp:useBean>` sam typ, w danym komponencie **MUSI** istnieć atrybut z określoną nazwą i zakresem.

- 2 Jaki będzie efekt wykonania następującego kodu serwletu?

```
foo.Osoba p = new foo.Osoba();
p.setImie("Evan");
request.setAttribute("osoba", p);
```

Tak naprawdę serwlet w takiej postaci nie zostanie nawet skompilowany. W tym przypadku trochę oszukiwaliśmy, ponieważ polecenie nie powinno brzmieć „Bądź kontenerem”, a raczej „Bądź KOMPILATOREM”. `foo.Osoba` jest klasą abstrakcyjną, zatem nie możemy jej konkretyzować (tworzyć jej obiektów).

abstrakcyjna



Obie klasy należą do pakietu „foo”.



- 1 Mamy dany formularz HTML, w którym zastosowano pola wyboru, aby umożliwić użytkownikom zaznaczanie wielu wartości dla parametru nazwanego **hobby**.

Które wyrażenia języka EL zwrócą pierwszą wartość parametru **hobby**?  
(Zaznacz wszystkie prawidłowe opcje).

- ☐ A. `${param.hobby}`
- ☐ B. `${paramValue.hobby}`
- ☐ C. `${paramValues.hobby[0]}`
- ☐ D. `${paramValues.hobby[1]}`
- ☐ E. `${paramValues[hobby][0]}`
- ☐ F. `${paramValues[hobby][1]}`

- 2 Mamy daną aplikację internetową, która przechowuje adres internetowy webmastera w postaci parametru inicjalizacji kontekstu nazwanego **email-webmastera**.

Które wyrażenia języka EL zwrócą właściwą wartość tego parametru?  
(Zaznacz wszystkie prawidłowe opcje).

- ☐ A. `<a href='mailto:${initParam.email-webmastera}'>napisz do mnie</a>`
- ☐ B. `<a href='mailto:${contextParam.email-webmastera}'>napisz do mnie</a>`
- ☐ C. `<a href='mailto:${initParam['email-webmastera']}'>napisz do mnie</a>`
- ☐ D. `<a href='mailto:${contextParam['email-webmastera']}'>napisz do mnie</a>`

2 Mamy daną następującą klasę Javy:

```
1. package com.mojafirma;
2. public class MojeFunkcje {
3. public static String witaj(String imie) {
4. return "Witaj "+imie;
5. }
6. }
```

Przedstawiona klasa w istocie odpowiada za obsługę funkcji będącej częścią następującej biblioteki znaczników: `<%@ taglib uri="http://mojafirma.com.tags" prefix="comp" %>`. Który wpis w deskrytorze biblioteki znaczników definiuje tę funkcję użytkownika w sposób umożliwiający jej wykorzystanie w wyrażeniu języka EL?

☐ A. `<taglib>`

```
...
<tag>
 <name>Witaj</name>
 <tag-class>com.mojafirma.MojeFunkcje</tag-class>
 <body-content>JSP</body-content>
</tag>
</taglib>
```

☐ B. `<taglib>`

```
...
<function>
 <name>Witaj</name>
 <function-class>com.mojafirma.MojeFunkcje</function-class>
 <function-signature>java.lang.String witaj(java.lang.String)
 </function-signature>
</function>
</taglib>
```

☐ C. `<web-app>`

```
...
<servlet>
 <name>Witaj</name>
 <servlet-class>com.mojafirma.MojeFunkcje</servlet-class>
</servlet>
</web-app>
```

☐ D. `<taglib>`

```
...
<function>
 <name>Witaj</name>
 <function-class>com.mojafirma.MojeFunkcje</function-class>
 <function-signature>witaj(java.lang.String)</function-signature>
</function>
</taglib>
```

#### 4 Mamy dany komponent:

```
1. package com.example;
2. public class Komponent {
3. private int wartosc;
4. public Komponent() { wartosc = 42; }
5. public int getWartosc() { return wartosc; }
6. public void setWartosc(int w) { wartosc = w; }
7. }
```

Zakładając, że nie został jeszcze utworzony żaden egzemplarz klasy **Komponent**, spróbuj określić, które standardowe akcje JSP utworzą nowy egzemplarz tej klasy i umieszczą go w zasięgu żądania. (Zaznacz wszystkie prawidłowe opcje).

- ☐ A. `<jsp:useBean name="mojKomponent" type="com.example.Komponent" />`
- ☐ B. `<jsp:makeBean name="mojKomponent" type="com.example.Komponent" />`
- ☐ C. `<jsp:useBean id="mojKomponent" class="com.example.Komponent" scope="request" />`
- ☐ D. `<jsp:makeBean id="mojKomponent" class="com.example.Komponent" scope="request" />`

#### 5 Mamy daną architekturę Model 1, w której pojedyncza strona JSP obsługuje wszystkie funkcje kontrolera — taka strona kontrolera musi mieć możliwość przydzielania żądań do innej strony JSP.

Który z przedstawionych poniżej kodów akcji standardowych wykona taki przydział?

- ☐ A. `<jsp:forward page="widok.jsp" />`
- ☐ B. `<jsp:forward file="widok.jsp" />`
- ☐ C. `<jsp:dispatch page="widok.jsp" />`
- ☐ D. `<jsp:dispatch file="widok.jsp" />`

**6** Mamy dany skryptlet:

```
11. <% java.util.List lista = new java.util.ArrayList();
12. lista.add("a");
13. lista.add("2");
14. lista.add("c");
15. request.setAttribute("lista", lista);
16. request.setAttribute("indeksListy", "1");
17. %>
18. <!-- tutaj wstaw odpowiednie wyrażenie --%>
```

Które z poniższych wyrażeń po wstawieniu w wierszu 18. przedstawionego kodu będzie poprawne i zwróci wartość c? (Zaznacz wszystkie prawidłowe opcje).

- ☐ A. `${lista.2}`
- ☐ B. `${lista[2]}`
- ☐ C. `${lista.indeksListy+1}`
- ☐ D. `${lista[indeksListy+1]}`
- ☐ E. `${lista['indeksListy' + 1]}`
- ☐ F. `${lista[lista[indeksListy]]}`

---

**7** Które zdania na temat stosowanego w języku EL operatora kropki (.) i operatora [] są prawdziwe? (Zaznacz wszystkie prawidłowe opcje).

- ☐ A. Wyrażenie `${foo.bar}` jest równoważne wyrażeniu `${foo[bar]}`.
- ☐ B. Wyrażenie `${foo.bar}` jest równoważne wyrażeniu `${foo["bar"]}`.
- ☐ C. Składnia wyrażenia `${foo["5"]}` jest poprawna, jeśli `foo` jest mapą.
- ☐ D. Wyrażenie `${header.User-Agent}` jest równoważne wyrażeniu `${header[User-Agent]}`.
- ☐ E. Wyrażenie `${header.User-Agent}` jest równoważne wyrażeniu `${header["User-Agent"]}`.
- ☐ F. Składnia wyrażenia `${foo[5]}` jest poprawna, jeśli `foo` jest tablicą lub egzemplarzem klasy `List`.

8 Mamy daną stronę JSP z następującym wierszem:

`${101 % 10}`

Co zostanie wyświetlone?

- ☐ A. 1
- ☐ B. 10
- ☐ C. 1001
- ☐ D. 101 % 10
- ☐ E. {101 % 10}

9 Mamy dane następujące wyrażenia EL:

10. `${param.pierwszeimie}`

11. `${param.drugieimie}`

12. `${param.nazwisko}`

13. `${paramValues.nazwisko[0]}`

Który z poniższych wyników reprezentuje dane wyjściowe wygenerowane przez przedstawiony fragment kodu strony JSP w przypadku przekazania następującego łańcucha zapytania:

`?pierwszeimie=John&nazwisko=Doe?`

- ☐ A. John Doe
- ☐ B. John Doe Doe
- ☐ C. John null Doe
- ☐ D. John null Doe Doe
- ☐ E. Zostanie rzucony wyjątek `NullPointerException`.

10 Które z poniższych wyrażeń zawierają prawidłowo użyte zmienne domyślne języka EL?  
(Zaznacz wszystkie prawidłowe opcje).

- ☐ A. `${cookies.foo}`
- ☐ B. `${initParam.foo}`
- ☐ C. `${pageContext.foo}`
- ☐ D. `${requestScope.foo}`
- ☐ E. `${header["User-Agent"]}`
- ☐ F. `${requestDispatcher.foo}`
- ☐ G. `${pageContext.request.requestURL}`

- 
- 11** Które zdania na temat standardowej akcji **<jsp:useBean>** są prawdziwe? (Zaznacz wszystkie prawidłowe opcje).
- ☐ A. Atrybut **id** jest opcjonalny.
  - ☐ B. Atrybut **scope** jest wymagany.
  - ☐ C. Atrybut **scope** jest opcjonalny, a jego domyślną wartością jest **request**.
  - ☐ D. W standardowej akcji można zdefiniować albo atrybut **class**, albo atrybut **type**, ale wymagane jest określenie przynajmniej jednego z nich.
  - ☐ E. Istnieje możliwość zdefiniowania w standardowej akcji zarówno atrybutu **class**, jak i atrybutu **type**, nawet jeśli ich wartości **NIE** są takie same.
- 
- 12** Jak dołączyłbyś dynamiczną treść do swojej strony JSP w sposób zbliżony do działania mechanizmu dołączania po stronie serwera (ang. Server-Side Include — SSI)? (Zaznacz wszystkie prawidłowe opcje).
- ☐ A. `<%@ include file="/segmenty/stopka.jspf" %>`
  - ☐ B. `<jsp:forward page="/segmenty/stopka.jspf" />`
  - ☐ C. `<jsp:include page="/segmenty/stopka.jspf" />`
  - ☐ D. 

```
RequestDispatcher dispatcher
= request.getRequestDispatcher("/segmenty/stopka.jspf");
dispatcher.include(request, response);
```
- 
- 13** Której ze standardowych akcji JSP można użyć w stronie HTML z bogatym układem graficznym do zaimportowania pliku obrazu do danej strony JSP?
- ☐ A. `<jsp:image page="logo.png" />`
  - ☐ B. `<jsp:image file="logo.png" />`
  - ☐ C. `<jsp:include page="logo.png" />`
  - ☐ D. `<jsp:include file="logo.png" />`
  - ☐ E. **NIE MOŻNA** tego zrobić wyłącznie w oparciu o standardowe akcje JSP.

14 Mamy daną klasę Javy:

```
1. package com.example;
2. public class MojeFunkcje {
3. public static String powtorz(int x, String str) {
4. // ciało metody
5. }
6. }
```

i następujący kod JSP:

```
1. <%@ taglib uri="/WEB-INF/mojefunkcje" prefix="moje" %>
2. <%-- tutaj wstaw odpowiedni kod --%>
```

Które z poniższych wyrażeń po wstawieniu w wierszu 2. kodu JSP będzie prawidłowym wywołaniem funkcji języka EL?

- ☐ A. `${powtorz(2, "420")}`
- ☐ B. `${powtorz("2", "420")}`
- ☐ C. `${moje:powtorz(2, "420")}`
- ☐ D. `${moje:powtorz("2", "420")}`
- ☐ E. Prawidłowego wywołania NIE MOŻNA określić.

15 Mamy dany komponent:

```
10. public class MojKomponent {
11. private java.util.Map parametry;
12. private java.util.List obiekty;
13. private String nazwa;
14. public java.util.Map getParametry() { return parametry; }
15. public String getNazwa() { return nazwa; }
16. public java.util.List getObiekty() { return obiekty; }
17. }
```

Które z poniższych wyrażeń spowodują błędy (zakładając, że mamy dostęp do atrybutu nazwanego `mojkomponent`, którego typem jest zdefiniowana przed chwilą klasa `MojKomponent`)? (Zaznacz wszystkie prawidłowe opcje).

- ☐ A. `${mojkomponent.nazwa}`
- ☐ B. `${mojkomponent["nazwa"]}`
- ☐ C. `${mojkomponent.obiekty.a}`
- ☐ D. `${mojkomponent["parametry"].a}`
- ☐ E. `${mojkomponent.parametry["a"]}`
- ☐ F. `${mojkomponent["obiekty"].a}`

**16** Mamy daną stronę JSP:

1. **Użytkownik poprawnie się zalogował lub wylogował:**
2. **`${param.zalogowany or param.wylogowany}`.**

Gdyby żądanie zawierało łańcuch zapytania „`wylogowany=true`”, jaki byłby wyświetlony na stronie wynik powyższego wyrażenia?

- ☐ A. **Użytkownik poprawnie się zalogował lub wylogował: `false`.**
  - ☐ B. **Użytkownik poprawnie się zalogował lub wylogował: `true`.**
  - ☐ C. **Użytkownik poprawnie się zalogował lub wylogował: `${param.zalogowany or param.wylogowany}`.**
  - ☐ D. **Użytkownik poprawnie się zalogował lub wylogował: `param.zalogowany or param.wylogowany`.**
  - ☐ E. **Użytkownik poprawnie się zalogował lub wylogował: `or true`.**
- 

**17** Które zdania na temat dostępnych w języku EL operatorów dostępu są prawdziwe? (Zaznacz wszystkie prawidłowe opcje).

- ☐ A. Wszędzie tam, gdzie można stosować operator kropki (`.`), równie dobrze można używać operatora `[]`.
  - ☐ B. Wszędzie tam, gdzie można stosować operator `[]`, równie dobrze można używać operatora kropki (`.`).
  - ☐ C. Jeśli operator kropki (`.`) jest używany do uzyskiwania dostępu do nieistniejącej właściwości komponentu, zostanie zwrócony wyjątek czasu wykonywania.
  - ☐ D. Istnieją pewne sytuacje, w których musimy użyć operatora kropki (`.`), ale są też sytuacje, w których niezbędne jest zastosowanie operatora `[]`.
- 

**18** W kodzie naszej strony JSP użyliśmy następującego fragmentu:

```
<jsp:include page="/jspf/naglowek.html"/>
```

Nasza strona JSP jest częścią aplikacji z katalogiem głównym kontekstu myapp.

Wiedząc, że najwyższy poziom struktury katalogów tej aplikacji tworzy katalog myapp, określ, która z wymienionych poniżej ścieżek do pliku naglowek.html jest prawidłowa?

- ☐ A. `/naglowek.html`
- ☐ B. `/jspf/naglowek.html`
- ☐ C. `/myapp/jspf/naglowek.html`
- ☐ D. `/includes/jspf/naglowek.html`

- 19 Internetowy sprzedawca biżuterii chce dostosowywać katalog oferowanych produktów do preferencji zalogowanych użytkowników. Chce proponować swoim klientom oferty specjalne w miesiącach ich urodzin. Oferty specjalne sprzedawcy są składowane w strukturze **Map<String, Special[]>** identyfikowanej w zasięgu aplikacji jako **specjalne** i aktualizowanej codziennie.

Aplikacja obejmuje komponent składowany w formie atrybutu zasięgu sesji i nazwany **daneUzytkownika**. W wyniku wywołania metody **getDataUrodzenia().getMiesiac()** tego komponentu otrzymujemy miesiąc urodzin danego użytkownika.

Który z poniższych fragmentów kodu umożliwia prawidłowe uzyskanie odpowiedniej oferty specjalnej?

- ☐ A. `${applicationScope[daneUzytkownika.dataUrodzenia.miesiac.specjalne]}`
- ☐ B. `${applicationScope.specjalne[daneUzytkownika.dataUrodzenia.miesiac]}`
- ☐ C. `${applicationScope["specjalne"].daneUzytkownika.dataUrodzenia.miesiac}`
- ☐ D. `${applicationScope["daneUzytkownika.dataUrodzenia.miesiac"].specjalne}`

- 20 Aplikacja internetowa wykorzystywana przez popularną wypożyczalnię filmów składa się w formie atrybutu sesji strukturę **List<Movie>** reprezentującą filmy wybrane przez użytkownika. Losowo wybrany zwiastun filmu z tej listy musi być prezentowany na stronie głównej użytkownika za każdym razem, gdy odwiedza on witrynę wypożyczalni.

Kierownictwo firmy uważa, że podobną funkcję w niedalekiej przyszłości należałoby wprowadzić także na pozostałych stronach prezentujących listy dostępnych filmów. Za obsługę strumienia wideo odpowiadają zwykle elementy HTML-a, zatem samo dodanie odpowiednich konstrukcji nie jest dużo trudniejsze (mimo bardziej złożonych znaczników) od wstawiania obrazów.

Zespół programistów potrzebuje rozwiązania gwarantującego jednocześnie elastyczność i łatwą konserwację. Jedną z możliwości jest utworzenie odpowiedniej funkcji języka EL. Poniższe stwierdzenia pochodzą z dyskusji prowadzonej przez członków wspomnianego zespołu właśnie na temat tego rozwiązania. Które z tych stwierdzeń są prawdziwe? (Zaznacz wszystkie prawidłowe opcje).

- ☐ A. Funkcje języka EL nie rozwiążą tego problemu, ponieważ nie mają dostępu do atrybutów sesji.
- ☐ B. Metody implementującej odpowiednią funkcję języka EL nie można zadeklarować jako składowej statycznej, ponieważ nie miałyby wówczas dostępu do zasięgu sesji.
- ☐ C. Nasza funkcja języka EL może otrzymywać na wejściu parametr typu **java.util.List**, co rozwiązuje problem dostarczenia listy filmów z poziomu wyrażenia EL.
- ☐ D. Być może będziemy zmuszeni opracować znaczniki HTML-a w kodzie Javy z wykorzystaniem funkcji języka EL, co znacznie utrudni konserwację tego rozwiązania.



BAR  
KAWOWY

## Examin próbny — odpowiedzi

- 1 Mamy dany formularz HTML, w którym zastosowano pola wyboru, aby umożliwić użytkownikom zaznaczanie wielu wartości dla parametru nazwanego **hobby**. (Specyfikacja JSP 2.0, punkt 2.2.3).

Które wyrażenia języka EL zwrócą pierwszą wartość parametru **hobby**?  
(Zaznacz wszystkie prawidłowe opcje).

- ☒ A. `${param.hobby}`
- ☐ B. `${paramValue.hobby}` — Odpowiedź B jest niepoprawna, ponieważ nie istnieje zmienna domyślna "paramValue".
- ☒ C. `${paramValues.hobby[0]}`
- ☐ D. `${paramValues.hobby[1]}` — Odpowiedź D jest niepoprawna, ponieważ tablice są indeksowane od zera.
- ☐ E. `${paramValues[hobby][0]}` — Odpowiedzi E i F mają niepoprawną składnię.
- ☐ F. `${paramValues[hobby][1]}`

- 2 Mamy daną aplikację internetową, która przechowuje adres internetowy webmastera w postaci parametru inicjalizacji kontekstu nazwanego **email-webmastera**. (Specyfikacja JSP 2.0, punkty 2.2.3 i 2.3.4).

Które wyrażenia języka EL zwrócą właściwą wartość tego parametru?  
(Zaznacz wszystkie prawidłowe opcje).

- ☐ A. `<a href='mailto:${initParam.email-webmastera}'> napisz do mnie</a>` — Odpowiedź A jest w istocie próbą odjęcia webmastera od poczty elektronicznej.
- ☐ B. `<a href='mailto:${contextParam.email-webmastera}'> napisz do mnie</a>` — Odpowiedź B jest niepoprawna, ponieważ nie istnieje zmienna domyślna contextParam.
- ☒ C. `<a href='mailto:${initParam['email-webmastera']}'> napisz do mnie</a>`
- ☐ D. `<a href='mailto:${contextParam['email-webmastera']}'> napisz do mnie</a>` — Odpowiedź D jest niepoprawna, ponieważ nie istnieje zmienna domyślna contextParam.

3 Mamy daną następującą klasę Javy:

(Specyfikacja JSP 2.0,  
punkt 2.6.3).

```
1. package com.mojafirma;
2. public class MojeFunkcje {
3. public static String witaj(String imie) {
4. return "Witaj "+imie;
5. }
6. }
```

Przedstawiona klasa w istocie odpowiada za obsługę funkcji będącej częścią następującej biblioteki znaczników: `<%@ taglib uri="http://mojafirma.com.tags" prefix="comp" %>`. Który wpis w deskryptorze biblioteki znaczników definiuje tę funkcję użytkownika w sposób umożliwiający jej wykorzystanie w wyrażeniu języka EL?

☐ A. `<taglib>`

```
...
<tag>
 <name>Witaj</name>
 <tag-class>com.mojafirma.MojeFunkcje</tag-class>
 <body-content>JSP</body-content>
</tag>
```

☒ B. `<taglib>`

— Odpowiedź B cechuje się  
prawidłową składnią.

```
...
<function>
 <name>Witaj</name>
 <function-class>com.mojafirma.MojeFunkcje</function-class>
 <function-signature>java.lang.String witaj(java.lang.String)
 </function-signature>
</function>
</taglib>
```

☐ C. `<web-app>`

```
...
<servlet>
 <name>Witaj</name>
 <servlet-class>com.mojafirma.MojeFunkcje</servlet-class>
</servlet>
</web-app>
```

☐ D. `<taglib>`

— Odpowiedź D jest niepoprawna,  
ponieważ sygnatura funkcji jest  
niekompletna.

```
...
<function>
 <name>Witaj</name>
 <function-class>com.mojafirma.MojeFunkcje</function-class>
 <function-signature>witaj(java.lang.String)</function-signature>
</function>
</taglib>
```

## 4 Mamy dany komponent:

(Specyfikacja JSP 2.0, podrozdział 5.1).

1. `package com.example;`
2. `public class Komponent {`
3. `private int wartosc;`
4. `public Komponent() { wartosc = 42; }`
5. `public int getWartosc() { return wartosc; }`
6. `public void setWartosc(int w) { wartosc = w; }`
7. `}`

Zakładając, że nie został jeszcze utworzony żaden egzemplarz klasy **Komponent**, spróbuj określić, które standardowe akcje JSP utworzą nowy egzemplarz tej klasy i umieszczą go w zasięgu żądania. (Zaznacz wszystkie prawidłowe opcje).

- ☐ A. `<jsp:useBean name="mojKomponent" type="com.example.Komponent" />` — Odpowiedź A jest niepoprawna, ponieważ atrybut typu **NIE** jest wykorzystywany podczas tworzenia nowego egzemplarza komponentu i ponieważ nie określono atrybutu zasięgu (jego domyślną wartością jest "page").
- ☐ B. `<jsp:makeBean name="mojKomponent" type="com.example.Komponent" />` — Odpowiedź B jest niepoprawna z tych samych powodów co odpowiedź A oraz dlatego, że **NIE** istnieje znacznik `<jsp:makeBean>`.
- ☒ C. `<jsp:useBean id="mojKomponent" class="com.example.Komponent" scope="request" />`
- ☐ D. `<jsp:makeBean id="mojKomponent" class="com.example.Komponent" scope="request" />` — Odpowiedź D jest niepoprawna, ponieważ **NIE** istnieje znacznik `<jsp:makeBean>`.

## 5 Mamy daną architekturę Model 1, w której pojedyncza strona JSP obsługuje wszystkie funkcje kontrolera — taka strona kontrolera musi mieć możliwość przydzielania żądań do innej strony JSP.

(Specyfikacja JSP 2.0, podrozdział 5.5).

Który z przedstawionych poniżej kodów akcji standardowych wykona taki przydział?

- ☒ A. `<jsp:forward page="widok.jsp" />` — Odpowiedź A jest prawidłowa (str. 1 – 110).
- ☐ B. `<jsp:forward file="widok.jsp" />` — Odpowiedź B jest niepoprawna, ponieważ standardowa akcja `<jsp:forward>` nie zawiera atrybutu `file`.
- ☐ C. `<jsp:dispatch page="widok.jsp" />` — Odpowiedzi C i D są niepoprawne, ponieważ nie istnieje standardowa akcja `<jsp:dispatch>`.
- ☐ D. `<jsp:dispatch file="widok.jsp" />`

6 Mamy dany skryptlet:

(Specyfikacja JSP 2.0, punkt 2.3.4).

```
11. <% java.util.List lista = new java.util.ArrayList();
12. lista.add("a");
13. lista.add("2");
14. lista.add("c");
15. request.setAttribute("lista", lista);
16. request.setAttribute("indeksListy", "1");
17. %>
18. <!-- tutaj wstaw odpowiednie wyrażenie --%>
```

Które z poniższych wyrażen po wstawieniu w wierszu 18. przedstawionego kodu będzie poprawne i zwróci wartość **c**? (Zaznacz wszystkie prawidłowe opcje).

- ☐ A. `${lista.2}` — Odpowiedzi A i C są niepoprawne, ponieważ operator kropki nie może być wykorzystywany z typem prostym.
- ☒ B. `${lista[2]}`
- ☐ C. `${lista.indeksListy+1}`
- ☒ D. `${lista[indeksListy+1]}`
- ☐ E. `${lista['indeksListy' + 1]}` — Odpowiedź E jest niepoprawna, ponieważ język EL próbuje dokonać konwersji wyrażenia 'indeksListy' na wartość typu Long, co nie jest możliwe.
- ☒ F. `${lista[lista[indeksListy]]}`

7 Które zdania na temat stosowanego w języku EL operatora kropki (.) i operatora [] są prawdziwe? (Zaznacz wszystkie prawidłowe opcje).

(Specyfikacja JSP 2.0, str. 1 – 69).

- ☐ A. Wyrażenie `${foo.bar}` jest równoważne wyrażeniu `${foo[bar]}`. — Odpowiedź A jest niepoprawna, ponieważ należałoby użyć zapisu `foo["bar"]`.
- ☒ B. Wyrażenie `${foo.bar}` jest równoważne wyrażeniu `${foo["bar"]}`.
- ☒ C. Składnia wyrażenia `${foo["5"]}` jest poprawna, jeśli **foo** jest mapą.
- ☐ D. Wyrażenie `${header.User-Agent}` jest równoważne wyrażeniu `${header[User-Agent]}`.
- ☐ E. Wyrażenie `${header.User-Agent}` jest równoważne wyrażeniu `${header["User-Agent"]}`. — Odpowiedzi D i E są niepoprawne ze względu na myślnik w nazwie User-Agent. Jedynym właściwym zapisem jest w tym przypadku `header["User-Agent"]`.
- ☒ F. Składnia wyrażenia `${foo[5]}` jest poprawna, jeśli **foo** jest tablicą lub egzemplarzem klasy **List**.

- 8 Mamy daną stronę JSP z następującym wierszem: (Specyfikacja JSP 2.0, str. 1 – 71).

`${101 % 10}`

Co zostanie wyświetlone?

- ☒ A. 1
- ☐ B. 10
- ☐ C. 1001
- ☐ D. 101 % 10
- ☐ E. {101 % 10}

— Odpowiedź A jest prawidłowa. Operator modulo zwraca resztę z operacji dzielenia.

- 9 Mamy dane następujące wyrażenia EL:

(Specyfikacja JSP 2.0, str. 1 – 67 oraz 1 – 79).

- 10. `${param.pierwszeimie}`
- 11. `${param.drugieimie}`
- 12. `${param.nazwisko}`
- 13. `${paramValues.nazwisko[0]}`

Który z poniższych wyników reprezentuje dane wyjściowe wygenerowane przez przedstawiony fragment kodu strony JSP w przypadku przekazania następującego łańcucha zapytania:

`?pierwszeimie=John&nazwisko=Doe?`

- ☐ A. John Doe
- ☒ B. John Doe Doe
- ☐ C. John null Doe
- ☐ D. John null Doe Doe
- ☐ E. Zostanie rzucony wyjątek `NullPointerException`.

— Odpowiedź A jest niepoprawna, ponieważ wiersz 13. także wyświetli nazwisko użytkownika.

— Odpowiedzi C i D są niepoprawne, ponieważ w wyniku wykonania wiersza 11. nic nie zostanie wyświetlone (na stronie wynikowej nie pojawi się wartość "null").

- 10 Które z poniższych wyrażeń zawierają prawidłowo użyte zmienne domyślne języka EL? (Zaznacz wszystkie prawidłowe opcje). (Specyfikacja JSP 2.0, str. 1 – 66).

- ☐ A. `${cookies.foo}`
- ☒ B. `${initParam.foo}`
- ☐ C. `${pageContext.foo}`
- ☒ D. `${requestScope.foo}`
- ☒ E. `${header["User-Agent"]}`
- ☐ F. `${requestDispatcher.foo}`
- ☒ G. `${pageContext.request.requestURL}`

— Odpowiedź A jest niepoprawna, ponieważ dostępną zmienną nazwano "cookie" (nie "cookies").

— Odpowiedź C jest niepoprawna, ponieważ zmienna `pageContext` nie tylko NIE jest mapą, ale także nie zawiera właściwości "foo".

— Odpowiedź F jest niepoprawna, ponieważ `requestDispatcher` NIE jest obiektem domyślnym.

- 11** Które zdania na temat standardowej akcji `<jsp:useBean>` są prawdziwe? (Zaznacz wszystkie prawidłowe opcje). (Specyfikacja JSP 2.0, str. 1 – 103 oraz 1 – 104).
- ☐ A. Atrybut **id** jest opcjonalny. — Odpowiedź A jest niepoprawna, ponieważ atrybut **id** jest wymagany.
  - ☐ B. Atrybut **scope** jest wymagany. — Odpowiedzi B i C są niepoprawne, ponieważ atrybut **scope** jest opcjonalny, a jego wartością domyślną jest `"page"`.
  - ☐ C. Atrybut **scope** jest opcjonalny, a jego domyślną wartością jest `request`.
  - ☒ D. W standardowej akcji można zdefiniować albo atrybut **class**, albo atrybut **type**, ale wymagane jest określenie przynajmniej jednego z nich.
  - ☒ E. Istnieje możliwość zdefiniowania w standardowej akcji zarówno atrybutu **class**, jak i atrybutu **type**, nawet jeśli ich wartości NIE są takie same.

- 12** Jak dołączyłbyś dynamiczną treść do swojej strony JSP w sposób zbliżony do działania mechanizmu dołączania po stronie serwera (ang. Server-Side Include — SSI)? (Specyfikacja JSP 2.0, podrozdział 5.4).
- (Zaznacz wszystkie prawidłowe opcje).
- ☐ A. `<%@ include file="/segmenty/stopka.jspf" %>` — Odpowiedź A jest niepoprawna, ponieważ wykorzystano w niej dyrektywę `include`, której należy używać wyłącznie do operacji dołączania statycznego realizowanego w czasie tłumaczenia kodu JSP.
  - ☐ B. `<jsp:forward page="/segmenty/stopka.jspf" />`
  - ☒ C. `<jsp:include page="/segmenty/stopka.jspf" />`
  - ☐ D. `RequestDispatcher dispatcher = request.getRequestDispatcher("/segmenty/stopka.jspf"); dispatcher.include(request, response);` — Odpowiedź D byłaby prawidłowa, gdyby użyty fragment kodu był skryptem — funkcjonalnie znaczenie tej opcji jest takie samo jak znaczenie opcji C, ale zastosowana składnia może być wykorzystywana wyłącznie na poziomie serwetów.

- 13** Której ze standardowych akcji JSP można użyć w stronie HTML z bogatym układem graficznym do zaimportowania pliku obrazu do danej strony JSP? (Specyfikacja JSP 2.0, podrozdział 5.4).
- ☐ A. `<jsp:image page="logo.png" />` — Odpowiedzi A i B są niepoprawne, ponieważ nie istnieje standardowa akcja `<jsp:image>`.
  - ☐ B. `<jsp:image file="logo.png" />` — Odpowiedź C jest niepoprawna nie dlatego, że zastosowana składnia standardowej akcji `include` jest niewłaściwa, tylko dlatego, że importowanie danych binarnych z pliku obrazu do wnętrza strony JSP jest działaniem pozbawionym sensu.
  - ☐ C. `<jsp:include page="logo.png" />`
  - ☐ D. `<jsp:include file="logo.png" />` — Odpowiedź D jest niepoprawna, ponieważ standardowa akcja `include` nie zawiera atrybutu `file`.
  - ☒ E. NIE MOŻNA tego zrobić wyłącznie w oparciu o standardowe akcje JSP.

To pytanie wbrew pozorom jest dosyć łatwe, ponieważ NIE istnieje możliwość importowania zawartości żadnego pliku binarnego do strony JSP, która z założenia ma generować odpowiedź w postaci kodu HTML.

**14** Mamy daną klasę Javy:

(Specyfikacja JSP 2.0,  
podrozdział 2.6).

```
1. package com.example;
2. public class MojeFunkcje {
3. public static String powtorz(int x, String str) {
4. // ciało metody
5. }
6. }
```

i następujący kod JSP:

```
1. <%@ taglib uri="/WEB-INF/mojefunkcje" prefix="moje" %>
2. <!-- tutaj wstaw odpowiedni kod --%>
```

Które z poniższych wyrażeń po wstawieniu w wierszu 2. kodu JSP będzie prawidłowym wywołaniem funkcji języka EL?

- ☐ A. `${powtorz(2, "420")}`
- ☐ B. `${powtorz("2", "420")}`
- ☐ C. `${moje:powtorz(2, "420")}`
- ☐ D. `${moje:powtorz("2", "420")}`
- ☒ E. Prawidłowego wywołania NIE MOŻNA określić.

— Odpowiedź E jest prawidłowa.  
Niezbędne informacje na temat  
odzworowania z deskryptora TLD  
NIE są znane.

**15** Mamy dany komponent:

(Specyfikacja JSP 2.0, str. 1 – 68).

```
10. public class MojKomponent {
11. private java.util.Map parametry;
12. private java.util.List obiekty;
13. private String nazwa;
14. public java.util.Map getParametry() { return parametry; }
15. public String getNazwa() { return nazwa; }
16. public java.util.List getObiekty() { return obiekty; }
17. }
```

Które z poniższych wyrażeń spowodują błędy (zakładając, że mamy dostęp do atrybutu nazwanego **mojkomponent**, którego typem jest zdefiniowana przed chwilą klasa **MojKomponent**)? (Zaznacz wszystkie prawidłowe opcje).

- ☐ A. `${mojkomponent.nazwa}`
- ☐ B. `${mojkomponent["nazwa"]}`
- ☒ C. `${mojkomponent.obiekty.a}`
- ☐ D. `${mojkomponent["parametry"].a}`
- ☐ E. `${mojkomponent.parametry["a"]}`
- ☒ F. `${mojkomponent["obiekty"].a}`

— Odpowiedzi C i F spowodują błędy. "a"  
NIE jest właściwością typu List, a ponieważ  
"obiekty" NIE jest mapą, wyszukiwanie  
odpowiednich wartości nie przyniesie rezultatu  
(w przeciwieństwie do odpowiedzi D i E).

16 Mamy daną stronę JSP:

(Specyfikacja JSP 2.0,  
str. 1 – 66 oraz 1 – 73).

1. Użytkownik poprawnie się zalogował lub wylogował:
2. `${param.zalogowany or param.wylogowany}`.

Gdyby żądanie zawierało łańcuch zapytania „wylogowany=true”, jaki byłby wyświetlony na stronie wynik powyższego wyrażenia?

- ☐ A. Użytkownik poprawnie się zalogował lub wylogował: false.
- ☒ B. Użytkownik poprawnie się zalogował lub wylogował: true.
- ☐ C. Użytkownik poprawnie się zalogował lub wylogował: `${param.zalogowany or param.wylogowany}`.
- ☐ D. Użytkownik poprawnie się zalogował lub wylogował: param.zalogowany or param.wylogowany.
- ☐ E. Użytkownik poprawnie się zalogował lub wylogował: or true.

— Odpowiedź B jest prawidłowa, ponieważ w przedstawionym wyrażeniu języka EL użyto operatora „or”, który zwróci wartość true, jeśli którykolwiek z pary parametrów zalogowany bądź wylogowany będzie miał wartość true.

17 Które zdania na temat dostępnych w języku EL operatorów dostępu są prawdziwe? (Zaznacz wszystkie prawidłowe opcje).

(Specyfikacja JSP 2.0,  
str. 1 – 69).

- ☒ A. Wszędzie tam, gdzie można stosować operator kropki (.), równie dobrze można używać operatora [].
- ☐ B. Wszędzie tam, gdzie można stosować operator [], równie dobrze można używać operatora kropki (.).
- ☒ C. Jeśli operator kropki (.) jest używany do uzyskiwania dostępu do nieistniejącej właściwości komponentu, zostanie zwrócony wyjątek czasu wykonywania.
- ☐ D. Istnieją pewne sytuacje, w których musimy użyć operatora kropki (.), ale są też sytuacje, w których niezbędne jest zastosowanie operatora [].

— Odpowiedź B jest niepoprawna, ponieważ tylko operator [] umożliwia uzyskanie dostępu do (a) egzemplarzy klasy List i tablic oraz (b) map, których klucze nie mają właściwego formatu.

— Odpowiedź D jest niepoprawna, ponieważ operator kropki zawsze może być przekształcony w odpowiedni operator [].

18 W kodzie naszej strony JSP użyliśmy następującego fragmentu:

(Specyfikacja JSP 2.0, podrozdział 5.4).

```
<jsp:include page="/jspf/naglowek.html"/>
```

Nasza strona JSP jest częścią aplikacji z katalogiem głównym kontekstu myapp.

Wiedząc, że najwyższy poziom struktury katalogów tej aplikacji tworzy katalog myapp, określ, która z wymienionych poniżej ścieżek do pliku naglowek.html jest prawidłowa?

- ☐ A. /naglowek.html
- ☐ B. /jspf/naglowek.html
- ☒ C. /myapp/jspf/naglowek.html
- ☐ D. /includes/jspf/naglowek.html

— Zdefiniowana w atrybucie page standardowej akcji `<jsp:include>` wartość „/jspf/naglowek.html” jest ścieżką względną wobec bieżącej aplikacji internetowej, zatem użycie znaku ukośnika („/”) w praktyce oznacza: „rozpocznij od najwyższego poziomu tej aplikacji”.

- 19 Internetowy sprzedawca biżuterii chce dostosowywać katalog oferowanych produktów do preferencji zalogowanych użytkowników. Chce proponować swoim klientom oferty specjalne w miesiącach ich urodzin. Oferty specjalne sprzedawcy są składowane w strukturze **Map<String, Special[]>** identyfikowanej w zasięgu aplikacji jako **specjalne** i aktualizowanej codziennie.

Aplikacja obejmuje komponent składowany w formie atrybutu zasięgu sesji i nazwany **daneUzytkownika**. W wyniku wywołania metody **getDataUrodzenia().getMiesiac()** tego komponentu otrzymujemy miesiąc urodzin danego użytkownika.

Który z poniższych fragmentów kodu umożliwia prawidłowe uzyskanie odpowiedniej oferty specjalnej?

- ☐ A. `${applicationScope[daneUzytkownika.dataUrodzenia.miesiac.specjalne]}`  
☒ B. `${applicationScope.specjalne[daneUzytkownika.dataUrodzenia.miesiac]}`  
☐ C. `${applicationScope["specjalne"].daneUzytkownika.dataUrodzenia.miesiac}`  
☐ D. `${applicationScope["daneUzytkownika.dataUrodzenia.miesiac"].specjalne}`

(Specyfikacja JSP 2.0, punkt 2.3.4).

— Tylko fragment przedstawiony w odpowiedzi B prawidłowo uzyskuje naszą strukturę `Map<String, Special[]>` z zasięgu aplikacji. W tym samym fragmencie próbujemy następnie uzyskać wartość miesiąca na podstawie daty urodzenia użytkownika i wykorzystać tę wartość w roli klucza identyfikującego strukturę `Special[]` w ramach tej mapy. Nasze wyrażenie zwraca właśnie strukturę `Special[]`, zakładając, że zostanie odnaleziona w przeszukiwanej mapie. Przedstawione wyrażenie można by wykorzystać w znaczniku `forEach` do iteracyjnego przeszukiwania zwróconych ofert specjalnych.

- 20 Aplikacja internetowa wykorzystywana przez popularną wypożyczalnię filmów składa się w formie atrybutu sesji strukturę **List<Movie>** reprezentującą filmy wybrane przez użytkownika. Losowo wybrany zwiastun filmu z tej listy musi być prezentowany na stronie głównej użytkownika za każdym razem, gdy odwiedza on witrynę wypożyczalni.

(Specyfikacja JSP 2.0, podrozdział 2.6).

Kierownictwo firmy uważa, że podobną funkcję w niedalekiej przyszłości należałoby wprowadzić także na pozostałych stronach prezentujących listy dostępnych filmów. Za obsługę strumienia wideo odpowiadają zwykle elementy HTML-a, zatem samo dodanie odpowiednich konstrukcji nie jest dużo trudniejsze (mimo bardziej złożonych znaczników) od wstawiania obrazów.

Zespół programistów potrzebuje rozwiązania gwarantującego jednocześnie elastyczność i łatwą konserwację. Jedną z możliwości jest utworzenie odpowiedniej funkcji języka EL. Poniższe stwierdzenia pochodzą z dyskusji prowadzonej przez członków wspomnianego zespołu właśnie na temat tego rozwiązania. Które z tych stwierdzeń są prawdziwe? (Zaznacz wszystkie prawidłowe opcje).

- ☐ A. Funkcje języka EL nie rozwiążą tego problemu, ponieważ nie mają dostępu do atrybutów sesji.  
☐ B. Metody implementującej odpowiednią funkcję języka EL nie można zadeklarować jako składowej statycznej, ponieważ nie miałyby wówczas dostępu do zasięgu sesji.  
☒ C. Nasza funkcja języka EL może otrzymywać na wejściu parametr typu **java.util.List**, co rozwiązuje problem dostarczenia listy filmów z poziomu wyrażenia EL.  
☒ D. Być może będziemy zmuszeni opracować znaczniki HTML-a w kodzie Javy z wykorzystaniem funkcji języka EL, co znacznie utrudni konserwację tego rozwiązania.

— Odpowiedź A jest niepoprawna, ponieważ listę filmów można przekazać w formie parametru funkcji.

— Odpowiedź B jest niepoprawna, ponieważ wszystkie metody implementujące funkcje języka EL muszą być deklarowane jako publiczne i statyczne.

— Odpowiedź D jest prawidłowa, ponieważ właśnie przytoczony argument przemawia przeciw stosowaniu funkcji języka EL. Ostatecznie zespół decyduje się na użycie pliku znacznika w roli właściwego rozwiązania uzupełnionego o funkcję EL otrzymującą na wejściu kolekcję i zwracającą liczbę losową zależną od rozmiaru tej kolekcji.

— Odpowiedź C jest prawidłowa, ponieważ przekazanie listy na wejściu funkcji jest możliwe. Takie rozwiązanie zapewnia większą elastyczność niż nakładanie na funkcję języka EL obowiązku obsługi zasięgu sesji (jak w odpowiedziach A i B).

## 9. Stosowanie biblioteki JSTL

# Potęga znaczników niestandardowych

Czy to oznacza, że kiedy używałem skryptletów do działań, których nie da się zrealizować za pomocą wyrażeń języka EL i akcji standardowych, tak naprawdę *mogłem* używać znaczników biblioteki JSTL?



**Czasami potrzebujemy czegoś więcej niż tylko języka wyrażeń (EL) i akcji standardowych.** Co będzie, jeśli zechcesz użyć pętli do przeszukania danych składowanych w tablicy i wyświetlenia po jednym elemencie w każdym wierszu generowanej dynamicznie tabeli HTML? Oczywiście zdajesz sobie sprawę z tego, że można w tym celu błyskawicznie skonstruować odpowiednią pętlę w skryptlecie. Z drugiej strony, naszym celem jest rezygnacja z elementów skryptowych. To żaden problem. Kiedy możliwości samego języka EL i standardowych akcji nie wystarczą, możesz użyć *znaczników niestandardowych*. Ich stosowanie w kodzie stron JSP jest równie proste jak stosowanie akcji standardowych. Co więcej, okazuje się, że ktoś już opracował cały zbiór najczęściej wykorzystywanych znaczników i upakował je w tzw. standardowej bibliotece znaczników JSP (ang. *JSP Standard Tag Library — JSTL*). W *niniejszym* rozdziale nauczysz się używać znaczników niestandardowych, natomiast z następnego rozdziału dowiesz się, jak można je tworzyć samemu.



### **Budowanie stron JSP w oparciu o biblioteki znaczników**

- 9.1.** Opisz składnię i semantykę dyrektywy `tag1 i b` dla standardowej biblioteki znaczników i dla biblioteki plików znaczników.
- 9.2.** Mając dany cel projektowy, opracuj strukturę znacznika niestandardowego, który pozwoli ten cel osiągnąć.
- 9.3.** Zidentyfikuj składnię znacznika i opisz semantykę akcji dla następujących znaczników należących do wersji 1.1 biblioteki JSTL (ang. *JSP Standard Tag Library*): (a) znaczniki podstawowe: `out`, `set`, `remove` i `catch`, (b) znaczniki warunkowe: `if`, `choose`, `when` i `otherwise`, (c) znaczniki iteracji: `forEach` oraz (d) znaczniki związane z obsługą adresów URL: `url`.

### **Uwagi wyjaśniające:**

*Wszystkie wymienione obok cele zostaną dogłębnie omówione jeszcze w tym rozdziale, chociaż część tych zagadnień raz jeszcze przeanalizujemy w kolejnym rozdziale („Tworzenie znaczników niestandardowych”).*

#### **Instalacja biblioteki JSTL 1.1**

Biblioteka JSTL 1.1 NIE jest częścią specyfikacji JSP 2.0! Samo posiadanie dostępu do interfejsów API serwletów i JSP wcale nie oznacza, że masz dostęp do biblioteki JSTL.

Zanim będziesz mógł korzystać z biblioteki JSTL, musisz umieścić pliki *jstl.jar* i *standard.jar* w katalogu *WEB-INF/lib* swojej aplikacji internetowej. Oznacza to, że każda Twoja aplikacja będzie musiała zawierać w swojej strukturze katalogów własną kopię tych plików.

Biblioteka JSTL jest dołączana do serwera (kontenera) Tomcat 5 w ramach jego przykładowych aplikacji internetowych, zatem musisz jedynie skopiować odpowiednie pliki z katalogu już istniejącej aplikacji do własnego katalogu *WEB-INF/lib*.

Skopiuj następujące pliki biblioteki JSTL z przykładów kontenera Tomcat:

***webapps/jsp-examples/WEB-INF/lib/jstl.jar***

***webapps/jsp-examples/WEB-INF/lib/standard.jar***

i umieść je w katalogu *WEB-INF/lib* swojej aplikacji internetowej.

Musi istnieć sposób iteracyjnego przeszukiwania struktur danych w JSP... bez stosowania elementów skryptowych. Chcę wyświetlać po jednym elemencie w każdym kolejnym wierszu tabeli...



## Możliwości języka EL i akcji standardowych są ograniczone

Co będzie, kiedy zderzysz się z twardą, ceglana ścianą? Możesz oczywiście wrócić do elementów skryptowych, ale z pewnością wiesz, że to nie rozwiązuje problemu.

Programiści oczekują zwykle *znacznie* więcej standardowych akcji lub — jeszcze częściej — możliwości tworzenia *własnych* akcji.

Właśnie w tym celu umożliwiono tworzenie i wykorzystywanie w kodzie JSP *znaczników niestandardowych*. Zamiast mówić `<jsp:setProperty>`, możesz powiedzieć np. `<moje:zróbCoŚNiestandardowego>`. Okazuje się, że jest to możliwe.

Tworzenie kodu odpowiedzialnego za obsługę znacznika niestandardowego nie jest jednak takie proste. Co prawda dla przeciętnego twórcy stron JSP stosowanie niestandardowych znaczników jest zdecydowanie prostsze niż korzystanie z elementów skryptowych, ale dla programisty Javy budowa *kodu obsługującego* taki znacznik (odpowiedniego kodu Javy, który będzie wywoływany w momencie znalezienia tego znacznika) jest już dużo trudniejsze.

Na szczęście istnieje standardowa biblioteka niestandardowych znaczników znana jako **standardowa biblioteka znaczników JSP** (ang. **JSP Standard Tag Library** — JSTL 1.1). Wiedząc, że nasze strony JSP nigdy nie powinny odpowiadać za realizację logiki biznesowej, powinniśmy wykorzystać możliwości oferowane przez bibliotekę JSTL (w połączeniu z językiem wyrażen EL) do wyeliminowania wszelkich elementów skryptowych. Budowa aplikacji w oparciu o wymienione technologie nie oznacza jednak, że w niektórych przypadkach nie będziemy musieli skorzystać np. z biblioteki niestandardowych znaczników opracowanej specjalnie dla naszej firmy.

W *niniejszym* rozdziale omówimy techniki stosowania nie tylko podstawowych znaczników biblioteki JSTL, ale także niestandardowych znaczników pochodzących z innych bibliotek. Z lektury *następnego* rozdziału dowiesz się, jak można budować klasy obsługujące wywołania niestandardowych znaczników — dzięki zdobytej wiedzy będziesz potrafił opracowywać własne znaczniki tego typu.

# Sprawa znikającego kodu HTML (i potrącenia pensji)

Na stronie 412 analizowaliśmy przykład wyrażenia języka EL przekazującego nieprzetworzony łańcuch treści bezpośrednio do strumienia odpowiedzi:

```
<div class='tipBox'>
 Porada dnia:

 ${pageContent.currentTip}
</div>
```

Pamiętasz to? Znaczniki `<b>`/`</b>` nie były prezentowane w formie tekstu na stronie, tylko zostały poddane wizualizacji do postaci pustej przestrzeni „wyróżnionej” pogrubieniem.

Co mamy

Czego chcemy

```
<div class='tipBox'>
 Porada dnia:

 Znaczniki pogrubiają tekst!
</div>
```

```
<div class='tipBox'>
 Porada dnia:

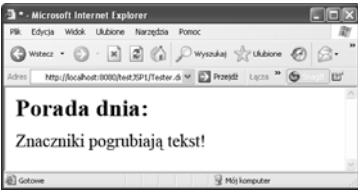
 Znaczniki pogrubiają tekst!
</div>
```

Ten fragment ostatecznie przybiera postać niewidocznej, pogrubionej, pustej przestrzeni.

`&lt;` jest wyświetlany jako `<`; `&gt;` jest wyświetlany jako `>`

Wizualizowane jako

Wizualizowane jako



Potrzebujemy więc rozwiązania konwertującego te nawiasy ostre na coś, co przeglądarka internetowa przedstawi właśnie jako nawiasy ostre. Można ten cel osiągnąć na dwa sposoby. W obu przypadkach musimy się posłużyć statyczną metodą Javy konwertującą znaki specjalne HTML-a na odpowiednie sekwencje:

## Użycie funkcji języka EL

```
<div class='tipBox'>
 Porada dnia:

 ${fn:convEntity(pageContent.currentTip)}
</div>
```

## Użycie metody pomocniczej Javy

```
<div class='tipBox'>
 Porada dnia:

 ${pageContent.convertedCurrentTip}
</div>
```

Oto nasza metoda pomocnicza niezbędna do działania tego rozwiązania.

```
public String getConvertedCurrentTip() {
 return HTML.convEntity(getCurrentTip());
}
```

## Jest lepsze rozwiązanie: użycie znacznika `<c:out>`

Niezależnie od podejścia, na które się zdecydujesz, w pierwszej chwili określenie faktycznego znaczenia tych mechanizmów nie jest łatwe... a możesz przecież stanąć przed koniecznością napisania podobnej metody pomocniczej dla wszystkich swoich serwetów. Na szczęście istnieje lepsze rozwiązanie. Do opisywanego zadania wprost idealnie nadaje się znacznik `<c:out>`. Oto, jak przebiega konwersja:

### Możesz wprost zadeklarować konwersję elementów XML-a

Jeśli wiesz lub podejrzewasz, że będziesz dysponował jakimiś elementami XML-a wymagającymi wyświetlenia (zamiast tradycyjnej wizualizacji w przeglądarce), możesz użyć atrybutu `escapeXml` znacznika `<c:out>`. Przypisanie temu atrybutowi wartości `true` powoduje, że wszelkie elementy XML-a (w tym nawiasy ostre) zostaną przekonwertowane na coś, co umożliwi ich prezentację w oknie przeglądarki:

```
<div class='tipBox'>
 Porada dnia:

 <c:out value='${pageContent.currentTip}' escapeXml='true' />
</div>
```

Twój kod HTML jest traktowany jak kod XHTML, który z kolei jest traktowany jak XML... takie rozwiązanie ma więc wpływ także na znaki HTML-a.

### Możesz wprost zadeklarować BRAK konwersji elementów XML-a

W pewnych sytuacjach jesteśmy zainteresowani czymś wręcz przeciwnym. Być może pracujesz nad stroną otrzymującą jakąś treść, którą należy wyświetlić zgodnie z regułami formatowania danych w języku HTML. W takim przypadku należy wyłączyć konwersję elementów XML-a:

```
<div class='tipBox'>
 Porada dnia:

 <c:out value='${pageContent.currentTip}' escapeXml='false' />
</div>
```

Ta konstrukcja jest równoważna pierwszej, nieudanej wersji... wszelkie znaczniki HTML-a zostaną przetworzone i wizualizowane, a nie wyświetlone w niezmienionej postaci.

### Konwersja jest realizowana domyślnie

Domyślną wartością atrybutu `escapeXml` jest `true`, zatem jego stosowanie nie jest konieczne. Znacznik `<c:out>` bez tego atrybutu jest równoważny znacznikowi `<c:out>` przypisującemu atrybutowi `escapeXml` wartość `true`:

```
<div class='tipBox'>
 Porada dnia:

 <c:out value='${pageContent.currentTip}' />
</div>
```

To ma takie samo znaczenie jak to.

Nie ma  
niemądrych pytań

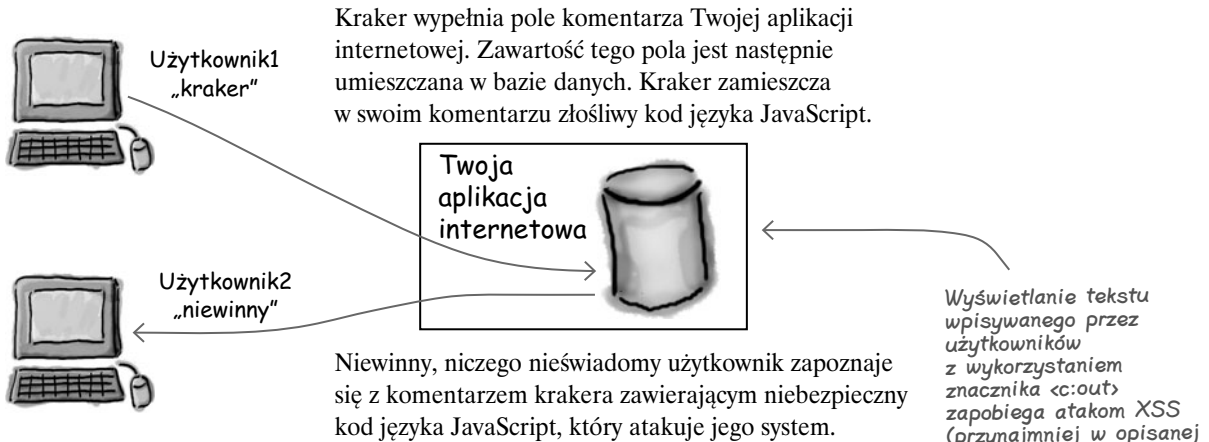
**P:** Które znaki specjalne HTML-a są konwertowane?

**U:** Okazuje się, że opisywana konwersja jest dość prosta. Zaledwie pięć znaków wymaga stosowania sekwencji ucieczki: `<`, `>`, `&` oraz apostrof (`'`) i cudzysłów (`"`). Wszystkie te symbole są konwertowane na odpowiednie kody języka HTML. Na przykład znak `<` jest zastępowany sekwencją `&lt;`, a znak `&` jest zamieniany na `&amp;`.

Znak	Kod znaku
<code>&lt;</code>	<code>&amp;lt;</code>
<code>&gt;</code>	<code>&amp;gt;</code>
<code>&amp;</code>	<code>&amp;amp;</code>
<code>'</code>	<code>&amp;#039;</code>
<code>"</code>	<code>&amp;#034;</code>

**P:** Miesiąc temu moja firma zatrudniła konsultanta, który miał dokonać audytu naszej aplikacji internetowej. Konsultant zwrócił uwagę na fakt stosowania języka wyrażeń (EL) wszędzie tam, gdzie są wyświetlane łańcuchy wpisane przez użytkownika. Stwierdził, że takie podejście zagraża bezpieczeństwu naszej aplikacji i zasugerował użycie znacznika `<c:out>`. Co ten znacznik zmienia?

**U:** Konsultant miał rację. Zagrożenie, o którym mówił, to ataki typu XSS (od ang. *cross-site scripting*). Takie ataki polegają na przekazywaniu złośliwego kodu do przeglądarek użytkowników za pośrednictwem źle zabezpieczonych aplikacji internetowych.



**P:** Co będzie, jeśli wyrażenie języka EL będzie równe null?

**U:** Dobre pytanie. Jak wiemy, wyrażenie języka EL w formie `${maWartośćNull}` generuje w danych wyjściowych odpowiedzi łańcuch pusty. Podobnie jest w przypadku znacznika `<c:out value="${maWartośćNull}" />`.

To jednak nie koniec opowieści o znaczniku `<c:out>`. Okazuje się, że opisywany znacznik jest na tyle „inteligentny”, że rozpoznaje wartość `null` i może podjąć specjalne działania. Tymi działaniami jest użycie wartości domyślnej...

## Wartości null są wizualizowane jako pusty tekst

Przypuśćmy, że mamy stronę witającą użytkownika słowami „Witaj <użytkownik>”. Ponieważ jednak ostatnio niewielu użytkowników decyduje się na logowanie, prezentowany komunikat wygląda dość dziwnie.

### Wyrażenie EL niczego nie wyświetla, jeśli użytkownik ma wartość null

```
Witaj ${uzytkownik}.
```

**Wizualizowane jako**

```
Witaj .
```

### Także znacznik wyrażenia JSP niczego nie wyświetla, jeśli użytkownik ma wartość null

```
Witaj <%= uzytkownik %>.
```

**Wizualizowane jako**

```
Witaj .
```

Ponieważ wyrażenia `${uzytkownik}` i `<%= uzytkownik %>` mają wartość null, otrzymujemy pustą przestrzeń pomiędzy „Witaj” a kropką. Wygląda to dość dziwnie.

## Wartość domyślną można ustawić za pośrednictwem atrybutu default

Przypuśćmy, że chcemy, aby niezalogowani, anonimowi użytkownicy widzieli w swoich przeglądarkach komunikat: „Witaj gościu”. Okazuje się, że znacznik `<c:out>` wprost doskonale nadaje się do tej roli. Wystarczy dodać atrybut **default** i określić wartość, która ma być wyświetlana w razie wykrycia wartości **null** we właściwym wyrażeniu.

### Znacznik `<c:out>` obsługuje atrybut default

```
Witaj <c:out value='${uzytkownik}' default='gościu' />.
```

**Wizualizowane jako**

```
Witaj gościu.
```

Ta wartość zostanie użyta w danych wynikowych, jeśli wartość atrybutu `value` będzie równa null.

Tym razem użyto wartości domyślnej... doskonale.

### Można też użyć rozwiązania alternatywnego:

```
Witaj <c:out value='${uzytkownik}'>gościu</c:out>.
```

# Pętle bez skryptów

Wyobraź sobie, że potrzebujesz czegoś, co będzie przeszukiwało w pętli kolekcję danych (np. tablicę pozycji katalogowych), wyciągało z tej kolekcji po jednym elemencie w każdej iteracji i wyświetlało ten element w generowanym dynamicznie wierszu tabeli. Najprawdopodobniej nie możesz zakodować na stałe kompletnej tabeli — projektując swoją stronę JSP, nie masz przecież pojęcia, ile wierszy będzie wyświetlanych w czasie jej wykonywania, nie znasz także wartości składowanych w kolekcji danych. Rozwiązaniem tego problemu jest znacznik <c:forEach>. Stosowanie tego znacznika nie wymaga szczególnie zaawansowanej wiedzy na temat tabel HTML, ale na wszelki wypadek dołączyliśmy odpowiednie notatki dla osób, które wcześniej nie miały do czynienia z tymi zagadnieniami.

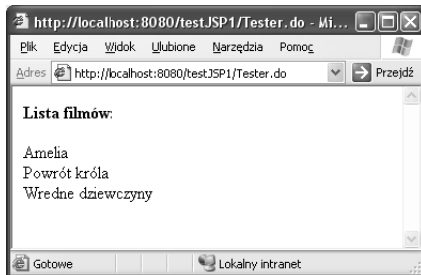
Nawiasem mówiąc, na egzaminie *powinieneś* wiedzieć, jak należy używać znacznika <c:forEach> właśnie z tabelami HTML.

## Kod serwletu

```
...
String[] listaFilmow = {"Amelia", "Powrót króla", "Wredne dziewczyny"};
request.setAttribute("listaFilmow", listaFilmow);
...
```

*Tworzymy tablicę typu String[] z tytułami filmów i ustawiamy ją w roli atrybutu żądania.*

## Co chcemy otrzymać



## W kodzie strony JSP z elementami skryptowymi

```
<table>
<% String[] elementy = (String[]) request.getAttribute("listaFilmow");
 String zmienna=null;
 for (int i = 0; i < elementy.length; i++) {
 zmienna = elementy[i];
 %>
<tr><td><%= zmienna %></td></tr>
<% } %>
</table>
```

## < c:forEach >

Dostępny w bibliotece JSTL znacznik `<c:forEach>` doskonale nadaje się do tego typu działań — oferuje prosty sposób iteracyjnego przeszukiwania tablic i kolekcji danych.

### Kod JSP

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<html><body>
 Lista filmów:


```

(Dyrektywę taglib szczegółowo omówimy w dalszej części rozdziału).

```
<table>
 <c:forEach var="film" items="${listaFilmow}" >
 <tr>
 <td>${film}</td>
 </tr>
 </c:forEach>
</table>
</body></html>
```

Przeszukujemy w pętli całą tablicę (atrybut „listaFilmow”) i wyświetlamy każdy element w nowym wierszu tabeli. (Nasza tabela składa się tylko z jednej kolumny).

### Błyskawiczne przypomnienie tabel HTML

Znacznik `<tr>` oznacza wiersz tabeli (ang. Table Row).  
Znacznik `<td>` oznacza dane tabeli (ang. Table Data).

```
<table>
 <tr>
 <td>dane dla tej komórki</td> <td>dane dla tej komórki</td> <td>dane dla tej komórki</td> </tr>
 <tr>
 <td>dane dla tej komórki</td> <td>dane dla tej komórki</td> <td>dane dla tej komórki</td> </tr>
 <tr>
 <td>dane dla tej komórki</td> <td>dane dla tej komórki</td> <td>dane dla tej komórki</td> </tr>
</table>
```

Tabele języka HTML są dosyć proste. Składają się z *komórek* tworzących *wiersze* i *kolumny*; właściwe dane są umieszczane właśnie w komórkach. Cały problem polega na określeniu, ile wierszy i kolumn ma się znajdować w tworzonej tabeli.

Wiersze definiuje się za pomocą znacznika `<tr>` (*Table Row*), natomiast kolumny definiujemy za pomocą znacznika `<td>` (*Table Data*). Liczba wierszy tabeli zależy od liczby użytych znaczników `<tr>`, zaś liczba kolumn zależy od liczby znaczników `<td>` użytych wewnątrz znaczników `<tr></tr>`.

**Wyświetlane dane mogą się znajdować wyłącznie wewnątrz znaczników `<td>` `</td>`!**

## Dekonstrukcja znacznika `<c:forEach>`

Znacznik `<c:forEach>` jest odwzorowywany na pętlę `for` — znacznik ten powtarza swoje ciało dla każdego elementu w danej kolekcji (w tym przypadku używamy słowa „kolekcja” w znaczeniu tablicy, egzemplarza klasy `Collection`, mapy albo łańcucha z przecinkami w roli separatorów kolejnych elementów).

Kluczowe znaczenie ma fakt, że znacznik `<c:forEach>` przypisuje każdy element przetwarzanej kolekcji danych do zmiennej, którą zadeklarowaliśmy za pomocą atrybutu `var`.

### Znacznik `<c:forEach>`

Zmienna, która kolejno reprezentuje każdy z ELEMENTÓW przetwarzanej kolekcji danych. Jej wartość zmienia się w każdej iteracji pętli.

```
<c:forEach var="film" items="${listaFilmow}" >
 <td>${film}</td>
</c:forEach>
```

Rzeczywista struktura, którą przetwarzamy w pętli (tablica, egzemplarz klasy `Collection`, mapy bądź łańcuch z elementami oddzielnymi przecinkami).

```
String[] elementy = (String[]) request.getAttribute("listaFilmow");
for (int i = 0; i < elementy.length; i++) {
 String film = elementy[i];
 out.println(film);
}
```

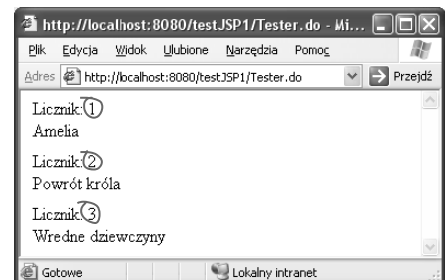
### Odczytywanie licznika pętli za pomocą opcjonalnego atrybutu `varStatus`

Atrybut `varStatus` tworzy nową zmienną, która jest w istocie egzemplarzem klasy `javax.servlet.jsp.jstl.core.LoopTagStatus`.

```
<table>
<c:forEach var="film" items="${listaFilmow}" varStatus="licznikPetliFilmow" >
 <tr>
 <td>Licznik: ${licznikPetliFilmow.count}</td>
 </tr>
 <tr>
 <td>${film}

</td>
 </tr>
</c:forEach>
</table>
```

Na nasze szczęście klasa `LoopTagStatus` zawiera właściwość `count`, która reprezentuje bieżącą wartość licznika iteracji. (Tak jak zmienna „i” w pętli `for`).



## Możemy nawet zagnieżdżać znaczniki `<c:forEach>`

Co będzie, jeśli będziemy musieli wykorzystać w kodzie strony JSP coś takiego jak kolekcja kolekcji? Tablicę tablic? Okazuje się, że możemy zagnieżdżać znaczniki `<c:forEach>` i konstruować za ich pomocą bardziej złożone struktury tabel. W przedstawionym poniżej przykładzie umieściliśmy tablice łańcuchów w egzemplarzu klasy `ArrayList`, po czym uczyniliśmy ten obiekt atrybutem żądania. Strona JSP musi w takim przypadku przeszukać w pętli właśnie tę listę, aby następnie (w pętli wewnętrznej) przeszukać każdą z przechowywanych tam tablic łańcuchów i wyświetlić składowane tam elementy.

### Kod serwletu

```
String[] filmy1 {"Matrix Rewolucje", "Kill Bill", "Święci z Bostonu"};
String[] filmy2 {"Amelia", "Powrót króla", "Wredne dziewczyny"};
java.util.List listaFilmow = new java.util.ArrayList();
listaFilmow.add(filmy1);
listaFilmow.add(filmy2);
request.setAttribute("filmy", listaFilmow);
```

### Kod JSP

```
<table>

 <c:forEach var="elementListy" items="${filmy}" >
 <c:forEach var="film" items="${elementListy}" >
 <td>${film}</td>
 </c:forEach>
 </c:forEach>

</table>
```

*Atrybut żądania typu ArrayList*

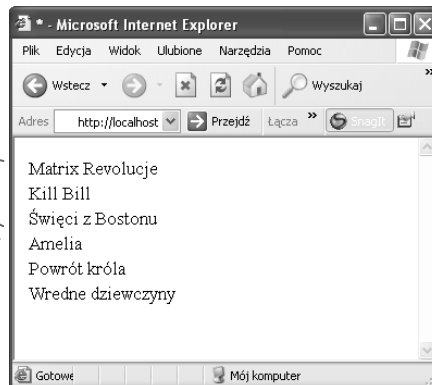
*pętla wewnętrzna*

*Jedna z tablic łańcuchów, która została przypisana do atrybutu "var" zewnętrznej pętli.*

*pętla zewnętrzna*

*Z pierwszej tablicy  
typu String[]*

*Z drugiej tablicy  
typu String[]*



## Nie ma niemądrych pytań

**P:** Skąd wiedziałeś, że atrybut „varStatus” jest egzemplarzem odpowiedniej klasy, i jak się dowiedziałeś, że ta klasa udostępnia właściwość „count”?

**U:** Ach... sami musieliśmy dotrzeć do tych informacji.

Wszystko to można znaleźć w specyfikacji biblioteki JSTL 1.1. Jeśli jeszcze tą specyfikacją nie dysponujesz, NATYCHMIAST wejdź na odpowiednią stronę internetową i pobierz ją (we wstępie do tej książki znajdziesz informacje o specyfikacjach, których znajomość (przynajmniej w części) jest niezbędna do zdania egzaminu). Specyfikacja biblioteki JSTL w wersji 1.1 DEFINIUJE wszystkie jej znaczniki i opisuje z myślą o programistach wszystkie możliwe atrybuty (zarówno te opcjonalne, jak i wymagane), typy atrybutów oraz pozostałe szczegóły związane z ich stosowaniem.

Wszystkie niezbędne (podczas egzaminu) informacje na temat tych znaczników znajdziesz także w tym rozdziale. Niektóre znaczniki oferują jednak kilka dodatkowych opcji, których nie zdołamy tutaj omówić, zatem w wybranych przypadkach lektura oficjalnej specyfikacji może być źródłem wiedzy potrzebnej do tworzenia praktycznych rozwiązań.

**P:** Skoro wiemy więcej, niż określamy w tym znaczniku... czy nie możemy jakoś wpływać na kroki iteracji? W kodzie Javy nie muszę stosować operacji `i++`, mogę, na przykład, użyć wyrażenia `i += 3`, aby przetwarzać w kolejnych iteracjach co trzeci element zamiast wszystkich elementów...

**U:** To żaden problem. Znacznik <c:forEach> zawiera opcjonalne atrybuty dla początku i końca przetwarzania (odpowiednio *begin* i *end*), za pomocą których można wymusić przetwarzanie podzbioru kolekcji, oraz atrybut *step*, za pomocą którego możemy pomijać niektóre elementy.

**P:** Czy przedrostek „c” w znaczniku <c:forEach> jest wymagany?

**U:** Cóż, oczywiście *jakiś* przedrostek zawsze jest wymagany; wszystkie znaczniki i funkcje języka EL muszą być poprzedzone przedrostkiem określającym na potrzeby kontenera przestrzeń nazw, do której należy dany znacznik lub funkcja. Nie MUSISZ jednak używać w tej roli akurat litery „c”. Jest to po prostu standardowa konwencja dla zbioru podstawowych znaczników biblioteki JSTL (ang. *core*). Zalecamy stosowanie przedrostków *innych* niż litera „c” zawsze wtedy, gdy zależy Ci na kompletnym wprowadzeniu w błąd swoich współpracowników.



Oglądaj to!

Zmienna „var” obejmuje swoim zasięgiem **WYŁĄCZNIE** bieżący znacznik!

To prawda, zmienna „var” należy do zasięgu znacznika. Nie jest to oczywiście pełnowartościowy zasięg, z którym można by wiązać atrybuty (tak jak w przypadku pozostałych czterech zasięgów: strony, żądania, sesji i aplikacji). Zasięg znacznika oznacza tylko tyle, że dana zmienna została zadeklarowana WEWNĄTRZ pętli.

Z pewnością zdajesz sobie sprawę z tego, co takie rozwiązanie oznacza w języku Java. Wkrótce przekonasz się, że także w większości pozostałych znaczników zmienna ustawiana za pomocą atrybutu „var” jest widoczna albo we wskazanym (za pomocą atrybutu „scope”) zasięgu, ALBO w domyślnym zasięgu strony.

Staraj się więc nie popełniać błędu polegającego na próbach użycia zmiennej zdefiniowanej za pomocą atrybutu „var” gdzieś PONIŻEJ końca ciała znacznika <c:forEach>!

```
<c:forEach var="foo" items="${listaFoo}" >
 ${foo} ← OK
</c:forEach>
```



NIE!! Zmienna „foo” wyszła poza zasięg swojego znacznika!

W zrozumieniu tego mechanizmu być może pomoże Ci traktowanie zasięgu znacznika tak jak zasięgu bloku w tradycyjnym kodzie Javy. Dobrym przykładem jest znana i lubiana przez wszystkich programistów pętla for:

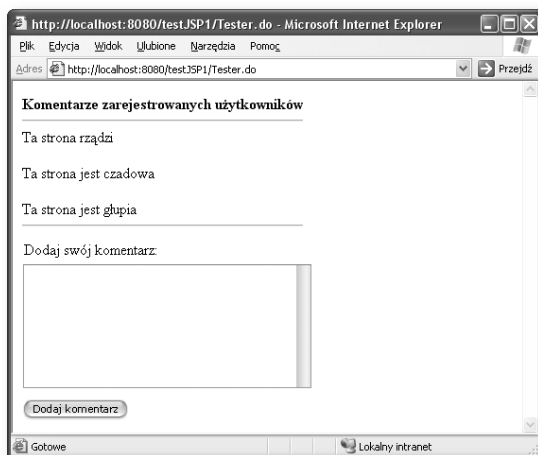
```
for (int i = 0; i < elementy.length; i++) {
 x + i;
}
zrobCos(i);
```

NIE!! Zmienna „i” jest poza bieżącym zasięgiem!

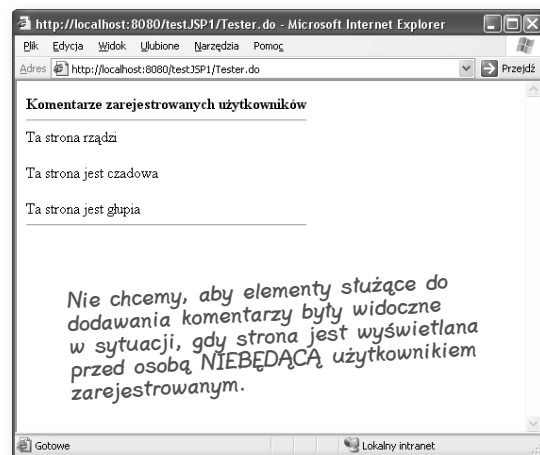
## Warunkowe dołączanie kodu za pomocą znacznika `<c:if>`

Wyobraź sobie, że masz stronę, na której użytkownicy mogą przeglądać komentarze wpisane przez innych użytkowników. Wyobraź sobie także, że komentarze mogą być zamieszczane na stronie wyłącznie przez zarejestrowanych użytkowników, natomiast odczytywanie tych komentarzy jest możliwe niezależnie od stanu użytkownika. **Chcesz, aby każdy korzystał z tej samej strony, ale jednocześnie zależy Ci, aby zarejestrowani użytkownicy „widzieli” więcej elementów na stronie.** Chcesz warunkowo użyć akcji `<jsp:include>` i oczywiście nie masz zamiaru korzystać z elementów skryptowych!

### Co widzą zarejestrowani użytkownicy:



### Co widzą użytkownicy niezarejestrowani:



### Kod JSP

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<html><body>
Komentarze zarejestrowanych użytkowników

<hr>${listaKomentarzy}<hr>
<c:if test="${typUzytkownika eq 'zarejestrowany'}" >
 <jsp:include page="wpisywanieKomentarzy.jsp"/>
</c:if>
</body></html>
```

Przyjmij, że serwet gdzieś ustawił atrybut `typUzytkownika` w oparciu o uzyskane dane logowania.

Tak, wokół słowa 'zarejestrowany' specjalnie użyliśmy APOSTROFÓW. Nie zapominaj, że w swoich znacznikach i wyrażeniach języka EL możesz stosować ZARÓWNO cudzysłowy, jak i apostrofy.

### Dołączana strona („wpisywanieKomentarzy.jsp”)

```
<form action="przetwarzanieKomentarzy.jsp" method="post">
Dodaj swój komentarz:

<textarea name="wpis" cols="40" rows="10"></textarea>

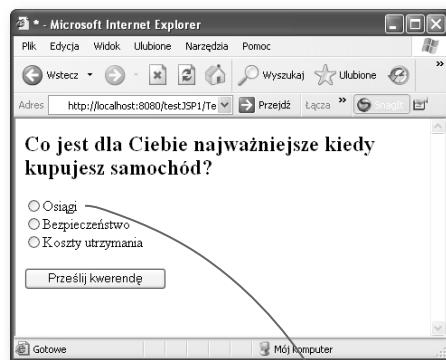
<input name="zatwierdzenieKomentarza" type="button" value="Dodaj komentarz">
</form>
```

# A jeśli potrzebujemy słowa else?

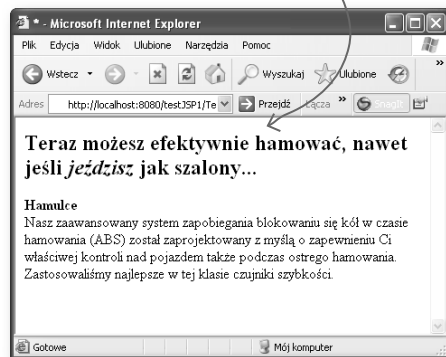
Co powinieneś zrobić, gdyby założyć, że jeśli dany warunek jest prawdziwy, należy wykonać *jedną* operację, a jeśli ten sam warunek jest fałszywy, należy wykonać zupełnie *inną* operację? Innymi słowy, co należałoby zrobić w sytuacji, gdybyśmy chcieli wyświetlić na naszej stronie jedno *lub* drugie, ale jednocześnie chcielibyśmy uniknąć sytuacji, w której *ktokolwiek* miałby dostęp do jednego i drugiego? Użyty na poprzedniej stronie znacznik <c:if> funkcjonował prawidłowo, ponieważ zrealizowaną za jego pomocą logikę można streścić w następujących słowach: *każdy* widzi pierwszą część strony, po czym, jeśli warunek testu jest prawdziwy, wyświetlane są dodatkowe elementy.

Ale wyobraź sobie teraz nieco inny scenariusz: masz witrynę internetową sprzedaży samochodów i chcesz **w oparciu o ustawiony wcześniej (w ramach tej samej sesji) atrybut użytkownika zmieniać nagłówek wyświetlany na stronie**. Większa część strony pozostaje taka sama niezależnie od preferencji użytkownika, ale każdy użytkownik widzi *nagłówek* dostosowany do dokonanego wcześniej wyboru — do opcji, która najlepiej pasuje do osobistych preferencji użytkownika w kwestii kupowania samochodu. (Przecież próbujemy za wszelką cenę sprzedać jak najwięcej samochodów i z czasem zostać nieprzyzwoicie bogatymi ludźmi). Na początku sesji odpowiedni formularz prosi użytkownika o wybór elementu, który jest dla niego najważniejszy...

## Na początku sesji:



## Gdzieś później w ramach tej samej sesji:



Strona użytkownika jest trochę zmieniona, dzięki czemu w większym stopniu odpowiada jego zainteresowaniom.

Wyobraź sobie witrynę internetową koncernu samochodowego. Pierwsza strona pyta użytkownika, co jest dla niego najważniejsze.

Taka witryna powinna funkcjonować podobnie jak dobry sprzedawca w salonie, który, rozmawiając z klientami o cechach oferowanych samochodów, stara się dostosować technikę prezentacji do preferencji użytkownika, aby ci mieli wrażenie, że każdy z tych parametrów został opracowany specjalnie z myślą o ich osobistych potrzebach...

## W tym przypadku sam znacznik `<c:if>` nie wystarczy

Sam znacznik `<c:if>` nie umożliwia osiągnięcia tego celu dokładnie w takiej postaci, w jakiej przed chwilą opisaliśmy, ponieważ **znacznik ten nie zawiera słowa „else”**. Możemy się jednak znacznie zbliżyć do przedstawionego modelu stosując następujące rozwiązania:

### Strona JSP wykorzystująca znacznik `<c:if>`, która jednak nie działa właściwie...

```
<c:if test="${preferencjeUzytkownika=='osiagi'}" >
 Teraz możesz efektywnie hamować, nawet jeśli jeździsz jak szalony...
</c:if>
<c:if test="${preferencjeUzytkownika=='bezpieczenstwo'}" >
 Nasze hamulce Cię nie zawiodą niezależnie od tego, jakim jesteś beznadziejnym kierowcą.
</c:if>
<c:if test="${preferencjeUzytkownika=='utrzymanie'}" >
 Straciłeś pracę? Nie martw się - nie będziesz musiał płacić za przegląd tych hamulców co
 najmniej przez trzy lata.
</c:if>
```

*Ale co będzie, jeśli wartości atrybutu preferencjeUzytkownika nie będą pasowały do żadnej ze zdefiniowanych opcji? Nie mamy przecież możliwości określenia domyślnego nagłówka naszej strony.*

*<!-- Kontynuacja tej części strony, która jest wyświetlana przed WSZYSTKIMI. -->*

Znacznik `<c:if>` nie spełni pokładanych w nim nadziei, chyba że jesteśmy absolutnie PEWNI, że nigdy nie będziemy potrzebowali wartości domyślnej. Tym, czego naprawdę nam trzeba, jest konstrukcja `if-else`.\*

### Strona JSP z elementami skryptowymi, która działa prawidłowo

```
<html><body><h2>
<% String pref = (String) session.getAttribute("preferencjeUzytkownika");
 if (pref.equals("osiagi")) {
 out.println("Teraz możesz efektywnie hamować, nawet jeśli jeździsz jak szalony...");
 } else if (pref.equals("bezpieczenstwo")) {
 out.println("Nasze hamulce Cię nie zawiodą niezależnie od tego, jakim jesteś beznadziejnym
 kierowcą.");
 } else if (pref.equals("utrzymanie")) {
 out.println("Straciłeś pracę? Nie martw się - nie będziesz musiał płacić za przegląd tych
 hamulców co najmniej przez trzy lata.");
 } else {
 // wartość atrybutu preferencjeUzytkownika nie pasowała do żadnego z powyższych warunków,
 // więc wyświetlamy domyślny nagłówek
 out.println("Nasze hamulce są najlepsze.");
 } %>
</h2>Hamulce

```

*Przyjmij, że atrybut "preferencjeUzytkownika" został już ustawiony w ramach tej samej sesji.*

Nasz zaawansowany system zapobiegania blokowaniu się kół w czasie hamowania (ABS) został zaprojektowany z myślą o zapewnieniu Ci właściwej kontroli nad pojazdem także podczas ostrego hamowania. Zastosowaliśmy najlepsze w tej klasie czujniki szybkości. <br>

```
</body></html>
```

\* Tak, zgadzamy się z Tobą — stosowanie połączonych wyrażeń warunkowych `if` niemal nigdy nie jest dobrym rozwiązaniem. Musisz się jednak powstrzymać z wyrażaniem swoich wątpliwości do czasu, aż poznasz wszystkie możliwości w tym obszarze...

WYBIORĘ Cię, KIEDY  
będziesz gotowy na zwalczenie  
własnej obsesji zajęć Pilates.  
W PRZECIWNYM RAZIE do zespołu  
pływania synchronicznego będę  
musiał wziąć Kenny'ego.



## Znacznik <c:choose> i jego partnerzy: <c:when> oraz <c:otherwise>

Nigdy nie zostanie przetworzone więcej niż JEDNO z czterech zdefiniowanych w ten sposób ciał znaczników (włącznie z ciałem znacznika <c:otherwise>).

(W przeciwieństwie do konstrukcji switch, w tym przypadku wybór jednej z opcji automatycznie wyklucza możliwość przetworzenia pozostałych).

<c:choose>

<c:when test="{preferencjeUzytkownika == 'osiagi'}" >

Teraz możesz efektywnie hamować, nawet jeśli <em>jeździsz</em> jak szalony...

</c:when>

<c:when test="{preferencjeUzytkownika == 'bezpieczenstwo'}" >

Nasze hamulce Cię nie zawiodą niezależnie od tego, jakim jesteś beznadziejnym kierowcą.

</c:when>

<c:when test="{preferencjeUzytkownika == 'utrzymanie'}" >

Straciłeś pracę? Nie martw się - nie będziesz musiał płacić za przegląd tych hamulców co najmniej przez trzy lata.

</c:when>

<c:otherwise>

Nasze hamulce są najlepsze.

</c:otherwise>

</c:choose>

<!-- W tym miejscu znajduje się reszta tej strony... -->

Jeśli żaden z testów <c:when> nie był prawdziwy, przetwarzany jest znacznik <c:otherwise> z ciałem domyślnym.

Uwaga: znacznik <c:choose> NIE wymaga definiowania znacznika <c:otherwise>.

# Znacznik `<c:set>` jest... znacznie lepszy od znacznika `<jsp:setProperty>`

Znacznik `<jsp:setProperty>` może robić tylko jedno — ustawiać właściwość wskazanego komponentu.

Ale co będzie, jeśli uznamy, że niezbędne jest ustawienie wartości np. w egzemplarzu mapy? Co, jeśli zechcemy dodać *nowy* element do tej mapy? A co, jeśli będziemy chcieli po prostu stworzyć nowy atrybut zasięgu żądania?

Wszystkie te zadania można zrealizować za pomocą znacznika `<c:set>`, ale musisz się wcześniej zapoznać z kilkoma prostymi regułami. Znacznik ten występuje w dwóch odmianach: **var** i **target**. Wersja *var* służy do ustawiania zmiennych atrybutów, natomiast wersja *target* została zaprojektowana z myślą o ustawianiu właściwości komponentów oraz wartości map. Dla każdej z tych wersji istnieją po dwa warianty: z ciałem i bez ciała. Ciało znacznika `<c:set>` jest po prostu nieco innym sposobem przekazania odpowiedniej wartości.

## Ustawianie zmiennej atrybutu za pomocą znacznika `<c:set>` z atrybutem **var**

### 1 BEZ ciała

Jeśli **NIE** będzie istniał atrybut zasięgu sesji nazwany "poziomUzytkownika", tak skonstruowany znacznik `<c:set>` utworzy odpowiedni atrybut (zakładając, że atrybut value jest różny od null).

```
<c:set var="poziomUzytkownika" scope="session" value="Kowboj" />
```

Atrybut scope jest opcjonalny; atrybut var jest wymagany. **MUSIMY** przecież określić atrybut, ale już decyzja odnośnie zdefiniowania jego wartości w atrybucie value lub w ciele znacznika (patrz przedstawiony poniżej punkt drugi) zależy tylko od nas.

Wartość atrybutu value wcale nie musi być tańcuchem...

```
<c:set var="Fido" value="{osoba.pies}" />
```

Jeśli wynikiem wyrażenia `{osoba.pies}` będzie obiekt klasy *Pies*, wówczas "Fido" będzie właśnie obiektem typu *Pies*.

### 2 Z ciałem

```
<c:set var="poziomUzytkownika" scope="session">
```

Szeryf, Barman, Cowgirl

```
</c:set>
```

Pamiętaj, że jeśli znacznik ma ciało, w tym miejscu nie należy umieszczać ukośnika.

Ciało jest odpowiednio wyznaczane i wykorzystywane w roli wartości zmiennej.



Oglądaj to!

**Jeśli wartością atrybutu value lub ciała znacznika będzie null, zmienna zostanie USUNIĘTA!**  
Tak, dobrze widzisz, usunięta.

Wyobraź sobie, że w roli wartości znacznika `<c:set>` (zdefiniowanej albo w ciele znacznika, albo za pomocą atrybutu value) używamy wyrażenia `{osoba.pies}`. Jeśli wynikiem tego wyrażenia jest wartość `null` (a więc jeśli nie istnieje obiekt *osoba* lub jeśli właściwość *pies* istniejącego obiektu *osoba* jest równa `null`), wówczas, jeśli **ISTNIEJE** atrybut zmiennej nazwany "Fido", atrybut ten zostanie usunięty! (Jeśli nie określimy zasięgu, wyszukiwanie takiego atrybutu rozpocznie się w zasięgu strony, następnie obejmie zasięg żądania itd.). W takim przypadku atrybut "Fido" zostanie usunięty nawet wtedy, gdy okaże się, że był ustawiony jako *String*, *Kaczka* lub *Broku*.

## Stosowanie znacznika `<c:set>` dla komponentów i map

Ta odmiana znacznika `<c:set>` (występująca dodatkowo w dwóch wersjach — z ciałem i bez ciała) działa tylko z dwiema strukturami danych: właściwościami komponentów i wartościami map. To wszystko. Nie możesz użyć znacznika w tej wersji do dodawania elementów do list lub tablic — możesz natomiast przekazać nazwę obiektu (komponentu lub mapy), nazwę właściwości lub klucza oraz wartość.

### Ustawianie docelowej właściwości lub wartości za pomocą znacznika `<c:set>`

#### ① BEZ ciała

```
<c:set target="${MapaZwierzat}" property="imiePsa" value="Clover" />
```

Celem NIE może być null!!

Jeśli celem jest mapa, zostanie ustawiona wartość klucza nazwanego "imiePsa".

Jeśli celem jest komponent, zostanie ustawiona wartość jego właściwości "imiePsa".

#### ② Z ciałem

```
<c:set target="${osoba}" property="imie" >
 ${foo.imie}
</c:set>
```

Nie umieszczaj w tym miejscu nazwy identyfikatora ("id") danego atrybutu.

Brak ukośnika... pamiętaj o tym na egzaminie.

Ciało może mieć postać łańcucha lub (jak w tym przypadku) wyrażenia języka EL.



Oglądaj to!

**Ostateczną wartością atrybutu "target" musi być OBIEKT! W znaczniku `<c:set>` nie podajemy łańcuchowej nazwy "id" komponentu ani atrybutu typu Map!**

Jest to bardzo częste źródło nieporozumień i błędów. Atrybut "target" znacznika `<c:set>` z pozoru powinien funkcjonować tak jak atrybut "id" w znaczniku `<jsp:setProperty>`. Przecież nawet stosowanemu w drugiej wersji znacznika `<c:set>` atrybutowi "var" przypisujemy stałą łańcuchową, która reprezentuje nazwę atrybutu należącego do bieżącego zasięgu. ALE... takie rozwiązanie po prostu nie działa w przypadku atrybutu "target"! Atrybut "target" NIE umożliwia podawania stałych łańcuchowych reprezentujących nazwy, do których zostały dowiązane atrybuty w ramach zasięgu strony, żądania itp. Nie, atrybut "target" wymaga wartości określającej STRUKTURĘ RZECZYWISTĄ. Oznacza to, że musi to być odpowiednie wyrażenie języka EL, wyrażenie skryptowe (`<%= %>`) lub coś, z czym nie mieliśmy jeszcze do czynienia w tej książce — standardowa akcja `<jsp:attribute>`.

## Kluczowe zagadnienia i pułapki znacznika `<c:set>`

Stosowanie znacznika `<c:set>` jest bardzo proste, ale istnieje kilka istotnych reguł, o których musisz pamiętać...

- W znaczniku `<c:set>` nigdy nie możesz używać JEDNOCZEŚNIE atrybutów `"var"` i `"target"`.
- Atrybut `"scope"` jest opcjonalny, ale jeśli go nie zdefiniujesz, zostanie użyty domyślny zasięg *strony*.
- Jeśli atrybut `"value"` jest równy `null`, atrybut wskazany przez `"var"` zostanie usunięty!
- Jeśli atrybut wskazywany przez `"var"` nie istnieje, zostanie utworzony, ale tylko wtedy, gdy atrybut `"value"` będzie różny od `null`.
- Jeśli wyrażenie zdefiniowane w atrybucie `"target"` jest równe `null`, kontener rzuci odpowiedni wyjątek.
- Atrybut `"target"` powinien zawierać wyrażenie, którego rozwiązanie jednoznacznie wskaże konkretny obiekt docelowy. Jeśli umieścisz w tym atrybucie stałą łańcuchową, która będzie reprezentowała nazwę `"id"` danego komponentu lub mapy, takie rozwiązanie po prostu nie zadziała. Innymi słowy, atrybut `"target"` nie służy do wskazywania *nazwy* atrybutu komponentu lub mapy — jego celem jest określanie rzeczywistego *obiekту* atrybutu.
- Jeśli wynikiem wyrażenia użytego w atrybucie `"target"` nie jest ani mapa, ani komponent, kontener rzuci odpowiedni wyjątek.
- Jeśli wynikiem wyrażenia użytego w atrybucie `"target"` jest komponent, który jednak nie zawiera właściwości pasującej do nazwy użytej w atrybucie `"property"`, kontener rzuci wyjątek. Pamiętaj, że wyjątek zostanie wygenerowany także w reakcji na wyrażenie języka EL w postaci: `${komponent.nieprawidłowaWłaściwość}`.

### Nie ma niemądrych pytań

**P:** Po co miałbym używać wersji z ciałem zamiast wersji bez ciała? Wygląda na to, że obie wersje robią dokładnie to samo.

**U:** Dlatego właśnie istnieją dwie różne wersje, które robią to samo. Wersja z ciałem stanowi pewne ułatwienie w sytuacji, gdy programista potrzebuje więcej przestrzeni dla definiowanej wartości. Przecież nie można wykluczyć, że taka wartość będzie miała postać długiego i skomplikowanego wyrażenia — wówczas umieszczenie jej w ciele ułatwia np. czytanie takiego kodu.

**P:** Jeśli nie określę zasięgu, czy będzie to oznaczało, że zostaną przeszukane atrybuty należące wyłącznie do zasięgu strony, czy może przeszukiwanie obejmie wszystkie zasięgi poczynwszy od zasięgu strony?

**U:** Jeśli w swoim znaczniku nie użyjesz opcjonalnego atrybutu `"scope"`, zostanie przeszukany tylko zasięg strony. Przykro nam — po prostu musisz wiedzieć, na którym zasięgu pracujesz.

**P:** Dlaczego słowo „atrybut” występuje w tylu znaczeniach? Słowo to może dotyczyć zarówno „tego, co umieszczamy wewnątrz znacznika”, jak i „tego, co jest związane z obiektami we wszystkich czterech zasięgach”. Efektem stosowania takiego nazewnictwa jest możliwość definiowania atrybutów znacznika, których wartości są atrybutami strony i...

**U:** Wiemy o tym. Niestety, właśnie takie nazwy zaproponowano w odpowiednich specyfikacjach. Znowu nikt NAS nie spytał o radę. Powiązane obiekty nazwalibyśmy nieco inaczej... np. „obektami powiązanyymi”.

Nie mogę uwierzyć,  
że używasz znacznika  
<c:set> do usuwania atrybutów.  
Musi istnieć jakieś lepsze  
rozwiązanie.



Atrybut *var* MUSI być stałą  
tańcuchową! Nie może to być  
wyrażenie!!

## Takim rozwiązaniem jest znacznik <c:remove>

Zgadamy się z Dickiem — używanie znacznika *ustawiającego* do *usuwania* czegokolwiek z pewnością nie jest najlepszym rozwiązaniem. (Pamiętaj jednak, że znacznik <c:set> *usuwa* atrybut tylko wtedy, gdy przekażesz wartość null).

Stosowanie znacznika <c:remove> jest intuicyjne i bardzo proste:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
```

```
<c:set var="statusUzytkownika" scope="request" value="Doskonaly" />
```

```
statusUzytkownika: ${statusUzytkownika}

```

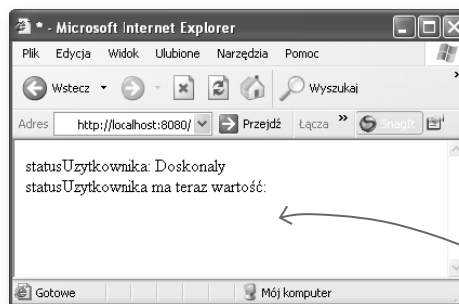
```
<c:remove var="statusUzytkownika" scope="request" />
```

```
statusUzytkownika ma teraz wartość: ${statusUzytkownika}

```

```
</body></html>
```

Atrybut zasięgu jest w tym przypadku  
opcjonalny; jeśli jednak nie określimy  
zasięgu, wskazany atrybut zostanie  
usunięty ze wszystkich zasięgów.



Wartość atrybutu  
statusUzytkownika została  
usunięta, zatem jeśli  
odpowiednie wyrażenie języka  
EL zostanie użyte już PO  
usunięciu atrybutu, nie  
zostanie wyświetlona żadna  
wartość.



## Sprawdź, co pamiętasz o znacznikach

Jeśli przygotowujesz się do egzaminu, koniecznie wykonaj poniższe ćwiczenie. Odpowiedzi znajdziesz na końcu rozdziału.

- ❶ Wpisz nazwę opcjonalnego atrybutu.

```
<c:forEach var="film" items="${listaFilmow}" ="foo" >
 ${film}
</c:forEach>
```

- ❷ Wpisz brakującą nazwę atrybutu.

```
<c:if ="${preferencjeUzytkownika=='bezpieczenstwo'}" >
 Może po prostu powinieneś wybrać spacer...
</c:if>
```

- ❸ Wpisz brakującą nazwę atrybutu.

```
<c:set var="poziomUzytkownika" scope="session" ="foo" />
```

- ❹ Wpisz brakujące nazwy znaczników (dwóch różnych typów) oraz brakującą nazwę atrybutu.

```
<c:choose>
 <c: ="${preferencjeUzytkownika == 'osiagi'}">
 Teraz możesz efektywnie hamować, nawet jeśli jeździsz jak szalony...
 </c: >
 <c: >
 Nasze hamulce są najlepsze.
 </c: >
</c:choose>
```

## Dzięki znacznikowi `<c:import>` mamy do dyspozycji aż TRZY metody dołączania treści

Do tej pory stosowaliśmy dwa różne sposoby dodawania do stron JSP treści pochodzącej z innych zasobów. Istnieje jednak *jeszcze jeden* sposób polegający na zastosowaniu biblioteki JSTL.

### ① Dyrektywa `include`

```
<%@ include file="Naglowek.html" %>
```

**Statyczny:** już w czasie *tłumaczenia* dodaje do bieżącej strony zawartość pliku wskazanego za pomocą atrybutu *file*.

### ② Standardowa akcja `<jsp:include>`

```
<jsp:include page="Naglowek.jsp" />
```

**Dynamiczny:** w czasie realizacji *żądania* dodaje do bieżącej strony zawartość pliku wskazanego za pomocą atrybutu *page*.

W przeciwieństwie do dwóch pozostałych mechanizmów dołączania, używany w znaczniku `<c:import>` adres URL może wskazywać na zasób spoza bieżącego kontenera WWW!

### ③ Znacznik `<c:import>` biblioteki JSTL

```
<c:import url="http://www.wickedlysmart.com/skyler/kon.html" />
```

**Dynamiczny:** w czasie realizacji *żądania* dodaje do bieżącej strony zawartość pliku wskazanego za pomocą atrybutu *url*. Znacznik `<c:import>` działa bardzo podobnie jak standardowa akcja `<jsp:include>`, ale oferuje przy tym większe możliwości i lepszą elastyczność.

NIE myl znacznika `<c:import>` (jednego z mechanizmów dołączania zawartości innych zasobów) z atrybutem *import* dyrektywy *page* (który umożliwia umieszczanie instrukcji importującej Javy w wygenerowanym serwlecie).



Oglądaj to!

**Wszystkie trzy mechanizmy wykorzystują różne nazwy atrybutów! (Pamiętaj także o niespójności w zakresie stosowania słów „include” i „import”).**

Każdy z wymienionych powyżej trzech mechanizmów dołączania do stron JSP zawartości innych zasobów wykorzystuje inaczej nazwane atrybuty. Dyrektywa `include` używa nazwy *file*, standardowa akcja `<jsp:include>` używa nazwy *page*, zaś znacznik `<c:import>` biblioteki JSTL używa nazwy *url*. Jeśli dobrze się nad tym zastanowisz, dojdiesz do wniosku, że takie różnice są uzasadnione... musisz jednak zapamiętać wszystkie trzy schematy. Dyrektywa `include` była początkowo projektowana z myślą o statycznych szablonach układu, takich jak nagłówki HTML. Innymi słowy, jej zadaniem jest dołączanie plików (atrybut *file*). Standardowa akcja `<jsp:include>` miała w zamyśle swoich twórców służyć w większym stopniu do dołączania dynamicznej zawartości innych stron JSP, w związku z czym odpowiedni atrybut nazwano *page*. Atrybut znacznika `<c:import>` nazwano w sposób dokładnie odpowiadający przekazywanej za jego pomocą wartości: URL! Pamiętaj, że pierwsze dwa mechanizmy *„include”* nie mogą dołączać zasobów spoza bieżącego kontenera, taką możliwość daje tylko znacznik `<c:import>`.

## Znacznik `<c:import>` może dołączać zasoby SPOZA bieżącej aplikacji internetowej

Za pomocą standardowej akcji `<jsp:include>` lub dyrektywy `include` możemy dołączać wyłącznie strony należące do bieżącej aplikacji internetowej. Ale obecnie, dzięki znacznikowi `<c:import>`, mamy także możliwość dołączania zawartości zasobów *spoza* danego kontenera. Przedstawiony poniżej prosty przykład pokazuje, jak strona JSP na serwerze *A* importuje zawartość zasobu z serwera *B* (na podstawie użytego adresu URL). Zaimportowany fragment kodu zawiera referencję do obrazu, który także znajduje się na serwerze *B*.

### Serwer A, strona JSP importująca zawartość zasobu

#### Strona JSP

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<html><body>

 <c:import url="http://www.wickedlysmart.com/skyler/kon.html" />

 To jest mój koń.

</body></html>
```



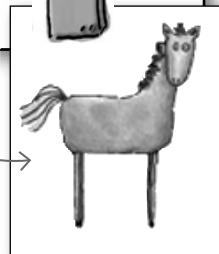
(Nie zapomnij jednego: podobnie jak w przypadku pozostałych mechanizmów dołączania zawartości zasobów zewnętrznych, importowany kod powinien być fragmentem strony HTML, a NIE kompletną stroną z otwierającymi i zamykającymi znacznikami `<html><body>`).

### Serwer B, importowana zawartość

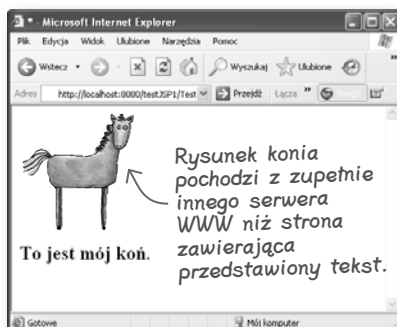
#### Importowany plik

```

```



### Odpowiedź



Zarówno plik „kon.html”, jak i plik graficzny „kon.gif” znajdują się na serwerze B, czyli zupełnie innym serwerze WWW niż ten, który udostępnia daną stronę JSP.

## Dostosowywanie dołączanej zawartości

Pamiętasz, jak w poprzednim rozdziale wykorzystaliśmy standardową akcję `<jsp:include>` do dołączania do strony JSP kodu układu nagłówka (grafiki z tekstem) i jak próbowaliśmy dostosować wyświetlany w tym nagłówku podtytuł do profilu użytkownika? Oczekiwany efekt udało nam się uzyskać dzięki standardowej akcji `<jsp:param>`...

### ① Strona JSP ze standardową akcją `<jsp:include>`

```
<html><body>

<jsp:include page="Naglowek.jsp">

 <jsp:param name="podTytuł" value="Wiemy jak ułatwić stosowanie
 ↳protokołu SOAP." />

</jsp:include>

Witamy w naszej grupie wsparcia dla usług internetowych.

Skontaktuj się z nami: ${initParam.głównyEmail}

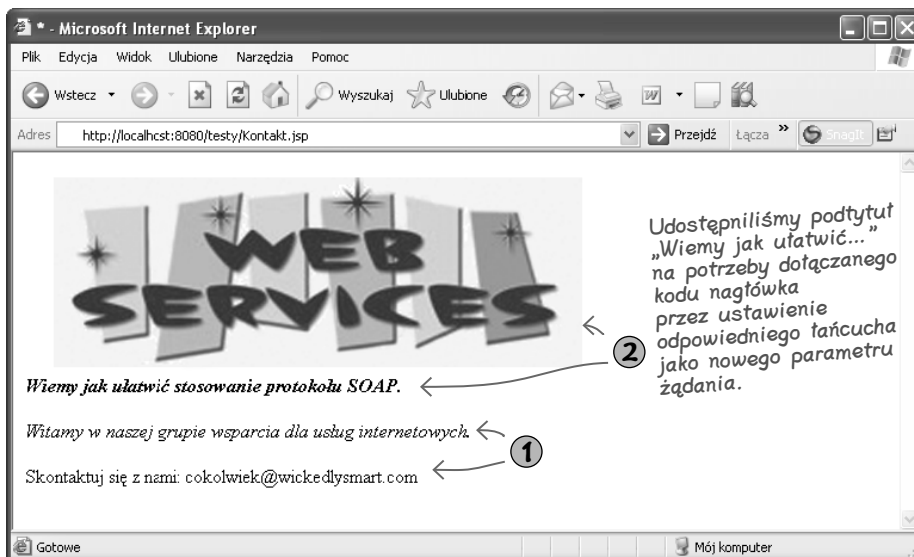
</body></html>
```

### ② Dołączany plik („Naglowek.jsp”)

```


${param.podTytuł}


```



## Realizacja tego samego zadania za pomocą znacznika `<c:param>`

Za moment osiągniemy ten sam cel co na poprzedniej stronie, ale tym razem posłużymy się kombinacją znaczników `<c:import>` oraz `<c:param>`. Przekonasz się, że struktura tego rozwiązania jest niemal identyczna jak wtedy, gdy używaliśmy akcji standardowych.

### 1 Strona JSP ze znacznikiem `<jsp:import>`

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<html><body>

<C:import url="Naglowek.jsp" >

 <C:param name="podTytuł" value="Wiemy jak ułatwić stosowanie protokołu SOAP." />

</C:import>

Witamy w naszej grupie wsparcia dla usług internetowych.

Skontaktuj się z nami: ${initParam.glownyEmail}

</body></html>
```

Brak ukośnika, ponieważ TYM RAZEM nasz znacznik ma ciało.

### 2 Dołączany plik („Naglowek.jsp”)

```


${param.podTytuł}


```

Dołączana strona w ogóle się nie zmienia. Dopóki nie mamy problemu z odczytaniem interesującego nas parametru, zupełnie nas nie obchodzi, JAK ten parametr jest umieszczany na właściwym miejscu.

## Przepisywanie adresów URL w kodzie JSP

Przepraszam za zmianę tematu w tym miejscu, ale właśnie wykryłem ZASADNICZY problem związany ze stosowaniem technologii JSP! Czy jesteś w stanie zagwarantować śledzenie sesji z poziomu kodu JSP... bez elementów skryptowych?



W przypadku stron JSP śledzenie sesji jest realizowane automatycznie, chyba że wprost wyłączymy ten mechanizm za pomocą dyrektywy page z atrybutem session równym false (session="false").



Chyba mnie nie zrozumiał... powiedziałem „zagwarantować”. Moje pytanie sprowadza się do jednego – jeśli klient nie obsługuje znaczników kontekstu klienta (ciasteczek), jak mogę wymusić przepisywanie adresów URL? Jak mogę odczytać identyfikator sesji dodany do adresów URL w mojej stronie JSP?



Ach... on pewnie nie wie jeszcze o znaczniku <c:url>. Za jego pomocą można przepisywać adresy URL zupełnie automatycznie.



## Stosowanie znacznika `<c:url>` do realizacji wszystkich zadań związanych z obsługą hiperłączy

Pamiętasz jeszcze dawne czasy, kiedy w naszym starym serwlecie chcieliśmy używać sesji? W pierwszej kolejności musieliśmy *uzyskać* sesję (albo już istniejącą, albo nową). Kontener wiedział wówczas, że oczekujemy od niego powiązania klienta, który przysłał bieżące żądanie, z konkretnym identyfikatorem sesji. Kontener *chce* w takich sytuacjach stosować znaczniki kontekstu klienta (tzw. *ciasteczka*) — chce dołączyć unikatowy znacznik do odpowiedzi, aby klient odsyłał ten znacznik wraz z każdym kolejnym żądaniem. Wszystko działa znakomicie, z jednym wyjątkiem... klient może korzystać z przeglądarki z wyłączoną obsługą ciasteczek. Co wtedy?

Kontener automatycznie będzie podejmował próby przepisania adresu URL, jeśli nie otrzyma ciasteczka od klienta. Jednak w przypadku serwetów musimy JESZCZE kodować swoje adresy URL. Innymi słowy, nadal *musimy* wymusić na kontenerze „dołączanie parametru `jsessionId` na koniec konkretnego adresu URL...” dla każdego adresu, w przypadku którego taki parametr ma znaczenie. Cóż, okazuje się, że możemy zrobić dokładnie to samo z poziomu kodu strony JSP — wystarczy użyć znacznika `<c:url>`.

### Przepisywanie adresów URL z poziomu serwetu

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
 throws IOException,
ServletException {

 response.setContentType("text/html");
 PrintWriter out = response.getWriter();
 HttpSession session = request.getSession();

 out.println("<html><body>");
 out.println("kliknij");
 out.println("</body></html>");
}
```

Do tego adresu URL dołączamy dodatkowe informacje o identyfikatorze sesji.

### Przepisywanie adresów URL z poziomu strony JSP

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<html><body>
```

To jest hiperłącze z włączonym przepisywaniem adresu URL.

```
<a href="<c:url value='/wpisywanieKomentarzy.jsp' />">Kliknij tutaj
```

</body></html>

W ten sposób dodajemy parametr `jsessionId` na koniec względnego adresu URL z atrybutu „value” (jeśli obsługa ciasteczek po stronie przeglądarki jest wyłączona).

## Co należy zrobić, jeśli adres URL wymaga kodowania?

Pamiętasz, że w ramach żądania GET protokołu HTTP do adresu URL można dołączać parametry w postaci łańcucha zapytania? Jeśli na przykład formularz na stronie HTML zawiera dwa pola tekstowe — dla imienia i nazwiska użytkownika — URL żądania będzie obejmował nazwy i wartości odpowiednich parametrów (umieszczone na końcu, w formie łańcucha zapytania). Ale żądanie HTTP nie będzie funkcjonowało prawidłowo, jeśli łańcuch zapytania będzie zawierał tzw. *niebezpieczne* znaki (choć większość współczesnych przeglądarek będzie próbowała zapobiec wynikającym z tego komplikacjom).

Jeśli jesteś programistą (projektantem) stron internetowych, doskonale zdajesz sobie z tego sprawę. Jeśli jednak zajmujesz się tworzeniem aplikacji internetowych od niedawna, musisz wiedzieć, że adresy URL często wymagają odpowiedniego *kodowania*. Kodowanie URL oznacza zastąpienie niebezpiecznych (zastrzeżonych) znaków odpowiednimi znakami zastępczymi — zakodowany adres URL trzeba oczywiście odkodować po stronie serwera. Przykładowo, w adresach URL zabronione jest stosowanie spacji, ale można je łatwo zastąpić znakiem plusa (+). Problem w tym, że znacznik <c:url> automatycznie NIE koduje naszych adresów URL!

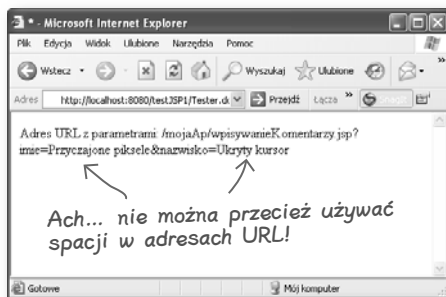
### Stosowanie znacznika <c:url> z łańcuchem zapytania

Pamiętaj, znacznik <c:url> odpowiada za przepisywanie adresu URL, ale *nie* za kodowanie URL!

```
<c:set var="nazwisko" value="Ukryty kursor" />
<c:set var="imie" value="Przyczajone piksele" />
<c:url value="/wpisywanieKomentarzy.jsp?imie=${imie}&nazwisko=${nazwisko}" var="wejsciowyURL" />
```

Adres URL z parametrami: \${wejsciowyURL} <br>

Jeśli chcesz mieć dostęp do tej wartości w kolejnych znacznikach, użyj opcjonalnego atrybutu "var".



Jasne! Parametry łańcucha zapytania muszą być kodowane... przykładowo, wszystkie spacje muszą zostać zastąpione znakiem plusa "+".

### Używanie znacznika <c:param> w ciele znacznika <c:url>

To rozwiązuje nasz problem! Mamy teraz zarówno mechanizm przepisywania adresów URL, jak i mechanizm ich kodowania.

```
<c:url value="/wpisywanieKomentarzy.jsp" var="wejsciowyURL" >
 <c:param name="imie" value="${imie}" />
 <c:param name="nazwisko" value="${nazwisko}" />
</c:url>
```

bez ukośnika

Teraz jesteśmy bezpieczni, ponieważ znacznik <c:param> zajmie się kodowaniem!

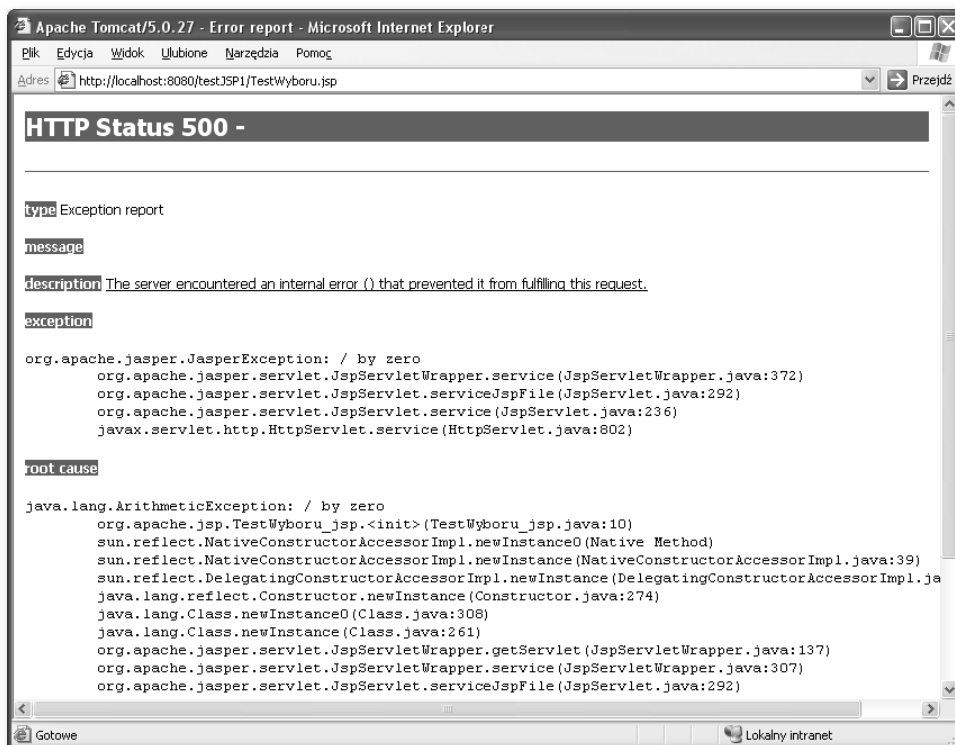
Przetworzony adres URL wygląda teraz następująco:

/mojaAp/wpisywanieKomentarzy.jsp?imie=Przyczajone+piksele&nazwisko=Ukryty+kursor

Pozwólcie,  
że przerwę na chwilę tę  
dyskusję o JSTL, aby zwrócić  
uwagę na właściwą obsługę błędów.  
Za chwilę zrobimy coś, co może  
doprowadzić do rzucenia  
wyjątku...



**NIE chcesz, aby Twoi klienci widzieli coś takiego:**



# Przygotuj własne strony o błędach

Facet surfujący po Twojej witrynie internetowej nie chce przecież oglądać tego, co znajdowało się na Twoim stosie w momencie wystąpienia błędu. Nie ma także wystarczająco mocnych nerwów, aby znosić kolejne strony z komunikatem *404 Not Found*.

Nie możesz oczywiście zapobiec *wszystkim* błędom, ale możesz przynajmniej zapewnić użytkownikowi bardziej przyjazną (i atrakcyjną) stronę odpowiedzi (w przypadku wystąpienia błędu). Możesz zaprojektować własną stronę obsługującą błędy, po czym użyć dyrektywy `page` do jej właściwego skonfigurowania.

## Wyznaczona strona BŁĘDU („stronaBledu.jsp”)

```
<%@ page isErrorPage="true" %>
```

Potwierdzenie dla kontenera: „Tak, to JEST oficjalnie wyznaczona strona błędu.”

```
<html><body>
Rozpacz.

</body></html>
```

## ZŁA strona, która zwraca wyjątek („zlaStrona.jsp”)

```
<%@ page errorPage="stronaBledu.jsp" %>
```

```
<html><body>
```

```
Stworzona, żeby szerzyć zło...
```

```
<% int x = 10/0; %>
```

```
</body></html>
```

Informacja dla kontenera: „Jeśli coś na tej stronie pójdzie nie tak, przekaz żądanie do strony `stronaBledu.jsp`.”

## Jaka będzie odpowiedź na żądanie strony „zlaStrona.jsp”



ŻĄDANIE było co prawda kierowane na stronę „`zlaStrona.jsp`”, ale w czasie przetwarzania tego żądania wystąpił wyjątek, zatem ODPOWIEDŹ pochodzi ze strony „`stronaBledu.jsp`”.

Umieszczanie we wszystkich moich stronach JSP dyrektyw `page` wskazujących na właściwą stronę błędu zajmie mi całe WIEKI. A co będzie, jeśli będę chciała używać różnych stron w zależności od wykrytych błędów? Gdyby tylko istniał sposób szybkiego skonfigurowania stron błędów dla całej aplikacji internetowej...

oo



## Ona nie wie jeszcze o znaczniku

### <error-page> deskryptora wdrożenia

Strony o błędach dla całej aplikacji internetowej można łatwo deklarować w deskrytorze wdrożenia, można nawet wyznaczyć *różne* strony dla różnych typów wyjątków lub różnych kodów błędów protokołu HTTP (404, 500 etc).

Warto przy tym pamiętać, że kontener wykorzystuje zdefiniowaną w deskrytorze wdrożenia konfigurację `<error-page>` wyłącznie w roli ustawień standardowych — jeśli w kodzie strony JSP dodatkowo użyjemy dyrektywy `errorPage`, kontener użyje właśnie tej dyrektywy.

# Konfiguracja stron błędów w deskrypcji wdrożenia

Możemy zadeklarować strony błędów w deskrypcji wdrożenia albo z wykorzystaniem znacznika `<exception-type>`, albo w oparciu o numer stanu protokołu HTTP (za pomocą znacznika `<error-code>`). W ten sposób można odsyłać klientowi różne strony w zależności od rodzaju problemu, który był źródłem błędu.

## Deklarowanie strony dla wszystkich możliwych błędów

Niniejsza deklaracja będzie stosowana dla wszystkich składników Twojej aplikacji internetowej — nie tylko stron JSP. Możesz jednak przykrywać tego typu ustawienia w poszczególnych stronach JSP przez dodawanie dyrektywy `page` z atrybutem `errorPage`.

```
<error-page>
 <exception-type>java.lang.Throwable</exception-type>
 <location>/stronaBledu.jsp</location>
</error-page>
```

## Deklarowanie strony błędu dla konkretnych wyjątków

Niniejsza deklaracja konfiguruje stronę błędu, która będzie wywoływana tylko wtedy, gdy w czasie wykonywania aplikacji wystąpi wyjątek `ArithmeticException` (operacji arytmetycznej). Jeśli użyjemy zarówno tej deklaracji, jak i przedstawionej przed chwilą deklaracji strony dla wszystkich błędów, wszelkie wyjątki różne od `ArithmeticException` nadal będą obsługiwane przez stronę `"stronaBledu.jsp"`.

```
<error-page>
 <exception-type>java.lang.ArithmeticException</exception-type>
 <location>/bladArytmetyczny.jsp</location>
</error-page>
```

## Deklarowanie strony błędu na bazie kodu stanu protokołu HTTP

Niniejsza deklaracja konfiguruje stronę błędu, która będzie wywoływana tylko wtedy, gdy kod stanu dla odpowiedzi będzie równy `"404"` (nie znaleziono pliku).

```
<error-page>
 <error-code>404</error-code>
 <location>/bladNieZnaleziona.jsp</location>
</error-page>
```

W znaczniku `<location>` MUSI znaleźć się ścieżka względna wobec podkatalogu `context` w katalogu głównym aplikacji internetowej, co oznacza, że stosowana w tym miejscu ścieżka MUSI rozpoczynać się od ukośnika. (Ten warunek obowiązuje nas niezależnie od tego, czy strona błędu jest zwracana w oparciu o znacznik `<error-code>`, czy znacznik `<exception-type>`).

## Strony błędów otrzymują dodatkowy obiekt — exception

Strona błędu jest w istocie stroną JSP, która *obsługuje* wyjątek, zatem kontener udostępnia tej stronie dodatkowy obiekt reprezentujący ten *wyjątek* (obiekt nazwany `exception`). Prawdopodobnie nie będziesz chciał wyświetlać przed użytkownikiem informacji o rzuconym wyjątku, ale na wszelki wypadek masz do nich dostęp. W skrypcie możesz użyć obiektu domyślnego `exception`, natomiast z poziomu kodu JSP możesz używać obiektu niejawnego języka EL: `${pageContext.exception}`. Udostępniany w ten sposób obiekt `exception` jest egzemplarzem typu `java.lang.Throwable`, zatem na poziomie swoich skryptów możesz wywoływać metody tego interfejsu, a w wyrażeniach EL możesz uzyskiwać dostęp do właściwości `stackTrace` oraz `message`.

### Bardziej konkretna strona BŁĘDU („stronaBledu.jsp”) ←

```
<%@ page isErrorPage="true" %>
```

```
<html><body>
```

```
Rozpacz.
```

```
Spowodowałeś na serwerze wyjątek ${pageContext.exception}.

```

```

```

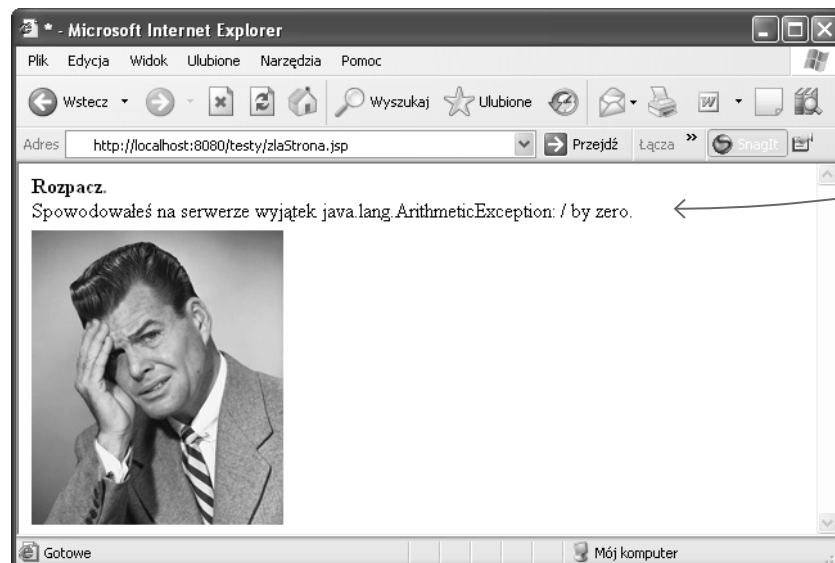
```
</body></html>
```

Uwaga: obiekt domyślny `exception` jest dostępny TYLKO dla stron błędów zawierających zdefiniowaną wprost dyrektywę `page`:

```
<%@ page isErrorPage="true" %>
```

Innymi słowy, samo skonfigurowanie strony błędu w deskrytorze wdrożenia nie wystarczy, by kontener przekazywał do tej strony domyślny obiekt wyjątku!

### Jaka będzie odpowiedź na żądanie strony „zlaStrona.jsp”



Tym razem otrzymaliśmy trochę więcej szczegółowych informacji. Najprawdopodobniej nie będziesz prezentował tego typu danych użytkownikowi... zrobiliśmy to tylko po to, abyś wiedział, jak korzystać z tego typu możliwości.

A co będzie, jeśli  
uznam, że istnieje wyjątek, który  
być może będę w stanie naprawić  
z poziomu mojej strony JSP? Co  
powinienem zrobić, jeśli uznam, że  
niektóre błędy chcę sam wykrywać  
i obsługiwać?

## Znacznik <c:catch>, który przypomina trochę... konstrukcje try-catch

Jeśli masz stronę, która wywołuje ryzykowny znacznik, ale sądzisz, że możliwe jest naprawienie ewentualnego błędu, możesz skorzystać z rozwiązania przygotowanego w tym celu przez twórców biblioteki JSTL. Możesz wykonać coś w rodzaju polecenia try-catch za pomocą specjalnego znacznika <c:catch>, aby otoczyć i odpowiednio obsłużyć ryzykowny znacznik lub wyrażenie. Jeśli tego nie zrobisz, a w czasie przetwarzania żądania zostanie rzucony wyjątek, kontener uruchomi domyślny mechanizm obsługi błędów, który wykorzysta stronę błędu zadeklarowaną w deskrytorze wdrożenia. Elementem, który może początkowo budzić zdziwienie, jest fakt, że znacznik <c:catch> występuje jednocześnie w roli polecenia próby, *jak i* w roli instrukcji obsługującej ewentualne błędy — nie istnieje osobny znacznik *try*. Ryzykowne wyrażenie języka EL, wywołania znaczników lub cokolwiek innego należy umieścić w ciele znacznika <c:catch>, ponieważ właśnie tam będą przechwytywane ewentualne wyjątki. Nie możesz jednak zakładać, że funkcjonowanie tego znacznika dokładnie odpowiada działaniu bloku catch, ponieważ kiedy już wystąpi wyjątek, kontrola jest natychmiast przenoszona na koniec ciała znacznika <c:catch> (więcej uwagi poświęcimy temu zagadnieniu za chwilę).

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ page errorPage="stronaBledu.jsp" %>
<html><body>
```

Zaraz zrobimy coś bardzo ryzykownego: <br>

```
<c:catch>
```

```
<% int x = 10/0; %>
```

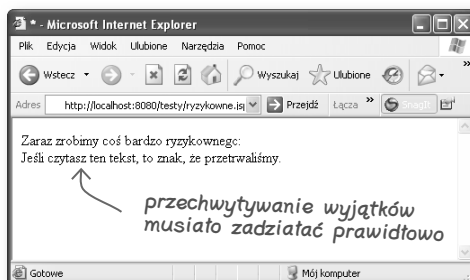
```
</c:catch>
```

Jeśli czytasz ten tekst, to znak, że przetrwaliśmy.

```
</body></html>
```

Ten skryptlet z **PEWNOŚCIĄ** spowoduje wyjątek...  
jednak postaramy się ten wyjątek przechwycić,  
zamiast wywoływać stronę błędu.

Jeśli ten tekst zostanie  
wyświetlony, będziemy  
**WIEDZIELI**, że wyjątek  
został obsłużony prawidłowo  
(co w tym przypadku oznacza,  
że przechwycenie wyjątku  
zakończyło się sukcesem).



Ale jak w tej sytuacji mogę uzyskać dostęp do obiektu wyjątku? Do wyjątku, który właśnie został rzucony. Ponieważ przechwycony wyjątek nie jest traktowany jak błąd (a więc nie powoduje otwarcia strony błędu), domyślny obiekt wyjątku pewnie już nie będzie działał jak do tej pory.



## Możesz przekształcić obiekt wyjątku w atrybut strony

W popularnych wyrażeniach try-catch języka Java argumentem instrukcji catch jest obiekt wyjątku. Podczas stosowania mechanizmów obsługi błędów w aplikacjach internetowych musisz jednak pamiętać, że *obiekt wyjątku jest udostępniany tylko stronom błędów, które zostały oficjalnie wyznaczone do tej roli*. Dla wszystkich pozostałych stron wyjątek po prostu nie istnieje. Oznacza to, że niniejsze rozwiązanie *nie* zadziała:

```
<c:catch>
```

W znaczniku catch...

```
<% int x = 10/0; %>
```

```
</c:catch>
```

Wyjątek: ~~`{pageContext.exception}`~~

Nie zadziała, ponieważ to nie jest oficjalnie wyznaczona do tego celu strona błędu, zatem nie mamy dostępu do obiektu wyjątku.

### Stosowanie atrybutu „var” w znaczniku <c:catch>

Jeśli chcemy mieć dostęp do obiektu wyjątku już po zakończeniu przetwarzania znacznika <c:catch>, powinniśmy użyć opcjonalnego atrybutu *var*. Atrybut ten umieszcza obiekt wyjątku w zasięgu strony (pod nazwą, którą *zadeklarujemy* w formie wartości atrybutu *var*).

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
```

```
<%@ page errorPage="stronaBledu.jsp" %>
```

```
<html><body>
```

Zaraz zrobimy coś bardzo ryzykownego: <br>

```
<c:catch var="mojWyjatek">
```

W znaczniku catch...

```
<% int x = 10/0; %>
```

```
</c:catch>
```

```
<c:if test="{mojWyjatek != null}">
```

```
Wystąpił wyjątek: {mojWyjatek.message}

```

```
</c:if>
```

```
Przetrwaliśmy.
```

```
</body></html>
```

W ten sposób tworzymy nowy atrybut zasięgu strony (nazwany „mojWyjatek”) i przypisujemy do tego atrybutu obiekt wyjątku.

Teraz mamy do dyspozycji atrybut *mojWyjatek*, a ponieważ typem tego atrybutu jest *Throwable*, musi udostępniać właściwość „message” (ponieważ interfejs *Throwable* zawiera metodę *getMessage()*).



**Sterowanie przepływem w przypadku znacznika <c:catch> działa w taki sam sposób jak w przypadku bloku try — po wykryciu wyjątku nie zostanie wykonana **ŻADNA** kolejna instrukcja z ciała tego znacznika.**

W tradycyjnym wyrażeniu try-catch języka Java wystąpienie wyjątku powoduje, że kod zdefiniowany **PONIŻEJ** tego punktu w bloku try nigdy nie zostanie wykonany — sterowanie jest przekazywane bezpośrednio do bloku catch. W przypadku znacznika <c:catch> w razie wystąpienia wyjątku wykonywane są dwie czynności:

- 1) Jeśli użyto opcjonalnego atrybutu "var", obiekt wyjątku jest do niego przypisywany.
- 2) Sterowanie jest przekazywane bezpośrednio **za** całe ciało znacznika <c:catch>.

```
<c:catch>
W znaczniku catch...
<% int x = 10/0; %>
Po przechwyceniu wyjątku...
</c:catch>
```

sterowanie

NIGDY tego nie zobaczymy!

Przetrwaliśmy.

Uważaj na to. Jeśli chcesz użyć obiektu wyjątku "var", musisz poczekać, aż znajdziesz się **POZA** ciałem znacznika <c:catch>. Innymi słowy, nie ma możliwości wykorzystywania informacji o wykrytym wyjątku **WEWNĄTRZ** ciała znacznika <c:catch>.

Wielu programistów ulega pokusie i traktuje znacznik <c:catch> tak jak normalny blok catch w kodzie Javy, znaczenie tych elementów jest jednak zupełnie inne. Znacznik <c:catch> bardziej przypomina blok try, ponieważ to w jego ciele umieszczamy ryzykowny kod. Różnica polega na tym, że znacznik <c:catch> jest jak blok try, który nigdy nie będzie potrzebował ani bloku catch, ani bloku finally. Czy to jasne? Chodzi tylko o jedno — naucz się korzystać z tego znacznika w takiej postaci, w jakiej rzeczywiście występuje; nie szukaj na siłę analogii pomiędzy znacznikiem <c:catch> a stosowaną w Javie konstrukcją try-catch. A jeśli na egzaminie zobaczysz kod wewnątrz znacznika <c:catch> umieszczony poniżej punktu będącego potencjalnym źródłem wyjątku, nie daj się zwieść.

## Co będzie, jeśli uznamy za niezbędne użycie znacznika SPOZA biblioteki JSTL?

Biblioteka JSTL jest ogromna. Wersja 1.1 tak naprawdę obejmuje aż *pięć* bibliotek — cztery z niestandardowymi *znacznikami* oraz jedną z zestawem *funkcji* do przetwarzania łańcuchów. Znaczniki prezentowane w tej książce (których znajomość jest wymagana podczas egzaminu), dotyczą wyłącznie ogólnych i najpopularniejszych działań, ale prawdopodobnie właśnie wśród znaczników i funkcji udostępnianych przez wszystkie pięć bibliotek znajdziesz wszystko, czego kiedykolwiek będziesz potrzebował. Na następnej stronie przystąpimy do analizy sytuacji, w której wymienione poniżej znaczniki okazują się niewystarczające.

### Biblioteka podstawowa („core”)

#### Ogólnego przeznaczenia

<c:out>  
<c:set>  
<c:remove>  
<c:catch>

#### Warunkowe

<c:if>  
<c:choose>  
<c:when>  
<c:otherwise>

#### Związane z obsługą URL

<c:import>  
<c:url>  
<c:redirect>  
<c:param>

#### Iteracyjne

<c:forEach>  
<c:forEachTokens>

### Biblioteka formatowania („formatting”)

#### Ustawienia międzynarodowe

<fmt:message>  
<fmt:setLocale>  
<fmt:bundle>  
<fmt:setBundle>  
<fmt:param>  
<fmt:requestEncoding>

#### Formatowanie

<fmt:timeZone>  
<fmt:setTimeZone>  
<fmt:formatNumber>  
<fmt:parseNumber>  
<fmt:parseDate>

### Biblioteka SQL („SQL”)

#### Dostęp do bazy danych

<sql:query>  
<sql:update>  
<sql:setDataSource>  
<sql:param>  
<sql:dateParam>

### Biblioteka XML („XML”)

#### Podstawowa obsługa XML-a

<x:parse>  
<x:out>  
<x:set>

#### Sterowanie przepływem w XML

<x:if>  
<x:choose>  
<x:when>  
<x:otherwise>  
<x:forEach>

#### Transformacje

<x:transform>  
<x:param>



Relax

Na egzaminie będzie sprawdzana wiedza wyłącznie z zakresu biblioteki podstawowej („core”)

Podstawowa biblioteka znaczników JSTL (którą zgodnie z konwencją cały czas oznaczamy przedrostkiem „c”) jest jedyną biblioteką JSTL, której znajomość jest sprawdzana na oficjalnym egzaminie. Pozostałe cztery biblioteki są bardziej wyspecjalizowane, więc nie będziemy ich szczegółowo omawiali. Powinieneś jednak wiedzieć, że takie biblioteki w ogóle istnieją. Przykładowo, znaczniki transformacji XML-a mogą nam uratować życie, jeśli będziemy zmuszeni do przetwarzania dokumentów RSS. Pisanie własnych, niestandardowych znaczników bywa dosyć czasochłonne, zatem przed przystąpieniem do tego typu czynności warto sprawdzić, czy nie będzie to ponowne odkrywanie koła.

Tego znacznika nie omawialiśmy... za jego pomocą możemy iteracyjnie przeszukiwać elementy struktury danych w oparciu o zdefiniowany przez NAS separator. Cóżś działa podobnie jak klasa StringTokenizer. Nie wspominaliśmy także o znaczniku <c:redirect>, ale jego obecność na przedstawionej liście powinna być dla Ciebie doskonałym pretekstem do otwarcia dokumentacji JSTL.

# Stosowanie biblioteki znaczników NIEBĘDĄCEJ częścią standardowej biblioteki JSTL

Tworzenie kodu, który będzie wykonywany w *tle* znacznika (a więc przygotowanie kodu Javy wywoływanego w przypadku umieszczenia danego znacznika w kodzie strony JSP) nie jest zadaniem trywialnym. Zagadnieniom związanym z opracowywaniem własnych klas obsługujących niestandardowe znaczniki poświęciliśmy cały następny rozdział. W ostatniej

części bieżącego rozdziału chcielibyśmy jednak omówić techniki *stosowania* takich znaczników. Co powinieneś zrobić, jeśli, na przykład, ktoś udostępni Ci swoją bibliotekę znaczników przygotowaną specjalnie dla Twojej firmy lub dla realizowanego przez Ciebie projektu? Skąd będziesz wiedział, jakie znaczniki są dostępne w tej bibliotece i jak z nich korzystać? W przypadku biblioteki JSTL nie mamy do czynienia z tego typu problemami, ponieważ specyfikacja JSTL 1.1 dokładnie *dokumentuje* każdy znacznik włącznie z technikami stosowania wszystkich wymaganych i opcjonalnych atrybutów.

Jednak nie każdy niestandardowy znacznik jest udostępniany w tak dobrze zaprojektowanym i precyzyjnie udokumentowanym pakiecie. Musisz więc wiedzieć, jak znaleźć potrzebny znacznik nawet wtedy, gdy odpowiednia dokumentacja jest nieprecyzyjna, lub gdy w ogóle nie dysponujesz dokumentacją. I jeszcze jedno — musisz wiedzieć, jak *wdrożyć* bibliotekę znaczników niestandardowych.

Zanim użyjesz niestandardowej biblioteki, **MUSISZ** zapoznać się z zawartością pliku TLD.

Znajdziesz tam wszystkie potrzebne informacje.

## Główne elementy, które powinieneś znać

### ① Nazwa i składnia znacznika

Znacznik ma oczywiście swoją *nazwę*. W przypadku `<c:set>` *nazwą* znacznika jest *set*, natomiast *c* jest jego *przedrostkiem*. Możemy używać dowolnego przedrostka znacznika, ale jego *nazwa* zawsze pochodzi z deskryptora TLD. Składnia obejmuje takie elementy jak wymagane i opcjonalne atrybuty, możliwość definiowania ciała znacznika (a jeśli tak, to co można tam umieszczać), typy poszczególnych atrybutów oraz to, czy atrybuty mogą być wyrażeniami (czy tylko stałymi łańcuchowymi).

### ② URI biblioteki znaczników

URI jest unikatowym identyfikatorem w deskrypcorze biblioteki znaczników (ang. *Tag Library Descriptor* — *TLD*). Innymi słowy, URI jest unikatową nazwą biblioteki znaczników opisanej w pliku TLD. Właśnie ten identyfikator umieszczamy w naszych dyrektywach `taglib`. Za pomocą URI określamy na potrzeby kontenera, jak należy szukać pliku TLD w ramach danej aplikacji internetowej — na podstawie zawartości tego pliku kontener może odwzorować użytą w kodzie JSP nazwę znacznika na właściwy kod Javy wykonywany w czasie przetwarzania strony JSP z tym znacznikiem.

# Korzystanie z możliwości deskryptora TLD

Deskryptor TLD opisuje dwa główne elementy: niestandardowe znaczniki i funkcje języka EL. Mieliliśmy już okazję zetknąć się z takim deskryptorem w poprzednim rozdziale, kiedy tworzyliśmy i wywoływaliśmy funkcję rzucania kostką, ale w ramach tamtego deskryptora TLD stosowaliśmy tylko element `<function>`. Teraz przyjrzymy się bardziej złożonemu elementowi `<tag>`. Przedstawiony poniżej deskryptor wdrożenia deklaruje nie tylko wspomnianą funkcję, ale także nowy znacznik, *porada*:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<taglib xmlns="http://java.sun.com/xml/ns/j2ee" xmlns:xsi="http://www.
w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/
xml/ns/j2ee/web-jsp.taglibrary_2_0.xsd" version="2.0">
```

To jest wersja schematu XML, której używamy dla JSP 2.0. Nie musisz tego pamiętać... wystarczy że skopiujesz to do swojego elementu `<taglib>`.

```
<tlib-version>1.2</tlib-version>
```

To jest **NIEZBĘDNE** (ten znacznik, nie użyta wartość) — programista umieszcza to w deskrytorze, aby zadeklarować wersję danej biblioteki znaczników.

```
<short-name>LosoweZnaczniki</short-name>
```

```
<function>
```

```
<name>rzut</name>
```

```
<function-class>foo.RzucanieKostka</function-class>
```

```
<function-signature>int rzutKostka()</function-signature>
```

```
</function>
```

Także **NIEZBĘDNE**; głównie ze względu na stosowane narzędzia...

Funkcja języka EL, którą wykorzystywaliśmy w poprzednim rozdziale.

```
<uri>elementyLosowe</uri>
```

Unikatowa nazwa, której użyjemy w dyrektywie `taglib`!

```
<tag>
```

Opcjonalne, ale często naprawdę przydatne...

```
<description>losowa porada</description>
```

```
<name>porada</name>
```

WYMAGANE! Właśnie tę nazwę będziemy wykorzystywali w znaczniku (przykład: `<moje:porada>`).

```
<tag-class>foo.KlasaObslugiZnacznikaPorady</tag-class>
```

WYMAGANE! W ten sposób mówimy kontenerowi, co ma wywołać, kiedy ktoś użyje tego znacznika w swoim kodzie JSP.

```
<body-content>empty</body-content>
```

WYMAGANE! Ten element określa, że definiowany znacznik **NIE** może zawierać żadnych dodatkowych elementów w swoim ciele.

```
<attribute>
```

```
<name>uzytkownik</name>
```

Jeśli Twój znacznik ma atrybuty, wówczas niezbędne jest zdefiniowanie po jednym elemencie `<attribute>` dla każdego z tych atrybutów.

```
<required>true</required>
```

W ten sposób określamy, że w znaczniku **TRZEBA** umieścić atrybut "uzytkownik".

```
<rtexprvalue>true</rtexprvalue>
```

```
</attribute>
```

W tym elemencie określamy, że atrybut "uzytkownik" może być wyrażeniem wyznaczanym w czasie wykonywania (ang. *run time expression value*), a więc nie musi być stałą tańczuchową.

```
</tag>
```

```
</taglib>
```

# Stosowanie niestandardowego znacznika „porada”

"porada" jest stosunkowo prostym znacznikiem, który pobiera tylko jeden atrybut — nazwę użytkownika — i wyświetla krótki tekst losowej porady. Nasz znacznik jest co prawda na tyle prosty, że równie dobrze *moglibyśmy* w jego miejsce użyć zwykłej, tradycyjnej funkcji języka EL (ze statyczną metodą `getPorada(String nazwa)`), jednak celowo zastosowaliśmy w tym przykładzie znacznik, aby pokazać Ci, jak to wszystko działa...

## Elementy TLD dla znacznika **porada**

```
<taglib ...>
...
<uri>elementyLosowe</uri>
<tag>
 <description>losowa porada</description>
 <name>porada</name>
 <tag-class>foo.KlasaObslugiZnacznikaPorady</tag-class>
 <body-content>empty</body-content>

 <attribute>
 <name>uzytkownik</name>
 <required>true</required>
 <rtexprvalue>true</rtexprvalue>
 </attribute>
</tag>
</taglib ...>
```

To jest dokładnie ten sam znacznik, który zadeklarowaliśmy na poprzedniej stronie, ale tym razem bez dodatkowych komentarzy.

## Kod JSP wykorzystujący ten znacznik

```
<html><body>

<%@ taglib prefix="moje" uri="elementyLosowe"%>

Strona porad

<moje:porada uzytkownik="${nazwaUzytkownika}" />

</body></html>
```

Użyty tutaj identyfikator URI odpowiada elementowi zdefiniowanemu w deskrytorze TLD.

Stosowanie wyrażeń języka EL jest w tym przypadku dopuszczalne, ponieważ dla tego atrybutu ustawiliśmy wartość "true" w elemencie `<rtexprvalue>`. (Przyjmij, że atrybut "nazwaUzytkownika" już istnieje).

Każda biblioteka, której używamy na stronie, musi mieć zdefiniowaną własną dyrektywę taglib z unikatowym przedrostkiem.

Deskryptor TLD określa, że znacznik `porada` nie może zawierać ciała, zatem użyliśmy tutaj znacznika pustego (stąd konieczność jego zakończenia ukośnikiem).

## Klasa obsługująca znacznik niestandardowy

Przedstawiony poniżej kod prostej klasy obsługującej znacznik rozszerza klasę `SimpleTagSupport` (klasę, którą omówimy bardziej szczegółowo w następnym rozdziale) oraz implementuje dwie kluczowe metody: `doTag()` (odpowiedzialną za rzeczywistą obsługę znacznika) oraz `setUzytkownik()` (otrzymującą i ustawiającą wartość atrybutu).

### Klasa Javy, która wykonuje pracę za nasz znacznik

```
package foo;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.tagext.SimpleTagSupport;
import java.io.IOException;
```

Klasa `SimpleTagSupport` implementuje elementy, których będziemy potrzebowali w naszych niestandardowych znacznikach.

```
public class KlasaObslugiZnacznikaPorady extends SimpleTagSupport {
```

```
 private String uzytkownik;
```

Kontener wywołuje metodę `doTag()` w momencie, w którym strona JSP wywołuje dany znacznik za pośrednictwem nazwy zadeklarowanej w deskrytorze TLD.

```
 public void doTag() throws JspException, IOException {
 getJspContext().getOut().write("Witaj, " + uzytkownik + "
");
 getJspContext().getOut().write("Twoja porada: " + getPorada());
 }
```

```
 public void setUzytkownik(String uzytkownik) {
 this.uzytkownik=uzytkownik;
 }
```

Kontener wywołuje tę metodę, aby ustawić wartość użytą w atrybucie znacznika. Należy w tym przypadku stosować konwencje nazewnictwa znane z właściwości komponentów `JavaBean` — dzięki temu zawsze będzie wiadomo, że atrybut "uzytkownik" powinien być przekazany do metody `setUzytkownik()`.

```
 String getPorada() {
 String[] lancuchyPorad = {"Ten kolor do Ciebie nie pasuje.",
 "Powinieneś chyba iść na zwolnienie lekarskie.", "Być może powinieneś
 jeszcze raz przemyśleć decyzję o ścięciu włosów."};
 int losowa = (int) (Math.random() * lancuchyPorad.length);
 return lancuchyPorad[losowa];
 }
}
```

Nasza własna metoda wewnętrzna.



Oglądaj to!

**W klasach obsługujących znaczniki niestandardowe nie stosuje się niestandardowych nazw metod!**

Dla funkcji języka EL musieliśmy utworzyć klasę Javy z jedną metodą statyczną, nazwać tę metodę według uznania i wykorzystać deskryptor TLD do odwzorowania rzeczywistej nazwy metody (`<function-signature>`) w nazwę funkcji EL (`<name>`). Nieco inaczej jest w przypadku znaczników niestandardowych, gdzie nazwą metody ZAWSZE jest `doTag()`, co oznacza, że nigdy nie musimy jej dodatkowo deklarować w pliku deskryptora TLD. Tylko funkcje wykorzystują deklarację sygnatury metody w deskrytorze TLD!

### Zwróć uwagę na element <rtexprvalue>

Znacznik <rtexprvalue> jest szczególnie ważny, ponieważ określa, czy wartość danego atrybutu ma być wyznaczana w czasie tłumaczenia czy w czasie wykonywania aplikacji. Jeśli użyjemy w ciele tego znacznika wartości `false` lub jeśli w ogóle go nie zdefiniujemy, wymusimy stosowanie wartości atrybutu w postaci stałej łańcuchowej.

#### Jeśli zobaczysz taki zapis:

```
<attribute>
 <name>kurs</name>
 <required>true</required>
 <rtexprvalue>false</rtexprvalue>
</attribute>
```

#### LUB taki zapis:

```
<attribute>
 <name>kurs</name>
 <required>true</required>
 </attribute>
```

← Jeśli w ciele znacznika <attribute> nie zdefiniujemy znacznika <rtexprvalue>, zostanie zastosowana domyślna wartość `false`.

#### Możesz być pewien, że następujące wywołanie znacznika NIE ZADZIAŁA!

```
<html><body>
 <%@ taglib prefix="moje" uri="mojeZnaczniki"%>
 <moje:obsluzTo kurs="${aktualnykurs}" />
</body></html>
```

← NIE! To nie może być wyrażenie... w tym miejscu możemy użyć wyłącznie stałej łańcuchowej.

**P:** Nadal nie odpowiedziałeś na pytanie, skąd wiesz, jaki jest typ danego atrybutu...

**U:** Rozpocznijmy od najprostszego przykładu. Jeśli dla elementu <rtexprvalue> zdefiniowano wartość `false` (lub nie zdefiniowano żadnej wartości), wówczas JEDYNYM możliwym typem atrybutu jest stała łańcuchowa. Jeśli jednak możesz stosować wyrażenia, musisz liczyć na możliwość jednoznacznego określenia typu wyrażenia na podstawie opisu znacznika i nazwy atrybutu LUB na to, że twórca znacznika dołączył opcjonalny podelement <type> elementu <attribute>. W znaczniku <type> definiuje się kompletną nazwę klasy występującej w roli typu atrybutu. Niezależnie od tego, czy w deskrytorze TLD zadeklarowano typ atrybutu, kontener i tak oczekuje zgodności wyniku wyrażenia z typem argumentu zdefiniowanej w klasie obsługującej metody ustawiającej dany atrybut. Innymi słowy, jeśli klasa ta zawiera metodę `setPies(Pies)` dla atrybutu "pies", wówczas będzie lepiej, jeśli wartością wyrażenia użytego dla tego atrybutu będzie obiekt klasy `Pies` (lub przynajmniej coś, co można automatycznie przypisać do obiektu tego typu)!

## Ustawienia elementu `<rtexprvalue>` NIE dotyczą wyłącznie wyrażeń języka EL

W roli wartości atrybutu (lub w ciele znacznika), który dopuszcza możliwość stosowania wyrażeń czasu wykonywania, możemy używać aż **trzech** różnych rodzajów tych wyrażeń.

### 1 Wyrażenia języka EL

```
<moje:porada uzytkownik="${nazwaUzytkownika}" />
```

### 2 Wyrażenia skryptowe

```
<moje:porada uzytkownik='<%= request.getAttribute("nazwaUzytkownika") %>' />
```

### 3 Standardowe akcje `<jsp:attribute>`

```
<moje:porada>
```

```
 <jsp:attribute name="uzytkownik">${nazwaUzytkownika}</jsp:attribute>
```

```
</moje:porada>
```

W tym miejscu musimy zdefiniować wyrażenie, a nie zwykły skryptlet. Oznacza to, że na początku niezbędny jest znak „=”, oraz że nie umieszczamy średnika na końcu.

*Cóż to jest?? Myślałem, że ten znacznik nie ma ciała...*



**Standardowa akcja `<jsp:attribute>` umożliwia nam umieszczanie atrybutów w CIELE znacznika nawet wtedy, gdy w deskrytorze TLD wprost zadeklarowano ten znacznik jako „pusty”!!**

Standardowa akcja `<jsp:attribute>` jest po prostu alternatywnym sposobem definiowania atrybutów znacznika. Kluczowe znaczenie ma fakt, że dla KAŻDEGO atrybutu w tak przetwarzanym znaczniku musi istnieć JEDNA standardowa akcja `<jsp:attribute>`. Jeśli więc mamy znacznik, który zawiera W sobie (w przeciwieństwie do swojego ciała) aż trzy atrybuty, wówczas w jego ciele będziemy musieli użyć trzech znaczników `<jsp:attribute>`, po jednym dla każdego z tych atrybutów. Zwróć także uwagę na fakt, że standardowa akcja `<jsp:attribute>` ma własny atrybut **name**, w którym określamy nazwę tego atrybutu zewnętrznego znacznika, którego wartość chcemy ustawić. Więcej informacji na ten temat znajdziesz na następnej stronie...

# Co może znaleźć się w ciele znacznika?

Znacznik może zawierać ciało *tylko* wtedy, gdy element `<body-content>` dla tego znacznika nie został skonfigurowany w deskrytorze TLD z wartością **empty**. Element `<body-content>` może zawierać jedną z trzech lub czterech wartości w zależności od typu definiowanego znacznika.

`<body-content>empty</body-content>`      Znacznik **NIE** może mieć ciała.

`<body-content>scriptless</body-content>`      Znacznik **NIE** może mieć w ciele elementów skryptowych (skryptetów, wyrażeń skryptowych ani deklaracji), ale **MOŻE** zawierać tekst szablonu, wyrażenia EL oraz standardowe i niestandardowe akcje.

`<body-content>tagdependent</body-content>`      Ciało znacznika będzie traktowane jak zwykły tekst, zatem wartości użytych tam wyrażeń języka EL **NIE** zostaną wyznaczone, a znaczniki (akcje) nie zostaną wywołane.

`<body-content>JSP</body-content>`      Ciało znacznika może zawierać wszelkie elementy, które można stosować w kodzie stron JSP.

## TRZY sposoby wywoływania znacznika, który nie może mieć ciała

Każdy z przedstawionych poniżej zapisów jest dopuszczalnym sposobem wywoływania znacznika skonfigurowanego w deskrytorze TLD z elementem `<body-content>empty</body-content>`.

### ① Pusty znacznik

`<moje:porada uzytkownik="${nazwaUzytkownika}" />`

Kiedy umieszczasz ukośnik w znaczniku otwierającym, jest oczywiste, że nie będziesz stosował znacznika zamykającego.

### ② Znacznik bez niczego pomiędzy znacznikiem otwierającym i zamykającym

`<moje:porada uzytkownik="${nazwaUzytkownika}"> </moje:porada>`

Mamy co prawda znacznik otwierający i zamykający, ale nie zdefiniowaliśmy **NICZEGO** między nimi.

### ③ Znacznik pusty zawierający tylko znaczniki `<jsp:attribute>` pomiędzy znacznikiem otwierającym i zamykającym

`<moje:porada>`

`<jsp:attribute name="uzytkownik">${nazwaUzytkownika}</jsp:attribute>`

`</moje:porada>`

Znacznik `<jsp:attribute>` jest **JEDYNYM** elementem, który możemy umieścić pomiędzy znacznikiem otwierającym i zamykającym znacznika zadeklarowanego w deskrytorze TLD z elementem `<body-content>` równym `empty`! To tylko jeden ze sposobów umieszczania atrybutów w kodzie JSP, ale znaczniki `<jsp:attribute>` nie są traktowane jak „zawartość ciała”.

## Klasa obsługująca znacznik, deskryptor TLD i strona JSP

Programista klasy obsługującej znacznik tworzy deskryptor TLD, aby zarówno kontenerowi, jak i programiście JSP przekazać niezbędne informacje na temat możliwości stosowania tego znacznika. Programiście JSP nie interesuje element `<tag-class>` w deskrytorze TLD — za to odpowiedzialny jest wyłącznie kontener. Z kolei programista JSP musi się zapoznać z identyfikatorem URI, nazwą znacznika oraz jego składnią. Czy dany znacznik może mieć ciało? Czy ten atrybut musi być stałą łańcuchową czy może użyć wyrażenia? Czy ten atrybut jest opcjonalny? Jaki powinien być typ wyniku ewentualnego wyrażenia?

Deskryptor TLD możesz traktować jak **interfejs API dla znaczników niestandardowych**. Musisz przecież wiedzieć, jak wywoływać znaczniki i jakich atrybutów używać.

*Przedstawione poniżej trzy fragmenty (klasy obsługującej znacznik, deskryptora TLD oraz strony JSP) są tak naprawdę wszystkim, czego potrzebujemy do wdrożenia i uruchomienia aplikacji internetowej wykorzystującej dany znacznik.*

### Kod JSP wykorzystujący ten znacznik

```
<html><body>

<%@ taglib prefix="moje" uri="elementyLosowe"%>

Strona porad

<moje:porada uzytkownik="${nazwaUzytkownika}" />

</body></html>
```

### Klasa KlasaObslugiZnacznikaPorady

```
void doTag() {
 // logika znacznika
}

void setUser(String uzytkownik) {
 this.uzytkownik=uzytkownik;
}
```

### Plik TLD

```
<taglib ...>
...
<uri>elementyLosowe</uri>


<tag>
 <description>losowa porada</description>
 <name>porada</name>
 <tag-class>foo.KlasaObslugiZnacznikaPorady</tag-class>
 <body-content>empty</body-content>
 <attribute>
 <name>uzytkownik</name>
 <required>true</required>
 <rtexprvalue>true</rtexprvalue>
 </attribute>
</tag>
```

## Podelement <uri> elementu taglib jest tylko nazwą, a nie lokalizacją

Element <uri> w deskrytorze TLD jest unikatową nazwą danej biblioteki znaczników. To wszystko. Element ten NIE musi reprezentować rzeczywistego położenia biblioteki (np. ścieżki lub adresu URL). Po prostu musi to być nazwa — *ta sama, której później użyjemy w dyrektywie taglib*.

„Ale...” — pytasz — „jak to się ma do JSTL, dla której podajemy pełny adres URL biblioteki?”  
Dyrektywa taglib dla biblioteki JSTL ma następującą postać:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
```

 To WYGLĄDA raczej jak adres URL zasobu internetowego, choć wcale takim adresem nie jest. Ta nazwa została po prostu sformatowana w sposób identyczny jak adresy URL.

Kontener WWW w normalnych warunkach nie próbuje wysłać *ządań* na podstawie atrybutu uri dyrektywy taglib. Kontener nie musi przecież wykorzystywać tego identyfikatora w roli *elementu lokalizującego* bibliotekę! Jeśli wpiszesz ten identyfikator w swojej przeglądarce, zostaniesz przekierowany pod inny adres URL, pod którym znajdują się *informacje* na temat biblioteki JSTL. Kontener nie zwraca szczególnej uwagi na to, że akurat ten identyfikator URI jest także poprawnym adresem URL (cały ten zapis "http://..."). Takie nazewnictwo jest jedynie elementem polityki firmy Sun w zakresie zapewniania unikatowości identyfikatorów URI. Firma Sun równie dobrze mogłaby oznaczyć swoją bibliotekę JSTL identyfikatorem "java\_foo\_tags", a wszystko działałoby dokładnie tak samo.

**W praktyce znaczenie ma tylko to, czy element <uri> w deskrytorze TLD pasuje do atrybutu uri dyrektywy taglib!**

Z drugiej strony także Ty, jako programista, powinieneś stosować schemat, który zapewni unikatowość wartości <uri> dla Twoich bibliotek znaczników — na poziomie aplikacji internetowej wszystkie stosowane identyfikatory URI muszą być *unikatowe*. Nie możesz na przykład w jednej aplikacji internetowej wykorzystywać dwóch plików TLD z takim samym znacznikiem <uri>. Stosowanie domenowej konwencji nazewnictwa jest więc w tym przypadku dobrym rozwiązaniem, co wcale nie oznacza, że musisz się do niej stosować w czasie opracowywania swoich aplikacji.

Warto jeszcze wspomnieć, że *istnieje* rozwiązanie, w którym identyfikator URI może być wykorzystywany w roli lokalizacji biblioteki, ale tego typu działania w żadnym przypadku nie są zalecane — jeśli nie określisz elementu <uri> w deskrytorze TLD, kontener spróbuje użyć atrybutu uri dyrektywy taglib jako ścieżki do rzeczywistego pliku deskryptora. Trwałe kodowanie położenia deskryptora oczywiście nie jest dobrym rozwiązaniem, więc udawaj, że nigdy nie słyszałeś o tej możliwości.

Kontener szuka w deskrytorze TLD elementu <uri>, który pasuje do wartości uri użytej w dyrektywie taglib. Identyfikator URI wcale NIE musi być lokalizacją rzeczywistej klasy obsługującej znacznik!

## Kontener buduje odwzorowanie

Zanim opracowano specyfikację JSP 2.0, programista musiał sam określać odwzorowanie pomiędzy elementem `<uri>` deskryptora TLD a rzeczywistym położeniem pliku tego deskryptora. Zatem, jeśli strona JSP zawierała następującą dyrektywę `taglib`:

```
<%@ taglib prefix="moje" uri="elementyLosowe" %>
```

w deskrytorze wdrożenia (*web.xml*) programista musiał wskazać kontenerowi miejsce składowania pliku TLD z odpowiednim elementem `<uri>`. W tym celu wykorzystywano element `<taglib>` deskryptora wdrożenia.

### STARY (sprzed specyfikacji JSP 2.0) sposób odwzorowania atrybutu uri dyrektywy taglib na plik TLD

```
<web-app>
```

```
...
```

```
<jsp-config>
```

```
<taglib>
```

```
<taglib-uri>elementyLosowe</taglib-uri>
```

```
<taglib-location>/WEB-INF/mojeFunkcje.tld</taglib-location>
```

```
</taglib>
```

```
</jsp-config>
```

```
</web-app>
```

*W deskrytorze wdrożenia należało odwzorować element `<uri>` z deskryptora TLD na rzeczywistą ścieżkę do pliku tego deskryptora.*

### NOWY (zgodny ze specyfikacją JSP 2.0) sposób odwzorowania atrybutu uri dyrektywy taglib na plik TLD

#### Brak elementu `<taglib>` w deskrytorze wdrożenia!

**Kontener automatycznie buduje odwzorowanie pomiędzy plikami TLD a nazwami `<uri>`**, dzięki czemu w momencie wywołania znacznika w kodzie strony JSP kontener wie, gdzie dokładnie należy szukać deskryptora TLD opisującego ten znacznik.

Jak to możliwe? Wystarczy, że kontener przeszuka określony zbiór lokalizacji, w których mogą się znajdować pliki TLD. Kiedy wdrażasz aplikację internetową, musisz jedynie umieścić swój deskryptor TLD w miejscu, które kontener uwzględni podczas takiego przeszukiwania — kontener znajdzie ten plik i zbuduje odpowiednie odwzorowanie dla Twojej biblioteki znaczników.

Jeśli mimo to użyjesz elementu `<taglib-location>` w swoim deskrytorze wdrożenia (*web.xml*), kontener zgodny ze specyfikacją JSP 2.0 użyje właśnie zadeklarowanego tam katalogu! W praktyce, kiedy kontener rozpoczyna budowę odwzorowań `<uri>` w TLD, w *pierwszej kolejności* zagląda do naszego deskryptora wdrożenia, aby sprawdzić, czy nie ma tam żadnych elementów `<taglib>`; jeśli okaże się, że są, zostaną one wykorzystane podczas konstruowania odwzorowania.

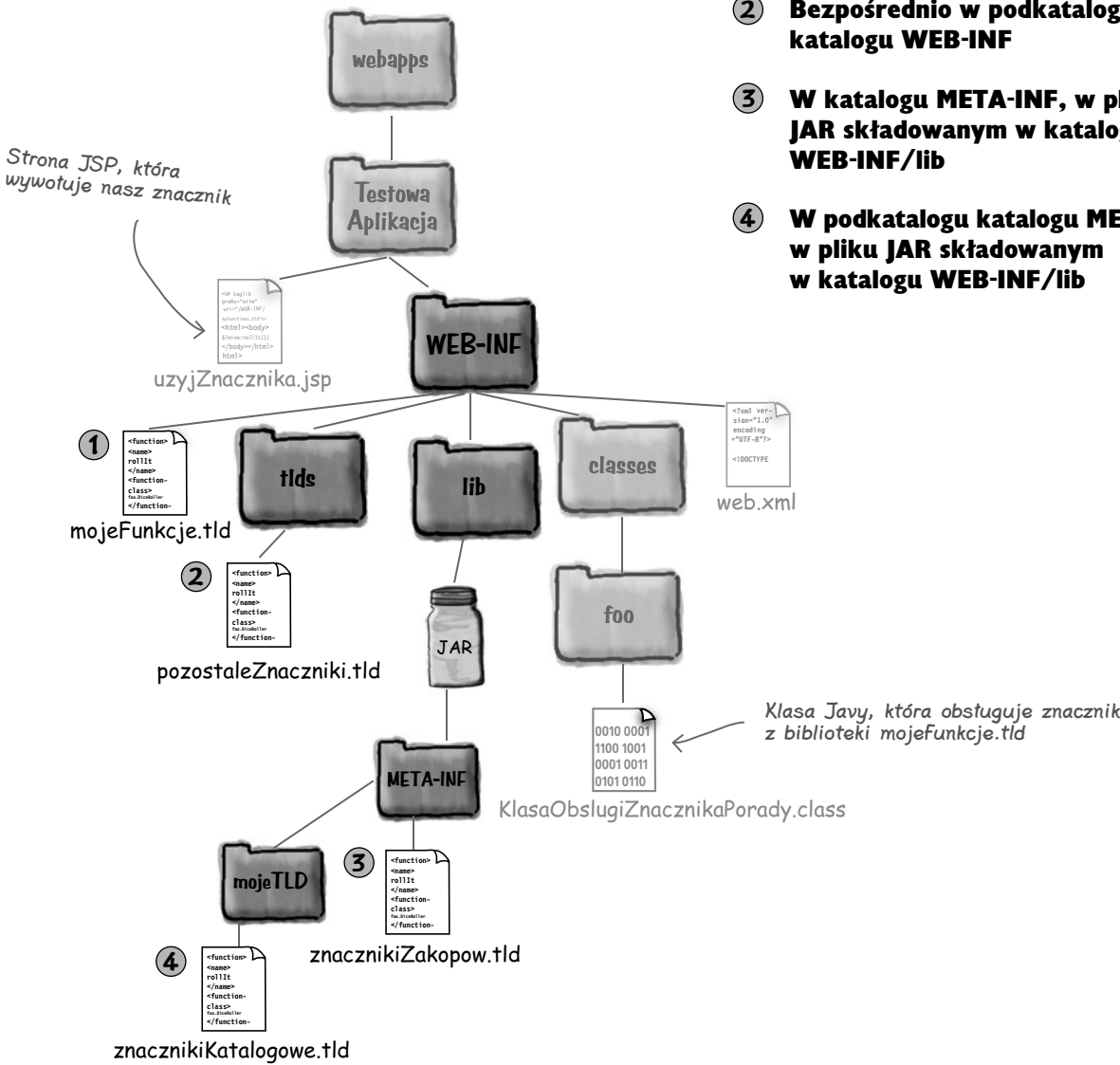
**Na egzaminie musisz wiedzieć o istnieniu elementu `<taglib-location>`, chociaż jego stosowanie nie jest wymagane przez specyfikację JSP 2.0.**

Naszym następnym krokiem jest analiza miejsc, w których kontener szuka plików TLD, oraz miejsc, które są przeglądane w poszukiwaniu zadeklarowanych w tych plikach *klas* obsługujących znaczniki.

# Cztery miejsca, w których kontener szuka plików TLD

W poszukiwaniu plików TLD kontener przegląda wiele miejsc — nie musimy więc robić niczego poza upewnieniem się, że nasze pliki deskryptorów TLD znajdują się w jednym z właściwych katalogów.

- 1 Bezpośrednio w katalogu WEB-INF
- 2 Bezpośrednio w podkatalogu katalogu WEB-INF
- 3 W katalogu META-INF, w pliku JAR składowanym w katalogu WEB-INF/lib
- 4 W podkatalogu katalogu META-INF, w pliku JAR składowanym w katalogu WEB-INF/lib



## Kiedy strona JSP wykorzystuje więcej niż jedną bibliotekę znaczników...

Jeśli chcesz używać w kodzie swojej strony JSP więcej niż jednej biblioteki znaczników, musisz zdefiniować po jednej dyrektywie taglib dla każdego deskryptora TLD. Warto przy tym pamiętać o kilku istotnych regułach:

- Upewnij się, że nazwy URI w dyrektywach taglib są unikatowe. Innymi słowy, nigdy nie stosuj więcej niż jednej dyrektywy z taką samą wartością atrybutu uri.
- NIE używaj przedrostków z listy przedrostków już zastrzeżonych. Do listy tej należą:

jsp:

jspx:

java:

javax:

servlet:

sun:

sunw:



### Zaostrz ołówek

#### Puste znaczniki

Napisz przykłady TRZECH różnych sposobów wywoływania znacznika, którego ciało musi być puste.

(Znajdź i sprawdź odpowiedzi we wcześniejszej części tego rozdziału. Nie, nie powiemy Ci, na której stronie).

- 1 \_\_\_\_\_
- 2 \_\_\_\_\_
- 3 \_\_\_\_\_



## Jakie są relacje pomiędzy stroną JSP, deskrytorem TLD a klasą atrybutu komponentu?

Wypełnij puste pola wyłącznie w oparciu o informacje widoczne w przedstawionym pliku TLD. Narysuj strzałki, aby oznaczyć związki pomiędzy poszczególnymi informacjami. Innymi słowy, dla każdego pustego pola spróbuj precyzyjnie wykazać, gdzie znalazłeś informacje niezbędne do jego wypełnienia.

### Kod JSP wykorzystujący ten znacznik

```
<html><body>

<%@ taglib prefix="moje" uri="_____" %>

Strona porad

<_____:_____="${nazwaUzytkownika}" />

</body></html>
```

### Klasa KlasaObslugiZnaczaikaPorady

```
void doTag() {
 // logika znacznika
}

void set_____ (String uzytkownik) {
 this.uzytkownik=uzytkownik;
}
```

### Plik TLD

```
<taglib ...>
...
<uri>elementyLosowe</uri>

<tag>
 <description>losowa porada</description>
 <name>porada</name>
 <tag-class>foo.KlasaObslugiZnaczaikaPorady</tag-class>
 <body-content>empty</body-content>
 <attribute>
 <name>uzytkownik</name>
 <required>true</required>
 <rtexprvalue>_____</rtexprvalue>
 </attribute>
</tag>
```



Zaostrz ołówek

## Sprawdź, co pamiętasz o znacznikach ODPOWIEDZI

- ❶ Wpisz nazwę opcjonalnego atrybutu.

Atrybut nazywający  
zmienną licznika pętli.

```
<c:forEach var="film" items="${listaFilmow}" ="foo" >
 ${film}
</c:forEach>
```

- ❷ Wpisz brakującą nazwę atrybutu.

```
<c:if ="${preferencjeUzytkownika=='bezpieczenstwo'}" >
 Może po prostu powinienes wybrać spacer...
</c:if>
```

Znacznik <c:set> musi mieć wartość, ale  
możesz zdecydować o umieszczeniu tej  
wartości w ciele tego znacznika (zamiast  
w postaci odpowiedniego atrybutu).

- ❸ Wpisz brakującą nazwę atrybutu.

```
<c:set var="poziomUzytkownika" scope="session" ="foo" />
```

- ❹ Wpisz brakujące nazwy znaczników (dwóch różnych typów) oraz brakującą nazwę atrybutu.

```
<c:choose>
 <c: ="${preferencjeUzytkownika == 'osiagi'}">
 Teraz możesz efektywnie hamować, nawet jeśli jeździsz jak szalony...
 </c: >
 <c: >
 Nasze hamulce są najlepsze.
 </c: >
</c:choose>
```

Znacznik <c:otherwise> jest opcjonalny.



**Jakie są relacje pomiędzy stroną JSP, deskryptorem TLD a klasą atrybutu komponentu?**

**ODPOWIEDZI**

### Kod JSP wykorzystujący ten znacznik

```
<html><body>

<%@ taglib prefix="moje" uri=" elementyLosowe "%>

Strona porad

< moje : porada uzytkownik ="${nazwaUzytkownika}" />

</body></html>
```

### Klasa **KlasaObslugiZnacznikaPorady**

```
void doTag() {
 // logika znacznika
}

void set Uzytkownik (String uzytkownik) {
 this.uzytkownik=uzytkownik;
}
```

### Plik TLD

```
<taglib ...>
...
<uri>elementyLosowe</uri>

<tag>
 <description>losowa porada</description>
 <name>porada</name>
 <tag-class>foo.KlasaObslugiZnacznikaPorady</tag-class>
 <body-content>empty</body-content>
 <attribute>
 <name>uzytkownik</name>
 <required>true</required>
 <rtexprvalue> true </rtexprvalue>
 </attribute>
</tag>
```



- 
- 1 Które zdanie na temat plików TLD jest prawdziwe?
- ☐ A. Pliki TLD mogą być umieszczane w dowolnym podkatalogu wewnątrz **WEB-INF**.
  - ☐ B. Pliki TLD służą do konfigurowania atrybutów środowiskowych JSP, takich jak **scripting-invalid**.
  - ☐ C. Pliki TLD mogą być umieszczane w pliku WAR wewnątrz katalogu **META-INF**.
  - ☐ D. Pliki TLD mogą deklarować zarówno proste, jak i klasyczne znaczniki, ale NIE można za ich pomocą deklarować plików znaczników.
- 
- 2 Zakładając, że stosujemy standardowe konwencje nazywania przedrostków biblioteki JSTL, których znaczników tej biblioteki użyłbyś do iteracyjnego przeglądania kolekcji obiektów? (Zaznacz wszystkie prawidłowe opcje).
- ☐ A. **<x:forEach>**
  - ☐ B. **<c:iterate>**
  - ☐ C. **<c:forEach>**
  - ☐ D. **<c:forEachTokens>**
  - ☐ E. **<logic:iterate>**
  - ☐ F. **<logic:forEach>**

- 2 Strona JSP zawiera dyrektywę **taglib**, której atrybut **uri** ma wartość **mojeZnaczniki**. Który z przedstawionych poniżej elementów deskryptora wdrożenia definiuje odpowiedni deskryptor TLD?

- ☐ A. **<taglib>**  
    **<uri>mojeZnaczniki</uri>**  
    **<location>/WEB-INF/mojeZnaczniki.tld</location>**  
    **</taglib>**
- ☐ B. **<taglib>**  
    **<uri>mojeZnaczniki</uri>**  
    **<tld-location>/WEB-INF/mojeZnaczniki.tld</tld-location>**  
    **</taglib>**
- ☐ C. **<taglib>**  
    **<tld-uri>mojeZnaczniki</tld-uri>**  
    **<tld-location>/WEB-INF/mojeZnaczniki.tld</tld-location>**  
    **</taglib>**
- ☐ D. **<taglib>**  
    **<taglib-uri>mojeZnaczniki</taglib-uri>**  
    **<taglib-location>/WEB-INF/mojeZnaczniki.tld</taglib-location>**  
    **</taglib>**

- 4 Komponent JavaBean **Osoba** zawiera właściwość nazwaną **adres**. Wartością tej właściwości jest inny komponent JavaBean **Adres** z następującymi właściwościami łańcuchowymi: **ulica1**, **ulica2**, **miasto**, **województwo** i **kod**. Serwlet kontrolera tworzy atrybut zasięgu sesji nazwany **klient**, który jest egzemplarzem komponentu **Osoba**.

Która z przedstawionych poniżej struktur JSP ustawi we właściwości **miasto** atrybutu **klient** wartość parametru żądania **miasto**? (Zaznacz wszystkie prawidłowe opcje).

- ☐ A. **`${sessionScope.klient.adres.miasto = param.miasto}`**
- ☐ B. **`<c:set target="${sessionScope.klient.adres}"  
property="miasto" value="${param.miasto}" />`**
- ☐ C. **`<c:set scope="session" var="${klient.adres}"  
property="miasto" value="${param.miasto}" />`**
- ☐ D. **`<c:set target="${sessionScope.klient.adres}"  
property="miasto">  
  ${param.miasto}  
</c:set>`**

5 Które z przedstawionych kombinacji elementów **<body-content>** w deskrytorze TLD są poprawne dla poniższego fragmentu kodu JSP? (Zaznacz wszystkie prawidłowe opcje).

```
11. <moje:znacznik1>
12. <moje:znacznik2 a="47" />
13. <% a = 420; %>
14. <moje:znacznik3>
15. value = ${a}
16. </moje:znacznik3>
17. </moje:znacznik1>
```

- ☐ A. <body-content> dla znacznika znacznik1: **empty**  
 <body-content> dla znacznika znacznik2: **JSP**  
 <body-content> dla znacznika znacznik3: **scriptless**
- ☐ B. <body-content> dla znacznika znacznik1: **JSP**  
 <body-content> dla znacznika znacznik2: **empty**  
 <body-content> dla znacznika znacznik3: **scriptless**
- ☐ C. <body-content> dla znacznika znacznik1: **JSP**  
 <body-content> dla znacznika znacznik2: **JSP**  
 <body-content> dla znacznika znacznik3: **JSP**
- ☐ D. <body-content> dla znacznika znacznik1: **scriptless**  
 <body-content> dla znacznika znacznik2: **JSP**  
 <body-content> dla znacznika znacznik3: **JSP**
- ☐ E. <body-content> dla znacznika znacznik1: **JSP**  
 <body-content> dla znacznika znacznik2: **scriptless**  
 <body-content> dla znacznika znacznik3: **scriptless**

6 Zakładając, że istnieją odpowiednie dyrektywy **taglib**, określ, które z poniższych przykładów reprezentują prawidłowe zastosowania niestandardowych znaczników. (Zaznacz wszystkie prawidłowe opcje).

- ☐ A. **<foo:bar />**
- ☐ B. **<moje:znacznik></moje:znacznik>**
- ☐ C. **<mojznacznik value="x" />**
- ☐ D. **<c:out value="x" />**
- ☐ E. **<jsp:setProperty name="a" property="b" value="c" />**

**7** Mamy dany następujący kod skryptletu:

```
11. <select name='idStylu'>
12. <% StylPiwa[] style = uslugapiwna.getStyle();
13. for (int=0; i < style.length; i++) {
14. StylPiwa styl = style[i]; %>
15. <option value='<%= styl.getIdObiektu() %>'>
16. <%= styl.getTytul() %>
17. </option>
18. <% } %>
19. </select>
```

Który z przedstawionych poniżej fragmentów kodu JSTL wygeneruje taki sam wynik?

- ☐ A. 

```
<select name='idStylu'>
 <c:for array='${uslugapiwna.style}'>
 <option value='${element.idObiektu}'>${element.tytul}</option>
 </c:for>
</select>
```
- ☐ B. 

```
<select name='idStylu'>
 <c:forEach var='styl' items='${uslugapiwna.style}'>
 <option value='${styl.idObiektu}'>${styl.tytul}</option>
 </c:forEach>
</select>
```
- ☐ C. 

```
<select name='idStylu'>
 <c:for var='styl' array='${uslugapiwna.style}'>
 <option value='${styl.idObiektu}'>${styl.tytul}</option>
 </c:for>
</select>
```
- ☐ D. 

```
<select name='idStylu'>
 <c:forEach var='styl' array='${uslugapiwna.style}'>
 <option value='${styl.idObiektu}'>${styl.tytul}</option>
 </c:forEach>
</select>
```



# BAR KAWOWY

## Egzamin próbny — odpowiedzi

- 1 Które zdanie na temat plików TLD jest prawdziwe? (Specyfikacja JSP 2.0, str. 3 – 16 oraz 1 – 160).
- ☒ A. Pliki TLD mogą być umieszczane w dowolnym podkatalogu wewnątrz **WEB-INF**.
    - Odpowiedź B jest niepoprawna, ponieważ pliki TLD konfiguruje klasy obsługujące znaczniki, a nie środowisko JSP.
  - ☐ B. Pliki TLD służą do konfigurowania atrybutów środowiskowych JSP, takich jak **scripting-invalid**.
  - ☐ C. Pliki TLD mogą być umieszczane w pliku WAR wewnątrz katalogu **META-INF**.
    - Odpowiedź C jest niepoprawna, ponieważ pliki TLD nie są rozpoznawane w pliku WAR w katalogu META-INF.
  - ☐ D. Pliki TLD mogą deklarować zarówno proste, jak i klasyczne znaczniki, ale NIE można za ich pomocą deklarować plików znaczników.
    - Odpowiedź D jest niepoprawna, ponieważ pliki znaczników mogą być deklarowane w deskrypcorze TLD (co jednak jest rzadką praktyką).
- 
- 2 Zakładając, że stosujemy standardowe konwencje nazywania przedrostków biblioteki JSTL, których znaczników tej biblioteki użyłbyś do iteracyjnego przeglądania kolekcji obiektów? (Zaznacz wszystkie prawidłowe opcje). (Specyfikacja JSTL 1.1, str. 42).
- ☐ A. **<x:forEach>**
    - Odpowiedź A jest niepoprawna, ponieważ zastosowano tutaj znacznik wykorzystywany jest do iteracyjnego przeszukiwania wyrażenia XPath.
  - ☐ B. **<c:iterate>**
    - Odpowiedź B jest niepoprawna, ponieważ taki znacznik w ogóle nie istnieje.
  - ☒ C. **<c:forEach>**
    - Odpowiedź D jest niepoprawna, ponieważ znacznik ten służy do iteracyjnego przeglądania elementów w ramach pojedynczego łańcucha.
  - ☐ D. **<c:forEach>**
  - ☐ E. **<logic:iterate>**
    - Odpowiedzi E i F są niepoprawne, ponieważ "logic" nie jest standardowym przedrostkiem biblioteki JSTL (przedrostek "logic" jest zazwyczaj stosowany przez znaczniki należące do pakietu Jakarta Struts).
  - ☐ F. **<logic:forEach>**

- 3 Strona JSP zawiera dyrektywę **taglib**, której atrybut **uri** ma wartość **mojeZnaczniki**. Który z przedstawionych poniżej elementów deskryptora wdrożenia definiuje odpowiedni deskryptor TLD?

(Specyfikacja JSP 2.0, str. 3 – 12, 13).

- ☐ A. **<taglib>**  
     **<uri>mojeZnaczniki</uri>**  
     **<location>/WEB-INF/mojeZnaczniki.tld</location>**  
**</taglib>**
- ☐ B. **<taglib>**  
     **<uri>mojeZnaczniki</uri>**  
     **<tld-location>/WEB-INF/mojeZnaczniki.tld</tld-location>**  
**</taglib>**
- ☐ C. **<taglib>**  
     **<tld-uri>mojeZnaczniki</tld-uri>**  
     **<tld-location>/WEB-INF/mojeZnaczniki.tld</tld-location>**  
**</taglib>**
- ☒ D. **<taglib>**  
     **<taglib-uri>mojeZnaczniki</taglib-uri>**  
     **<taglib-location>/WEB-INF/mojeZnaczniki.tld</taglib-location>**  
**</taglib>**

— Odpowiedź D określa prawidłowe elementy znacznika.

- 4 Komponent JavaBean **Osoba** zawiera właściwość nazwaną **adres**. Wartością tej właściwości jest inny komponent JavaBean **Adres** z następującymi właściwościami łańcuchowymi: **ulica1**, **ulica2**, **miasto**, **województwo** i **kod**. Serwlet kontrolera tworzy atrybut zasięgu sesji nazwany **klient**, który jest egzemplarzem komponentu **Osoba**.

(Specyfikacja JSTL 1.1, str. 4 – 28).

Która z przedstawionych poniżej struktur JSP ustawi we właściwości **miasto** atrybutu **klient** wartość parametru żądania **miasto**? (Zaznacz wszystkie prawidłowe opcje).

- ☐ A. **`\${sessionScope.klient.adres.miasto = param.miasto}**
- ☒ B. **<c:set target="\${sessionScope.klient.adres}"**  
     **property="miasto" value="\${param.miasto}" />**
- ☐ C. **<c:set scope="session" var="\${klient.adres}"**  
     **property="miasto" value="\${param.miasto}" />**
- ☒ D. **<c:set target="\${sessionScope.klient.adres}"**  
     **property="miasto"**  
     **\${param.miasto}**  
**</c:set>**

— Odpowiedź A jest niepoprawna, ponieważ język wyrażeń EL nie zezwala na stosowanie operacji przypisania.

— Odpowiedź C jest niepoprawna, ponieważ atrybutowi **var** nie można przypisywać wartości wyznaczonej w czasie wykonywania aplikacji, a sam atrybut **var** nie może występować łącznie z atrybutem **property**.

- 5 Które z przedstawionych kombinacji elementów **<body-content>** w deskrytorze TLD są poprawne dla poniższego fragmentu kodu JSP? (Zaznacz wszystkie prawidłowe opcje). (Specyfikacja JSP 2.0, dodatek JSP.C, w szczególności str. 3 – 21 oraz 3 – 30).

```

11. <moje:znacznik1>
12. <moje:znacznik2 a="47" />
13. <% a = 420; %>
14. <moje:znacznik3>
15. value = ${a}
16. </moje:znacznik3>
17. </moje:znacznik1>

```

— znacznik1 zawiera kod skryptu, zatem w elemencie **<body-content>** musi mieć zdefiniowaną przynajmniej wartość 'JSP'. znacznik2 jest w przedstawionym fragmencie znacznikiem pustym, a więc w elemencie **<body-content>** mógłby równie dobrze mieć zdefiniowaną wartość 'JSP' lub 'scriptless'. znacznik3 nie zawiera elementów skryptowych, zatem w elemencie **<body-content>** może mieć zdefiniowaną albo wartość 'JSP', albo wartość 'scriptless'.

- ☐ A. **<body-content>** dla znacznika znacznik1: **empty**  
**<body-content>** dla znacznika znacznik2: **JSP**  
**<body-content>** dla znacznika znacznik3: **scriptless**
- ☒ B. **<body-content>** dla znacznika znacznik1: **JSP**  
**<body-content>** dla znacznika znacznik2: **empty**  
**<body-content>** dla znacznika znacznik3: **scriptless**
- ☒ C. **<body-content>** dla znacznika znacznik1: **JSP**  
**<body-content>** dla znacznika znacznik2: **JSP**  
**<body-content>** dla znacznika znacznik3: **JSP**
- ☐ D. **<body-content>** dla znacznika znacznik1: **scriptless**  
**<body-content>** dla znacznika znacznik2: **JSP**  
**<body-content>** dla znacznika znacznik3: **JSP**
- ☒ E. **<body-content>** dla znacznika znacznik1: **JSP**  
**<body-content>** dla znacznika znacznik2: **scriptless**  
**<body-content>** dla znacznika znacznik3: **scriptless**
- Odpowiedź A jest niepoprawna, ponieważ znacznik1 nie może być 'empty'.  
 — Odpowiedź D jest niepoprawna, ponieważ znacznik1 nie może być 'scriptless'.

- 6 Zakładając, że istnieją odpowiednie dyrektywy **taglib**, określ, które z poniższych przykładów reprezentują prawidłowe zastosowania niestandardowych znaczników. (Zaznacz wszystkie prawidłowe opcje). (Specyfikacja JSP 2.0, rozdział 7.).

- ☒ A. **<foo:bar />**
- ☒ B. **<moje:znacznik></moje:znacznik>**
- ☐ C. **<mojznacznik value="x" />** — Odpowiedź C jest niepoprawna, ponieważ brakuje w niej przedrostka.
- ☒ D. **<c:out value="x" />**
- ☐ E. **<jsp:setProperty name="a" property="b" value="c" />** — Odpowiedź E jest niepoprawna, ponieważ reprezentuje przykład standardowej akcji JSP, nie akcji niestandardowej.

7 Mamy dany następujący kod skryptletu:

(Specyfikacja JSTL 1.1,  
str. 6 – 48).

```
11. <select name='idStylu'>
12. <% StylPiwa[] style = uslugapiwna.getStyle();
13. for (int=0; i < style.length; i++) {
14. StylPiwa styl = style[i]; %>
15. <option value='<%= styl.getIdObiektu() %>'>
16. <%= styl.getTytul() %>
17. </option>
18. <% } %>
19. </select>
```

Który z przedstawionych poniżej fragmentów kodu JSTL wygeneruje taki sam wynik?

☐ A. 

```
<select name='idStylu'>
 <c:for array='${uslugapiwna.style}'>
 <option value='${element.idObiektu}'>${element.tytul}</option>
 </c:for>
</select>
```

☒ B. 

```
<select name='idStylu'>
 <c:forEach var='styl' items='${uslugapiwna.style}'>
 <option value='${styl.idObiektu}'>${styl.tytul}</option>
 </c:forEach>
</select>
```

— Odpowiedź B jest prawidłowa,  
ponieważ wykorzystuje prawidłowe nazwy  
znaczników i atrybutów biblioteki JSTL.

☐ C. 

```
<select name='idStylu'>
 <c:for var='styl' array='${uslugapiwna.style}'>
 <option value='${styl.idObiektu}'>${styl.tytul}</option>
 </c:for>
</select>
```

☐ D. 

```
<select name='idStylu'>
 <c:forEach var='styl' array='${uslugapiwna.style}'>
 <option value='${styl.idObiektu}'>${styl.tytul}</option>
 </c:forEach>
</select>
```

## 10. Tworzenie znaczników niestandardowych

### ***Kiedy JSTL nie wystarcza...***



**Czasami JSTL i standardowe akcje nie wystarczają.** Kiedy trzeba zrobić coś niestandardowego, a nie chcesz uciekać się do stosowania skryptu, możesz stworzyć własne klasy obsługi *znaczników*. Dzięki temu projektanci stron będą mogli używać w budowanych stronach Twoich znaczników, a całą „czarną robotę” będzie, w niewidoczny sposób, realizować Twoja klasa obsługi znacznika. Niemniej jednak czeka Cię dużo nauki, gdyż własne niestandardowe znaczniki można tworzyć na trzy różne sposoby. Dwa spośród nich zostały wprowadzone w JSP 2.0, aby ułatwić nam życie (chodzi o *znaczniki proste*, ang. *Simple Tags*, oraz *pliki znaczników*, ang. *Tag Files*). Mimo to warto poznać sposób tworzenia znaczników „klasycznych” na wypadek, gdybyś znalazł się w tej nieprawdopodobnie rzadkiej sytuacji, w której żadna z dwóch pozostałych metod nie zapewni Ci niezbędnych możliwości. Znaczniki niestandardowe dają ogromne, niemal nieskończone możliwości, jeśli tylko nauczysz się nimi władać...



### Tworzenie biblioteki znaczników niestandardowych

- 10.1.** Proszę opisać semantykę modelu zdarzeń „klasycznych” znaczników niestandardowych dla przypadków wywołania metod zdarzeń (na przykład: `doStartTag()`, `doAfterBody()` oraz `doEndTag()`) i wyjaśnić znaczenie wartości wynikowych zwracanych przez każdą z tych metod. Dodatkowo proszę napisać kod klasy obsługi znacznika.
- 10.2.** Używając metod klasy `PageContext`, proszę napisać kod procedur obsługi znacznika, który będzie odwoływał się do domyślnych zmiennych JSP oraz atrybutów aplikacji internetowej.
- 10.3.** Dysponując pewnym scenariuszem, proszę napisać kod procedury obsługi znacznika, który będzie odwoływał się do znacznika macierzystego oraz do dowolnego znacznika-przodka.
- 10.4.** Proszę opisać semantykę modelu zdarzeń „prostego” znacznika niestandardowego w przypadku wywołania metody zdarzenia (`doTag()`), napisać klasę obsługi takiego znacznika i opisać, jakim ograniczeniom podlega jego zawartość (umieszczana w kodzie strony JSP).
- 10.5.** Proszę opisać semantykę modelu pliku znacznika, strukturę aplikacji WWW dla takich plików, napisać kod klasy obsługi takiego znacznika oraz opisać ograniczenia, jakim podlega zawartość umieszczana w takim znaczniku.

### Uwagi wyjaśniające:

*Chociaż w punkcie 10.1 nie wspomniano wprost o metodach cyklu życia skojarzonych z interfejsami `BodyTag` (`doInitBody()` oraz `setBodyContext()`), na egzaminie może się pojawić pytanie na ich temat! W niniejszym rozdziale zamieściliśmy wszystkie niezbędne dane dotyczące klasycznych znaczników niestandardowych, w tym informacje, które nie wynikają w jawny sposób z celu 10.1.*

*Cel 10.2 (interfejs API klasy `PageContext`) został w niniejszym rozdziale opisany bardzo pobieżnie, gdyż niemal wszystkie niezbędne informacje na jego temat zostały już zamieszczone w poprzednich rozdziałach książki. Ten cel niemal w całości dotyczy wykorzystania klasy `PageContext` do operowania na zmiennych domyślnych oraz atrybutach o określonym zasięgu (oba te zagadnienia zostały opisane szczegółowo w rozdziale 8., „Strony bezskryptowe”, choć w tym rozdziale zamieściliśmy ich krótkie podsumowanie i przypomnienie).*

Podoba mi się idea posiadania fragmentów treści, których można wielokrotnie używać, jednak znaczniki `<jsp:include>` oraz `<c:import>` nie są doskonałe. Nie ma wypracowanej standardowej struktury katalogów, w których są umieszczane dołączane pliki, kod JSP jest trudny do analizy, a pomysł tworzenia nowych parametrów żądania w celu przekazania informacji do dołączanego pliku wydaje mi się chybiony...



## Dołączanie i importowanie może być kłopotliwe

Znaczniki `<jsp:include>` oraz `<c:import>` pozwalają na dynamiczne dołączanie do strony fragmentów treści. Istnieje nawet możliwość określania sposobu działania takiego dołączanego pliku dzięki przekazywaniu do niego informacji pod postacią parametrów żądania.

Jasne, że to rozwiązanie doskonale działa. Ale czy naprawdę musimy tworzyć nowe *parametry żądania* tylko po to, by przekazać do dołączanego pliku jakieś informacje niezbędne do jego dostosowania?

Z drugiej strony, czy parametry nie mają reprezentować informacji przesyłanych *od klienta* i stanowiących części żądania? Choć można sobie wyobrazić uzasadnione powody do dodawania lub modyfikacji parametrów żądania przez aplikację, to jednak używanie ich wyłącznie w celu przesłania ich do dołączanego pliku nie jest rozwiązaniem eleganckim.

Aż do momentu wprowadzenia standardu JSP 2.0 nie było żadnego standardowego sposobu rozmieszczania dołączanych plików na serwerze — można je było umieszczać w absolutnie dowolnym miejscu aplikacji. Dodatkowo kod JSP zawierający znaczniki `<jsp:include>` lub `<c:import>` nie jest łatwy do przeglądania i analizy. Czy nie byłoby lepiej, gdyby sam znacznik dawał nam pewne wskazówki dotyczące dołączanych treści? Czy nie można by z powodzeniem używać takich znaczników jak:

`<x:naglowekLogo>` lub `<x:pasekNawigacyjny>`

**Pewnie już wiesz, dokąd to rozumowanie zmierza...**

# Pliki znaczników — podobnie jak znacznik include, tylko lepiej

Pliki znaczników pozwalają na dołączanie treści przy wykorzystaniu znaczników niestandardowych, a nie ogólnych znaczników `<jsp:include>` lub `<c:import>`. Pliki znaczników można sobie wyobrażać jako „uproszczone procedury obsługi znaczników”. Dzięki nim autorzy stron mogą tworzyć i używać znaczników niestandardowych bez konieczności pisania skomplikowanych klas obsługi znaczników. Nie zmienia to jednak faktu, iż pliki znaczników są jedynie lekko ulepszonym sposobem *dołączania* treści.

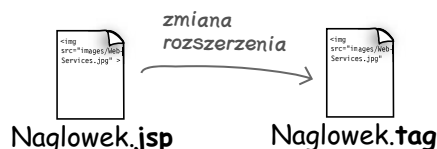
## Najprostszy sposób tworzenia i zastosowania pliku znacznika

### ① Zmień rozszerzenie dołączanego pliku (na przykład: „Naglowek.jsp”) na .tag.

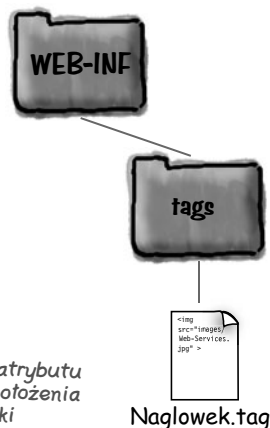
```


```

Oto jest cała zawartość pliku...  
Pamiętaj, że usunęliśmy z niego  
otwierające i zamykające znaczniki  
`<html>` oraz `<body>`, tak by nie zostały  
one powtórzone w kodzie strony JSP.



### ② Umieść plik znacznika („Naglowek.tag”) w katalogu „tags” znajdującym się w katalogu „WEB-INF”.



### ③ W kodzie strony JSP umieść dyrektywę taglib (zawierającą atrybut tagdir), a następnie wywołaj znacznik.

```
<%@ taglib prefix="mojeZnaczniki" tagdir="/WEB-INF/tags" %>
```

```
<html><body>
```

```
<mojeZnaczniki:Naglowek/>
```

Nazwa znacznika po prostu odpowiada  
nazwie pliku! (oczywiście bez  
rozszerzenia .tag).

W dyrektywie taglib, zamiast atrybutu  
uri używanego do określenia położenia  
pliku TLD opisującego biblioteki  
znaczników, umieść atrybut tagdir.

```
Witamy na naszej witrynie.
```

```
</body></html>
```

A zatem, zamiast:

```
<jsp:include page="Naglowek.jsp"/>
```

mamy:

```
<mojeZnaczniki:Naglowek/>
```

## A w jaki sposób można przesyłać parametry?

Kiedy dołączaliśmy plik z wykorzystaniem znacznika `<jsp:include>`, informacje przekazywane do dołączanego pliku podawaliśmy w znaczniku `<jsp:param>` umieszczanym wewnątrz znacznika `<jsp:include>`. Poniżej zamieściliśmy krótkie przypomnienie, które ma na celu odświeżenie Twojej pamięci:

### Stary sposób: Plik dołączany wykorzystujący parametry (podawane w znaczniku `<jsp:param>` wywołującej strony JSP)

```


${param.podTytul}
```

Ponownie mamy do czynienia z CAŁYM dołączanym plikiem, nie jego fragmentem.

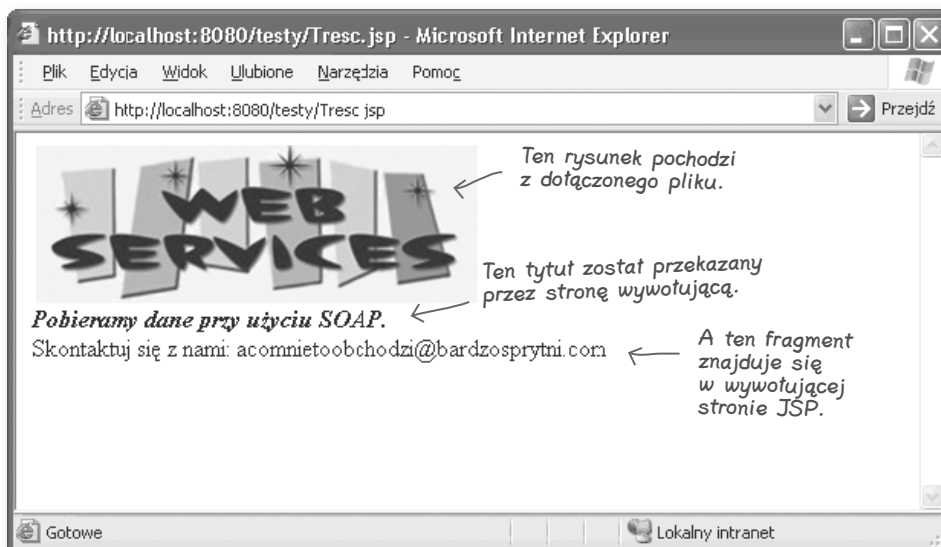
### Stary sposób: Plik JSP zawierający znaczniki `<jsp:include>` oraz `<jsp:param>`

```
<html><body>
<jsp:include page="Naglowek.jsp">
 <jsp:param name="podTytul" value="Pobieramy dane przy użyciu SOAP." />
</jsp:include>

Skontaktuj się z nami: ${initParam.adresEmail}
</body></html>
```

Ten znacznik definiuje nowy parametr żądania, którego strona dołączana może używać w taki sam sposób jak wszelkich INNYCH parametrów

### A oto wynik:



# Przekazując informacje do plików znaczników, nie używamy parametrów żądania, tylko atributów znacznika!

Aby skorzystać z pliku znacznika, umieszczamy w kodzie strony JSP odpowiedni znacznik. A, jak wiadomo, znaczniki mogą mieć atrybuty. W tej sytuacji trudno się dziwić, że twórca pliku znacznika może chcieć umieszczać w znaczniku atrybuty... atrybuty wysyłane do pliku znacznika.

## Umieszczanie znacznika w kodzie JSP

Wcześniej (kiedy parametry żądania ustawiano za pomocą znacznika `<jsp:param>`)

```
<jsp:include page="Naglowek.jsp">
 <jsp:param name="podTytuł" value="Pobieramy dane przy użyciu SOAP." />
</jsp:include>
```

Teraz (przy wykorzystaniu znacznika i atrybutów)

```
<mojeZnaczniki:Naglowek podTytuł="Pobieramy dane przy użyciu SOAP." />
```

## Wykorzystanie atrybutu w pliku znacznika

Wcześniej (gdy stosowany był parametr żądania)

```
${param.podTytuł}
```

Teraz (gdy stosowany jest znacznik z atrybutem)

```
${podTytuł}
```

Ten kod jest umieszczany w pliku znacznika  
(czyli w dotaczanym pliku).



Oglądaj to!

**Wszystkie atrybuty znacznika istnieją tylko w zasięgu samego ZNACZNIKA.  
Tak — wyłącznie w zasięgu tego znacznika. Wraz z odpowiednim znacznikiem  
zamykającym kończy się zasięg zdefiniowanych w nim atrybutów!**

Musisz to dobrze zrozumieć — wartości określone przy użyciu znaczników `<jsp:include>` `<jsp:param>` są przekazywane jako parametry żądania, a to nie to samo co atrybuty zasięgu żądania. Pamiętaj! Aplikacja traktuje parę nazwa-wartość zdefiniowaną przy użyciu znacznika `<jsp:param>` zupełnie tak jak pary przesyłane z formularzy. Zresztą jest to jeden z powodów, dla których nie lubimy stosować tego rozwiązania — wartość, którą chcieliśmy przekazać wyłącznie do dołączonego pliku, staje się dostępna dla każdego komponentu aplikacji zaangażowanego w obsługę danego żądania (na przykład wszystkich serwetów i stron JSP, do których to żądanie zostanie przekierowane). Dużo bardziej eleganckim aspektem atrybutów znaczników dla plików znaczników jest ich ograniczony zasięg. Upewnij się, że dobrze rozumiesz konsekwencje tego rozwiązania. Na przykład poniższy fragment kodu NIE będzie działał:

```
<%@ taglib prefix="mojeZnaczniki" tagdir="/WEB-INF/tags" %>
<html><body>
<mojeZnaczniki:Naglowek podTytuł="Pobieramy dane przy użyciu SOAP." />


```

```
${podTytuł}
```

```
</body></html>
```

To nie zadziała! Atrybut  
jest poza zasięgiem

Chwileczkę... coś tu nie gra. A skąd osoba tworząca stronę JSP ma wiedzieć o istnieniu atrybutu znacznika? Gdzie znajduje się deskryptor (TLD) opisujący typ atrybutu?



## Czy atrybuty znaczników nie są deklarowane w TLD?

Atrybuty znaczników niestandardowych, w tym także znaczników wchodzących w skład biblioteki JSTL, są definiowane w deskrytorze TLD. Pamiętasz? Oto TLD niestandardowego znacznika `<moje:porada>`, który zdefiniowaliśmy w poprzednim rozdziale:

```
<tag>
 <description>losowa porada</description>
 <name>porada</name>
 <tag-class>foo.KlasaObslugiZnacznikaPorady</tag-class>
 <body-content>empty</body-content>

 <attribute>
 <name>uzytkownik</name>
 <required>true</required>
 <rtexprvalue>true</rtexprvalue>
 </attribute>
</tag>
```

Jak zatem widać, są pewne sprawy, o których osoba używająca znacznika powinna wiedzieć. Jak się nazywa atrybut? Czy jest on wymagany czy też można go pominąć? Czy atrybut może być wyrażeniem, czy wyłącznie stałą łańcuchową?

Jednak w odróżnieniu od atrybutów *znaczników niestandardowych*, które są definiowane w TLD, informacje o atrybutach *plików znaczników* nie są umieszczane w TLD.

A to oznacza, że mamy problem — skąd osoba tworząca stronę ma *wiedzieć*, jakie atrybuty można umieścić w znaczniku, ewentualnie jakich atrybutów ten znacznik wymaga. *Odwróć stronę...*

# W plikach znaczników używamy dyrektywy attribute

Istnieje nowy, wspaniały typ dyrektywy, która jest ściśle związana z plikami znaczników. Omawiana dyrektywa nie może być używana w żadnych innych plikach. Pod wieloma względami przypomina podelement `<attribute>` umieszczany w sekcji `<tag>` deskryptora TLD opisującego postać znacznika niestandardowego.

## Wewnątrz pliku znacznika (Naglowek.tag)

```
<%@ attribute name="podTytuł" required="true" rtexprvalue="true" %>

${podTytuł}
```

To oznacza, że atrybut jest wymagany.

Wartością atrybutu może być stała tańcuchowa LUB wyrażenie.

## Wewnątrz strony JSP używającej znacznika

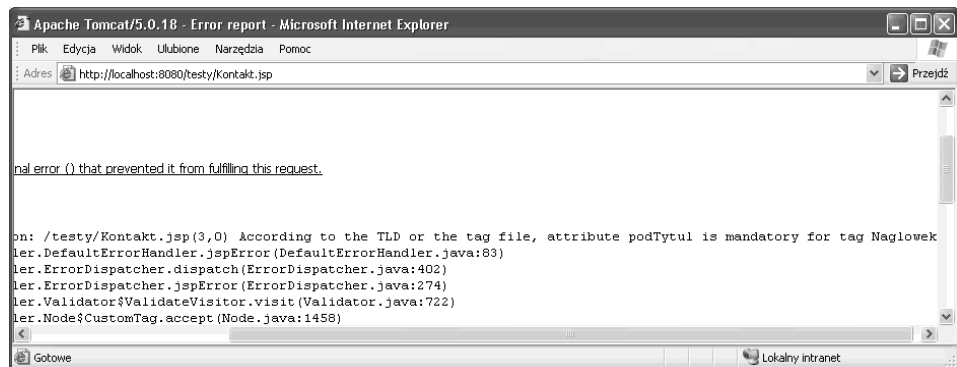
```
<%@ taglib prefix="mojeZnaczniki" tagdir="/WEB-INF/tags" %>
<html><body>
<mojeZnaczniki:Naglowek podTytuł="Pobieramy dane przy użyciu SOAP." />

Skontaktuj się z nami: ${initParam.adresEmail}
</html></body>
```

## A co się stanie, jeśli używając znacznika w dokumencie, NIE podamy jego atrybutu?

`<mojeZnaczniki:Naglowek />`

Tego nie można zrobić... Nie można pominąć atrybutu `podTytuł`, ponieważ dyrektywa `attribute` w pliku znacznika mówi: `required="true"`.



## Kiedy zawartość atrybutu jest naprawdę duża...

Wyobraźmy sobie, że mamy atrybut, którego wartość może mieć długość, na przykład, całego akapitu tekstu. Umieszczanie go w znaczniku otwierającym elementu mogłoby być fatalne dla przejrzystości kodu strony. Można zatem wybrać inne rozwiązanie, polegające na umieszczeniu treści jako zawartości znacznika, a następnie zastosowaniu go jako rodzaju atrybutu.

Tym razem umieścimy atrybut `podTytuł` *poza* znacznikiem, a jego zawartość umieścimy w *ciele* znacznika `<mojeZnaczniki:Naglowek>`.

### Wewnątrz pliku znacznika (Naglowek.tag)

```


<jsp:doBody/>
```

Nie potrzebujemy już dyrektywy `attribute`!

Ten znacznik informuje, że w tym miejscu należy umieścić zawartość znacznika użytego do wywołania tego pliku.

### Wewnątrz pliku JSP używającego tego znacznika

```
<%@ taglib prefix="mojeZnaczniki" tagdir="/WEB-INF/tags" %>
<html><body>
```

#### `<mojeZnaczniki:Naglowek>`

Pobieramy dane przy użyciu SOAP. OK, może nie jest to to samo co Jini, `<br>`, ale pomożemy Ci przejść przez naukę tej metody w jak najmniej stresujący sposób.

#### `</mojeZnaczniki:Naglowek>`

```


```

```
Skontaktuj się z nami: ${initParam.adresEmail}
```

```
</body></html>
```

Obecnie odpowiedni tekst definiujemy w ciele elementu, nie jako wartość atrybutu umieszczanego w znaczniku otwierającym.

*Jednak i w tym przypadku pojawia się ten sam problem co wcześniej — skoro nie używamy TLD, to gdzie deklarujemy typ zawartości znacznika?*



# Deklarowanie zawartości znacznika dla pliku znacznika

Jedynym sposobem zadeklarowania typu zawartości znacznika używanego do wywołania pliku znacznika jest użycie kolejnej nowej dyrektywy — **dyrektywy `tag`**. Dyrektywa `tag` jest w plikach znaczników tym, czym dla stron dyrektywa `page`. Obie te dyrektywy posiadają niemal identyczne atrybuty, z jednym wyjątkiem — dyrektywa `tag` posiada jeden dodatkowy atrybut: **`body-content`**.

W przypadku znaczników niestandardowych niezbędne jest podanie elementu `<body-content>` wewnątrz elementu `<tag>` w deskrytorze TLD! Jednak w pliku znacznika nie trzeba deklarować atrybutu `body-content`, jeśli w znaczniku można umieścić treść domyślną — czyli pozbawioną kodu skryptowego (odpowiada jej wartość *scriptless* atrybutu `body-content`). Wartość **`scriptless`** oznacza, że w zawartości nie mogą wystąpić żadne elementy skryptowe. Pamiętaj, iż do takich elementów zaliczane są: *skrypty* (`<% ... %>`), *wyrażenia skryptletów* (`<%= ... %>`) oraz *deklaracje* (`<%! ... %>`).

W zasadzie *wewnątrz plików znaczników nigdy nie można umieszczać kodu skryptowego*, a zatem pod tym względem nie mamy żadnej możliwości wyboru. Można jednak zadeklarować typ zawartości (za pomocą dyrektywy `tag` z atrybutem `body-content`), jeśli chcemy, by mogła ona być *pusta* (*empty*) lub *zależna od znacznika* (*tagdependent*).

## Wewnątrz pliku znacznika, w którym użyto dyrektywy `tag` (Nagłówek `tag`)

```
<%@ attribute name="kolorCzcionki" required="true" %>
```

```
<%@ tag body-content="tagdependent" %>
```

Ta wartość oznacza, że zawartość znacznika będzie traktowana jak normalny tekst; to z kolei sugeruje, że język wyrażień (EL), znaczniki oraz kod skryptowy NIE będą w żaden sposób przetwarzane. Innymi dopuszczalnymi wartościami tego atrybutu są: „empty” oraz (domyślnie) „scriptless”.

```


```

```
<jsp:doBody/>

```

W ciele pliku znacznika **NIE MOŻNA** umieszczać kodu skryptowego!

Domyślnie zawartość pliku znacznika jest pozbawiona kodu skryptowego, zatem nie trzeba deklarować jej typu, chyba że chcemy użyć jednej z dwóch POZOSTAŁYCH opcji: „empty” (zawartość znacznika jest pusta) lub „tagdependent” (zawartość będzie traktowana jako zwyczajny tekst).

## Wewnątrz strony JSP używającej tego znacznika

```
<%@ taglib prefix="mojeZnaczniki" tagdir="/WEB-INF/tags" %>
```

```
<html><body>
```

```
<mojeZnaczniki:Naglowek kolorCzcionki="#660099">
```

Pobieramy dane przy użyciu SOAP. OK, może nie jest to to samo co Jini, `<br>`, ale pomożemy Ci przejść przez naukę tej metody w jak najmniej stresujący sposób.

```
</mojeZnaczniki:Naglowek>
```

```


```

```
Skontaktuj się z nami: ${initParam.adresEmail}
```

```
</body></html>
```

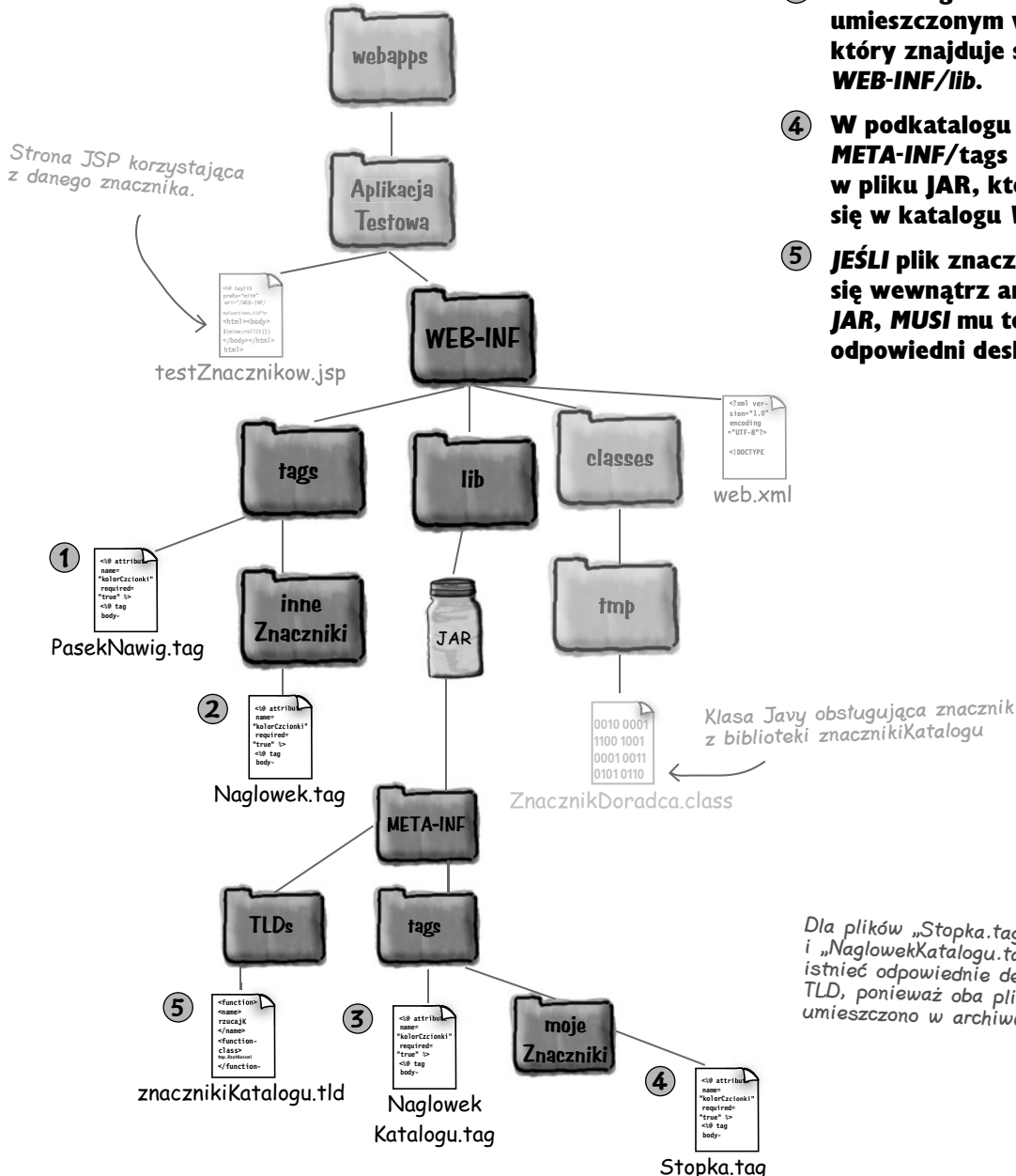
Atrybut „kolorCzcionki” został zadeklarowany w pliku znacznika przy użyciu dyrektywy `attribute`.

Typ tej zawartości został zadeklarowany w pliku znacznika przy użyciu dyrektywy `tag` z atrybutem `body-content`.

## Gdzie kontener szuka plików znaczników?

Kontener szuka plików znaczników w czterech różnych miejscach. Jeśli plik znacznika jest wdrażany wewnątrz archiwum JAR, odpowiedni deskryptor TLD jest wymagany; jeśli natomiast jest umieszczony bezpośrednio w aplikacji (w katalogu *WEB-INF/tags* lub w jego podkatalogu), deskryptor TLD nie jest konieczny.

- ❶ Bezpośrednio w katalogu **WEB-INF/tags**.
- ❷ W jednym z podkatalogów katalogu **WEB-INF/tags**.
- ❸ W katalogu **META-INF/tags** umieszczonym w pliku JAR, który znajduje się w katalogu **WEB-INF/lib**.
- ❹ W podkatalogu katalogu **META-INF/tags** umieszczonym w pliku JAR, który znajduje się w katalogu **WEB-INF/lib**.
- ❺ JEŚLI plik znacznika znajduje się wewnątrz archiwum JAR, **MUSI** mu towarzyszyć odpowiedni deskryptor TLD.



### Nie ma niemądrych pytań

**P.** Czy plik znacznika ma dostęp do domyślnych obiektów żądania i odpowiedzi?

**U.** Tak! Pamiętaj, iż choć jest to plik z rozszerzeniem `.tag`, to ostatecznie będzie on stanowić część strony JSP. Dlatego też można używać domyślnych obiektów żądania i odpowiedzi (jeśli zastosujemy *kod skryptowy*... dostępne są także domyślne obiekty języka wyrażen EL). Dysponujemy także dostępem do obiektu `JspContext`.

W plikach znaczników nie ma jednak dostępu do obiektu `ServletContext` — jego rolę spełnia obiekt `JspContext`.

**P.** Wydawało mi się, że na poprzedniej stronie napisaliście, iż w pliku znacznika nie można umieszczać żadnego kodu skryptowego!

**U.** Nie. Napisaliśmy coś innego — że *nie można* stosować kodu skryptowego *w znaczniku* używanym do wywołania pliku znacznika, natomiast *wewnątrz pliku* znacznika kod skryptowy jest dopuszczalny.

**P.** Czy w tym samym katalogu można umieszczać pliki znaczników i deskryptory TLD znaczników niestandardowych?

**U.** Tak. W rzeczywistości, jeśli stworzymy deskryptor TLD odwołujący się do pliku znacznika, to kontener uzna, iż plik znacznika oraz znaczniki niestandardowe zdefiniowane w *tych* samym deskryptorze *należą do tej samej biblioteki*.

**P.** Chwileczkę! Myślałem, że napisaliście, iż pliki znaczników nie mają deskryptorów TLD. Przecież właśnie dlatego dodano dyrektywę `attribute`. Bo przecież w deskryptorze nie można zadeklarować atrybutu. Jak to jest?

**U.** To „podchwytliwe” pytanie. Jeśli pliki znaczników są umieszczone w archiwum JAR, to konieczne jest stworzenie odpowiednich deskryptorów TLD określających ich położenie. Niemniej jednak deskryptor nie opisuje atrybutów znacznika, jego zawartości itp. Informacje na temat pliku znacznika podawane w deskryptorze opisują jedynie jego położenie.

Oto przykład deskryptora TLD dla pliku znacznika:

```
<taglib ...>
 <tlib-version>1.0</tlib-version>
 <uri>mojaBibliotekaZnacznikow</uri>
 <tag-file>
 <name>Naglowek</name>
 <path>/META-INF/tags/Naglowek.tag</path>
 </tag-file>
</taglib>
```

Należy zauważyć, że zawartość elementu `<tag-file>` znacznie się różni od zawartości elementu `<tag>`.

**P.** Ale dlaczego deskryptor pliku znacznika jest inny od deskryptora znacznika niestandardowego? Czy nie byłoby znacznie prościej, gdyby zarówno znaczniki niestandardowe, jak i pliki znaczników były deklarowane w deskryptorach TLD w identyczny sposób? Ale NIE... zamiast tego musieli wymyślić całkowicie nowe rozwiązanie, wymagające zastosowania nowych dyrektyw do definiowania atrybutów i zawartości znacznika. A zatem, dlaczego metody zastosowania znaczników i plików znaczników są odmienne?

**U.** Z jednej strony to fakt, iż byłoby znacznie prościej gdyby znaczniki i pliki znaczników były deklarowane w taki sam sposób, w deskryptorze TLD. Tylko dla kogo takie rozwiązanie byłoby łatwiejsze? Dla twórców znaczników niestandardowych? To na pewno. Jednak pliki znaczników są tworzone i dodawane do specyfikacji nie z myślą o ich twórcach, lecz o projektantach stron.

Pliki znaczników pozwalają osobom, które nie znają Javy, na tworzenie znaczników niestandardowych bez konieczności tworzenia klas implementujących ich możliwości funkcjonalne. Poza tym fakt, iż nie trzeba pisać TLD dla pliku znacznika, jedynie upraszcza życie ich twórcom. (Pamiętaj, że pliki znaczników wymagają TLD, jeśli są umieszczane w archiwach JAR, jednak osoby, które nie znają Javy, i tak nie będą ich używać w taki sposób).

Wniosek: Znaczniki niestandardowe *muszą* posiadać odpowiednie deskryptory TLD, natomiast w przypadku plików znaczników zawartość elementu oraz jego atrybuty można zdefiniować bezpośrednio w pliku znacznika; deskryptory są wymagane *wyłącznie* w przypadku, gdy plik znacznika jest umieszczany wewnątrz archiwum JAR.



## Zaostrz ołówek

**Zapamiętywanie sposobu stosowania plików znaczników**

Zanim zajmiemy się następnym zagadnieniem, upewnij się, że potrafisz samodzielnie napisać i zastosować plik znacznika (odpowiedzi na pytania można znaleźć na końcu rozdziału).

- 1 W poniższej ramce zapisz to, co należałoby umieścić w pliku znacznika, aby zadeklarować, że wymaga on podania jednego atrybutu o nazwie „tytuł”, którego wartością może być wyrażenie EL.

&lt;%@

%&gt;

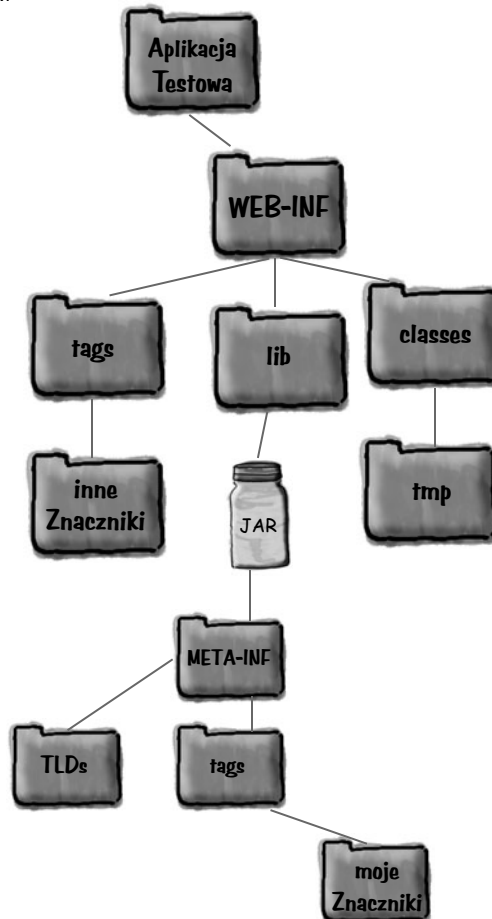
W poniższej ramce zapisz deklarację informującą, że znacznik nie może zawierać ciała.

- 2

&lt;%@

%&gt;

- 3 Dorysuj dokument pliku znacznika w każdym z miejsc, w jakich kontener będzie poszukiwał plików znaczników.



# Kiedy potrzeby przekraczają możliwości plików znaczników... Czasami potrzebna będzie Java

Pliki znaczników doskonale spełniają swoje zadania, jeśli należy *dołączyć* zawartość — kiedy wszystkie operacje związane z obsługą znacznika można wykonać w *innej* stronie JSP (w której rozszerzenie zmieniono na *.tag* i dodano odpowiednie dyrektywy). Jednak niejednokrotnie te możliwości nie wystarczą. Czasami konieczne będzie zastosowanie kodu pisanego w Javie, a jednocześnie nie będziemy chcieli wykonywać niezbędnych operacji z poziomu skryptletu, gdyż właśnie tego chcemy uniknąć poprzez zastosowanie znaczników.

Jeśli pojawi się konieczność zastosowania Javy, będziemy musieli stworzyć *klasę obsługi* znacznika niestandardowego. Zgodnie z tym, co sugeruje nazwa, jest to zwyczajna klasa Javy, która zapewnia możliwości funkcjonalne znacznika. Klasy te w pewnym stopniu przypominają funkcje języka EL, przy czym mają znacznie większe możliwości i są bardziej elastyczne. Funkcje EL są jedynie zwyczajnymi, statycznymi metodami, natomiast klasy obsługi znaczników niestandardowych mają pełny dostęp do atrybutów znacznika, jego zawartości, a nawet do kontekstu strony, dzięki czemu mogą korzystać z atrybutów zasięgu strony oraz obiektów żądania i odpowiedzi.

Istnieją dwa rodzaje klas obsługi znaczników niestandardowych: *klasyczne* i *proste*. Znaczniki klasyczne były dostępne w poprzednich wersjach technologii JSP, jednak w wersji JSP 2.0 dodano nowy, *znacznie prostszy* model znaczników. Trudno obecnie wskazać choć jeden powód, który usprawiedliwiłoby zastosowanie klasycznych klas obsługi znaczników, ponieważ nowy, prosty model (zwłaszcza jeśli zastosujemy go wraz z JSTL oraz plikami znaczników) daje niemal wszystkie możliwości, jakich moglibyśmy potrzebować. Niemniej jednak istnieją dwie przyczyny, które sprawiają, że nie można całkowicie odrzucić modelu klasycznego oraz że pytania na jego temat wciąż pojawiają się na egzaminie. Oto one:

1. Podobnie jak kod skryptowy klasyczne klasy obsługi znaczników niestandardowych *istnieją* i, być może, będziesz musiał je analizować i utrzymywać, nawet jeśli samemu nie będziesz ich tworzyć.
2. Istnieją pewne rzadkie sytuacje, w których klasyczne klasy obsługi znaczników są najlepszym rozwiązaniem. Nie można jednak tego wyjaśnić w prosty i krótki sposób, a zatem zdecydowanie najważniejszym powodem do nauki znaczników klasycznych jest punkt nr 1.

W pierwszej kolejności, niejako „na rozgrzewkę” omówimy *prosty model znaczników niestandardowych*.

Pliki znaczników implementują możliwości znacznika, wykorzystując do tego celu inną stronę (JSP).

Klasy obsługi znaczników implementują możliwości znacznika przy użyciu specjalnej klasy języka Java.

Istnieją dwa rodzaje klas obsługi znaczników niestandardowych: proste i klasyczne.

# Tworzenie prostej klasy obsługi znacznika

W najprostszym przypadku znaczników prostych proces tworzenia klasy ich obsługi jest... *prosty*.

## 1 Napisz klasę dziedziczącą po klasie SimpleTagSupport.

```
package tmp;
import javax.servlet.jsp.tagext.SimpleTagSupport;
// trzeba także zaimportować inne klasy
public class TestProstegoZnacznika1 extends SimpleTagSupport {
 // kod obsługujący znacznik
}
```

## 2 Zaimplementuj metodę doTag()

```
public void doTag() throws JspException, IOException {
 getJspContext().getOut().print("Oto jeden z gorszych przykładów użycia znacznika
niestandardowego");
}
```

Metoda `doTag()` deklaruje wyjątek `IOException`, więc nie trzeba umieszczać wywołania metody `print` wewnątrz bloku `try-catch`.

## 3 Napisz deskryptor TLD dla znacznika

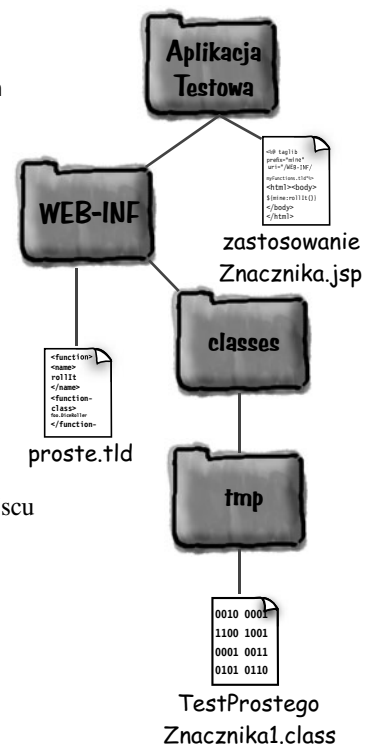
```
<taglib ...>
 <tlib-version>1.2</tlib-version>
 <uri>prosteZnaczniki</uri>
 <tag>
 <description>Najgorsze zastosowanie znaczników niestandardowych
 </description>
 <name>znacznik1</name>
 <tag-class>tmp.TestProstegoZnacznika1</tag-class>
 <body-content>empty</body-content>
 </tag>
</taglib>
```

## 4 Umieść klasę obsługi znacznika oraz TLD na serwerze

Deskryptor TLD umieść w katalogu *WEB-INF*, a klasę obsługi znacznika niestandardowego w katalogu *WEB-INF/classes* (oczywiście stosując odpowiednią strukturę podkatalogów odpowiadającą nazwie pakietu, do którego należy klasa obsługi). Innymi słowy, klasa obsługi znacznika jest umieszczana w tym samym miejscu co wszelkie inne klasy wchodzące w skład aplikacji.

## 5 Napisz stronę JSP wykorzystującą znacznik

```
<%@ taglib prefix="mojeZnaczniki" uri="prosteZnaczniki" %>
<html><body>
<mojeZnaczniki:znacznik1/>
</body></html>
```



## Znacznik prosty z zawartością

Jeśli znacznik musi mieć jakąś zawartość (ciało), należy to uwzględnić w elemencie `<body-content>`; oprócz tego należy umieścić specjalne wywołanie w metodzie `doTag()`.

### Kod JSP używający znacznika

```
<%@ taglib prefix="mojeZnaczniki" uri="prosteZnaczniki" %>
```

```
<html><body>
```

Znacznik prosty nr. 2:

```
<mojeZnaczniki:znacznik2>
```

To jest zawartość znacznika

Tym razem w wywołaniu znacznika  
umieszczamy zawartość...

```
</mojeZnaczniki:znacznik2>
```

```
</body></html>
```

### Klasa obsługująca znacznik

```
package tmp;
```

```
import javax.servlet.jsp.JspException;
```

```
import javax.servlet.jsp.tagext.SimpleTagSupport;
```

```
import javax.io.IOException;
```

```
public class TestProstegoZnacznika2 extends SimpleTagSupport {
```

```
 public void doTag() throws JspException, IOException {
```

```
 getJspBody().invoke(null);
```

```
 }
```

```
}
```

To wywołanie oznacza: „Przetwórz  
zawartość znacznika i przekaz ją  
do odpowiedzi”. Argument `null`  
sugeruje, że wyniki mają trafić do  
odpowiedzi, a nie do jakiegoś INNEGO,  
przekazanego obiektu `Writer`.

### Deskryptor TLD znacznika

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
```

```
<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
```

```
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
 xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/web-jsptaglibrary_2_0.xsd"
```

```
 version="2.0">
```

```
<tlib-version>1.2</tlib-version>
```

```
<uri>prosteZnaczniki</uri>
```

```
<tag>
```

```
 <description>Nieco lepszy sposób użycia znaczników niestandardowych</description>
```

```
 <name>znacznik2</name>
```

```
 <tag-class>tmp.TestProstegoZnacznika2</tag-class>
```

```
 <body-content>scriptless</body-content>
```

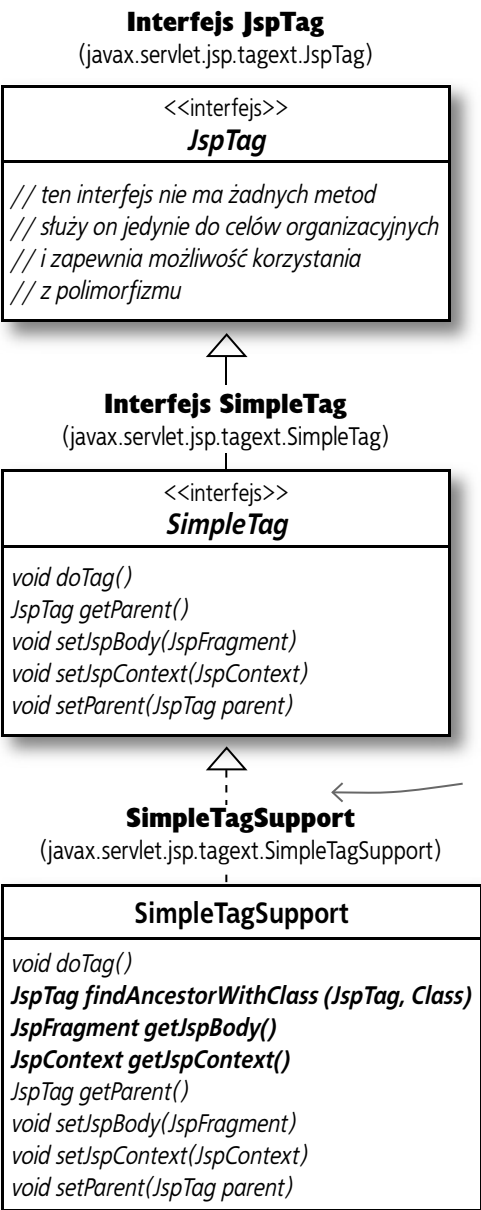
```
</tag>
```

```
</taglib>
```

Ta deklaracja oznacza, że znacznik  
może mieć zawartość, która jednak  
nie może zawierać kodu skryptowego  
żadnego typu (ani skryptletów, ani  
wyrażeń skryptowych, ani deklaracji).

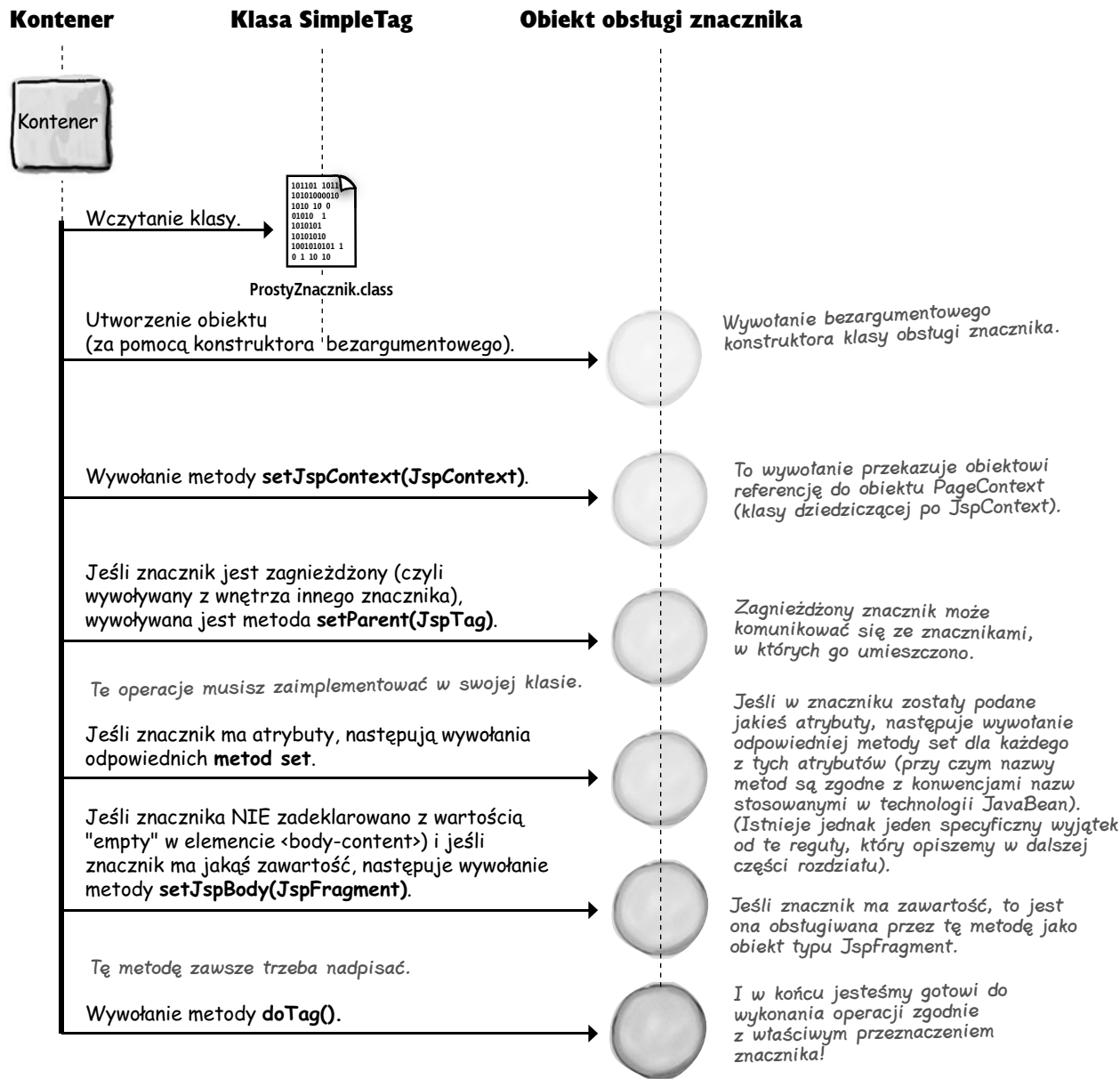
# Interfejs programowy znaczników prostych

Klasa obsługująca znacznik prosty musi implementować interfejs SimpleTag. Najprostszym sposobem spełnienia tego wymogu jest stworzenie klasy dziedziczącej po klasie SimpleTagSupport i nadpisanie w niej odpowiedniej metody — doTag(). Oczywiście nie trzeba korzystać z klasy SimpleTagSupport, choć przypuszczamy, że właśnie z niej korzysta 99,99999% programistów tworzących proste znaczniki niestandardowe.



# Cykl życia obiektu obsługi prostego znacznika niestandardowego

Kiedy kod JSP wywołuje znacznik niestandardowy, tworzony jest nowy obiekt klasy obsługi tego znacznika. Następnie wywoływana jest jedna lub dwie metody tego obiektu, a w końcu, po zakończeniu metody `doTag()`, obiekt przestaje być potrzebny i jest niszczone. (Innymi słowy, obiekty te *nie* są używane wielokrotnie).



## BĄDŹ kontenerem



Przyjrzyj się obu przedstawionym poniżej parom składającym się z deskryptora TLD i kodu JSP. Załóż, że klasa obsługi znacznika nie musi wykonywać żadnych operacji, a następnie odpowiedz na następujące pytania dla każdej z przedstawionych par: Jakiego będą efekty ich zastosowania? Jeśli dana para okaże się prawidłowa, co zostanie wyświetlone? Które metody zdefiniowane w klasie obsługi znacznika niestandardowego zostaną wywołane?

① `<tag>`  
`<description></description>`  
`<name>prostyZnacznik</name>`  
`<tag-class>tmp.TestProstegoZnacznika</tag-class>`  
`<body-content>empty</body-content>`  
`</tag>`

Znacznik prosty:  
`<mojeZnaczniki:prostyZnacznik>`  
 To jest zawartość znacznika  
`</mojeZnaczniki:prostyZnacznik>`

### Co zobaczymy w przeglądarce?

**Jeśli powyższy deskryptor TLD i kod JSP działają poprawnie, które z metod cyklu życia znacznika należących do interfejsu SimpleTag zostaną wywołane?**

☐ `void doTag()`   ☐ `JspTag getParent()`   ☐ `void setJspBody()`   ☐ `void setJspContext()`   ☐ `void setParent()`

② `<tag>`  
`<description></description>`  
`<name>prostyZnacznik</name>`  
`<tag-class>tmp.TestProstegoZnacznika</tag-class>`  
`<body-content>scriptless</body-content>`  
`</tag>`

Znacznik prosty:  
`<mojeZnaczniki:prostyZnacznik>`  
`${2*3}`  
`</mojeZnaczniki:prostyZnacznik>`

### Co zobaczymy w przeglądarce?

**Jeśli powyższy deskryptor TLD i kod JSP działają poprawnie, które z metod cyklu życia znacznika należących do interfejsu SimpleTag zostaną wywołane?**

☐ `void doTag()`   ☐ `JspTag getParent()`   ☐ `void setJspBody()`   ☐ `void setJspContext()`   ☐ `void setParent()`

# BĄDŹ kontenerem

Odpowiedzi



❶ `<tag>`  
    `<description></description>`  
    `<name>prostyZnacznik</name>`  
    `<tag-class>tmp.TestProstegoZnacznika</tag-class>`  
    `<body-content>empty</body-content>`  
`</tag>`

Znacznik prosty:  
`<mojeZnaczniki:prostyZnacznik>`  
    To jest zawartość znacznika  
`</mojeZnaczniki:prostyZnacznik>`

## Co zobaczymy w przeglądarce?

Ten przykład nie działa, ponieważ `prostyZnacznik` zadeklarowano jako znacznik pusty.  
`org.apache.jasper.JasperException: /prostyZnacznik1.jsp(1,76)`  
According to TLD, tag `mojeZnaczniki:prostyZnacznik` must be empty, but is not

Żadna z metod nie zostanie wywołana, gdyż przykład nie działa.

**Jeśli powyższy deskryptor TLD i kod JSP działają poprawnie, które z metod cyklu życia znacznika należących do interfejsu `SimpleTag` zostaną wywołane?**

☐ `void doTag()`   ☐ `JspTag getParent()`   ☐ `void setJspBody()`   ☐ `void setJspContext()`   ☐ `void setParent()`

❷ `<tag>`  
    `<description></description>`  
    `<name>prostyZnacznik</name>`  
    `<tag-class>tmp.TestProstegoZnacznika</tag-class>`  
    `<body-content>scriptless</body-content>`  
`</tag>`

Znacznik prosty:  
`<mojeZnaczniki:prostyZnacznik>`  
    `${2*3}`  
`</mojeZnaczniki:prostyZnacznik>`

## Co zobaczymy w przeglądarce?

Znacznik prosty: 6

Metoda `setParent()` jest wywoływana wyłącznie w przypadku, gdy znacznik zostanie wywołany z WNETRZA innego znacznika. Ponieważ w naszym przykładzie znacznik nie jest zagnieżdżony, metoda `setParent()` NIE zostanie wywołana.

**Jeśli powyższy deskryptor TLD i kod JSP działają poprawnie, które z metod cyklu życia znacznika należących do interfejsu `SimpleTag` zostaną wywołane?**

☒ `void doTag()`   ☐ `JspTag getParent()`   ☒ `void setJspBody()`   ☒ `void setJspContext()`   ☐ `void setParent()` ←

## A co, jeśli w zawartości znacznika pojawi się wyrażenie?

Wyobraź sobie, że dysponujesz znacznikiem z ciałem, w którym użyto wyrażenia EL odwołującego się do jednego z atrybutów. A teraz wyobraź sobie, że w momencie wywołania Twojego znacznika wspomniany atrybut nie istnieje! Innymi słowy, *zawartość* znacznika zależy od jego *klasy obsługi*, która musi określić wartość atrybutu. Zamieszczony poniżej przykład nie jest może zbyt przydatny, ale dobrze pokazuje, jak działa cały interesujący nas mechanizm, i przygotowuje nas do większego i bardziej użytecznego przykładu.

### Wywołanie znacznika w kodzie JSP

```
<mojeZnaczniki:znacznikProsty3>
 Oto komunikat: ${komunikat}
</mojeZnaczniki:znacznikProsty3>
```

W chwili wywoływania znacznika atrybut „komunikat” **NIE** istnieje! Jeśli wyrażenie odwołujące się do tego atrybutu zostanie umieszczone poza znacznikiem, zwróci wartość null.

### Metoda doTag() w klasie obsługi znacznika

```
public void doTag() throws JspException, IOException {
 getJspContext().setAttribute("komunikat", "Używaj kremu z filtrem!");
 getJspBody().invoke(null);
}
```

Metoda określa wartość atrybutu a NASTĘPNIE przetwarza ciało atrybutu

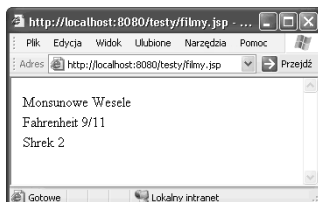
Zaostrz ołówek



Wyobraź sobie, że dysponujesz następującym znacznikiem:

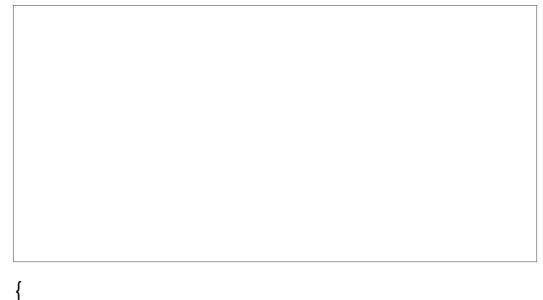
```
<table>
<mojeZnaczniki:znacznikProsty4>
 <tr><td>${film}</td></tr>
</mojeZnaczniki:znacznikProsty4>
</table>
```

Załóż, że klasa obsługi znacznika ma dostęp do tablicy łańcuchów reprezentujących tytuły filmów, a każdy tytuł należy wyświetlić w osobnym wierszu tabeli. W przeglądarce ma to wyglądać mniej więcej tak jak na poniższym rysunku:



Napisz kod metody doTag(), która będzie generować stosowny kod HTML.

```
public void doTag() throws JspException,
IOException {
```



# Znacznik z dynamicznie generowaną zawartością wierszy

## — iteracyjne przetwarzanie ciała znacznika

W przykładzie przedstawionym na tej stronie wyrażenie umieszczone wewnątrz znacznika reprezentuje pojedynczą wartość z kolekcji, a naszym zadaniem jest sprawienie, by znacznik wygenerował po jednym wierszu dla każdego elementu tej kolekcji. Realizacja tego zadania jest całkiem prosta — metoda `doTag()` musi po prostu zawierać pętlę, której każda iteracja będzie polegać na przetworzeniu ciała znacznika.

### Wywołanie znacznika w kodzie JSP

```
<table>
 <mojeZnaczniki:znacznikProsty4>
 <tr><td>${film}</td></tr>
 </mojeZnaczniki:znacznikProsty4>
</table>
```

W chwili wywoływania znacznika atrybut `film` nie istnieje. Zostanie on utworzony podczas obsługi znacznika, którego zawartość będzie cyklicznie przetwarzana w pętli.

### Metoda `doTag()` klasy obsługi znacznika

```
String [] filmy = {"Monsunowe Wesele", "Fahrenheit 9/11", "Shrek 2"};

public void doTag() throws JspException, IOException {
 for (int i = 0; i < filmy.length; i++) {
 getJspContext().setAttribute("film", filmy[i]);
 getJspBody().invoke(null);
 }
}
```

W atrybucie zapisujemy kolejną wartość z tablicy.

Ponownie przetwarzamy zawartość znacznika.

### JSP

```
<mojeZnaczniki:znacznikProsty4>
 <tr><td>
 ${film}
 </td></tr>
</mojeZnaczniki:znacznikProsty4>
```

### Kod klasy obsługi znacznika

```
for (int i = 0; i < filmy.length; i++) {
 getJspContext().setAttribute("film", filmy[i]);
 getJspBody().invoke(null);
}
```

W każdej iteracji pętli określamy nową wartość atrybutu `"film"` i wywołujemy metodę `getJspBody().invoke()`.

## Znacznik prosty z atrybutem

Jeśli znacznik musi mieć atrybut, należy ten atrybut zadeklarować w TLD, a w klasie obsługi znacznika musimy zdefiniować metodę ustawiającą jego wartość (zgodnie z konwencją nazewnictwa stosowaną w technologii JavaBeans). Jeśli w wywołaniu znacznika pojawiają się jakieś atrybuty, dla każdego z nich kontener wywoła odpowiednią metodę set.

### Wywołanie znacznika w kodzie JSP

```
<table>
 <mojeZnaczniki:znacznikProsty5 listaFilmow="${kolekcjaFilmow}">
 <tr>
 <td>${film.nazwa}</td>
 <td>${film.rodzaj}</td>
 </tr>
 </mojeZnaczniki:znacznikProsty5>
</table>
```

To jest zwyczajny atrybut, taki sam jak wszelkie inne atrybuty znaczników. Nie ma tutaj znaczenia fakt obsługi tego znacznika przez klasę TestZnacznikaProstego5.

Pominęliśmy instrukcje importu...

### Klasa obsługi znacznika

```
public class TestZnacznikaProstego5 extends SimpleTagSupport {
 private List listaFilmow;
 public void setListaFilmow(List listaFilmow) {
 this.listaFilmow = listaFilmow;
 }
 public void doTag() throws JspException, IOException {
 Iterator i = listaFilmow.iterator();
 while (i.hasNext()) {
 Film film = (Film) i.next();
 getJspContext().setAttribute("film", film);
 getJspBody().invoke(null);
 }
 }
}
```

Zadeklaruj zmienną, która będzie przechowywać wartość atrybutu.

Napisz metodę ustawiającą wartość zmiennej reprezentującej atrybut. Nazwa tej metody MUSI odpowiadać nazwie atrybutu podanej w deskrytorze TLD (oczywiście pomijając początkowe litery „set” i z uwzględnieniem zmiany pierwszej litery na wielką).

### Deskryptor TLD znacznika

```
<tag>
 <description>Znacznik pobiera atrybut i iteracyjnie przetwarza ciało</description>
 <name>znacznikProsty5</name>
 <tag-class>tmp.TestZnacznikaProstego5</tag-class>
 <body-content>scriptless</body-content>
 <attribute>
 <name>listaFilmow</name>
 <required>true</required>
 <rtexprvalue>true</rtexprvalue>
 </attribute>
</tag>
```

W TLD należy standardowo zadeklarować atrybut przy użyciu elementu <attribute> umieszczonego w elemencie <tag>, a więc podobnie jak w przypadku znaczników niestandardowych (oczywiście z wyjątkiem plików znaczników).

# Czym dokładnie JEST obiekt JspFragment?

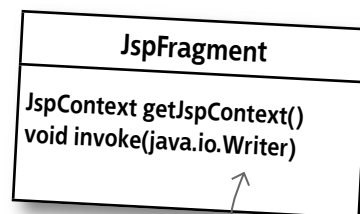
JspFragment to obiekt reprezentujący kod JSP. Obiekty te istnieją tylko w jednym celu — aby je wywoływać. Innymi słowy, obiekt JspFragment służy do *wykonania* i wygenerowania *danych wynikowych*. Zawartość znacznika wywołującego klasę obsługi znacznika prostego jest umieszczana w obiekcie JspFragment, a następnie, w wyniku wywołania metody `setJspBody()`, przekazywana do klasy obsługi tego znacznika.

Konieczne musisz zapamiętać, że obiekty JspFragment **NIE** mogą zawierać żadnych elementów skryptowych! Mogą zawierać tekst, akcje (zarówno standardowe, jak i niestandardowe), wyrażenia EL; jednak nie mogą zawierać skryptletów, deklaracji ani wyrażen skryptowych.

Warto zauważyć, że JspFragment jest obiektem i jako taki może być przekazywany do innych obiektów pomocniczych, co bywa bardzo przydatne. Te *inne* obiekty mogą z kolei uzyskiwać potrzebne informacje za pośrednictwem obiektu JspFragment i jego metod (takich jak `getJspContext()`). Oczywiście po pobraniu kontekstu można także pobrać wartości atrybutów. Jak zatem widać, metoda `getJspContext()` pozwala na przekazywanie informacji innym obiektom biorącym udział w przetwarzaniu zawartości znacznika.

Z drugiej strony, w większości przypadków obiekty JspFragment wykorzystuje się tylko do przekazywania zawartości znacznika do obiektu odpowiedzi. Nie można jednak wykluczyć, że w pewnych okolicznościach stanimy przed koniecznością uzyskania dostępu do *zawartości* ciała znacznika. Należy zauważyć, że obiekt JspFragment nie dysponuje żadnymi metodami typu `getContent()` lub `getBody()`. Można co prawda *zapisać* zawartość znacznika w jakimś obiekcie, jednak takie rozwiązanie nie zapewni nam bezpośredniego *dostępu* do ciała. Jeśli mimo wszystko chcemy takim dostępem dysponować, możemy na wejściu metody `invoke()` przekazać (w formie argumentu) obiekt typu `java.io.Writer`, po czym użyć jego metod do przetworzenia zawartości ciała znacznika.

W ten sposób wyczerpaliśmy wiedzę szczegółową na temat obiektu JspFragment, której będziesz potrzebował na egzaminie i w praktycznych zastosowaniach — w tej sytuacji nie będziemy temu obiektowi poświęcać więcej czasu i uwagi.



Metoda `invoke()` otrzymuje na wejściu obiekt `Writer`. Można też przekazać wartość `null`, aby zawartość znacznika została przekazana bezpośrednio do odpowiedzi. Jeśli chcemy uzyskać bezpośredni dostęp do zawartości znacznika, należy przekazać w wywołaniu metody `invoke()` obiekt typu `Writer`.

**W wywołaniu metody `invoke()` należy przekazać argument typu `java.io.Writer`. Jeśli jednak ciało znacznika ma być tylko zapisane w odpowiedzi, na wejściu tej metody należy przekazać wartość null.**

**W większości przypadków będziemy stosowali właśnie takie rozwiązanie. Jeśli jednak będziemy chcieli uzyskać dostęp do zawartości znacznika, wystarczy w wywołaniu metody `invoke()` przekazać obiekt typu `Writer`, a następnie przetworzyć zawartość znacznika, posługując się metodami tego obiektu.**

## Wyjątek `SkipPageException` przerywa przetwarzanie strony...

Wyobraź sobie, że tworzymy stronę wywołującą znacznik niestandardowy, którego działanie zależy od określonych atrybutów żądania (znacznik pobiera je, używając obiektu `JspContext` dostępnego w klasie obsługi znacznika).

A teraz wyobraź sobie, że znacznik nie jest w stanie odnaleźć niezbędnych atrybutów, a jednocześnie „wie”, że w razie braku możliwości jego poprawnego wygenerowania cały proces generowania strony zakończy się niepowodzeniem. Co należy zrobić w takiej sytuacji? Można by zgłosić wyjątek `JspException`, a to doprowadziłoby do przerwania przetwarzania strony... ale co zrobić, jeśli niepowodzenie procesu generowania znacznika uniemożliwi wyłącznie generowanie *dalszej części* strony? Innymi słowy, co zrobić, jeśli chcemy, by poprzednia część strony — wygenerowana *przed* nieudanym przetworzeniem znacznika — została przekazana w odpowiedzi, a chcemy pominąć jedynie to, co należałoby przetworzyć *po* zgłoszeniu wyjątku?

Nie ma żadnego problemu. Właśnie w tym celu stworzono wyjątek `SkipPageException`.

### Metoda `doTag()` klasy obsługi znacznika

```
public void doTag() throws JspException, IOException {
 getJspContext().getOut().print("Komunikat wygenerowany w metodzie doTag().
");
 getJspContext().getOut().print("Zgłaszamy wyjątek SkipPageException");
 if (cosNieZadzialalo) {
 throw new SkipPageException();
 }
}
```

W tym przypadku podejmujemy decyzję o przerwaniu dalszego przetwarzania znacznika i strony. W odpowiedzi pojawi się zatem wyłącznie część strony i znacznika wygenerowana PRZED zgłoszeniem wyjątku.

### Kod JSP wywołujący znacznik

```
<%@ taglib prefix="mojeZnaczniki" uri="znacznikiProste" %>

<html><body>
 Zaraz wywołamy znacznik zgłaszający wyjątek SkipPageException

 <mojeZnaczniki:znacznikProsty6/>

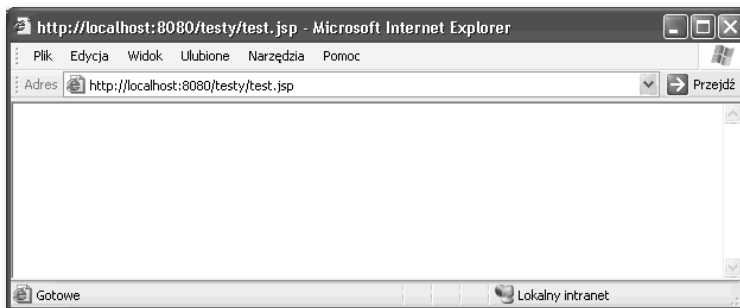
Znowu jesteśmy na stronie po obsłudze znacznika niestandardowego.
</body></html>
```

Znacznik przetwarzany w przedstawionej powyżej metodzie `doTag()` (wewnątrz której zgłaszany jest wyjątek `SkipPageException`).



Co się pojawi w przeglądarce, jeśli zmienna `cosNieZadzialalo` przyjmie wartość `true`?

Narysuj odpowiedź w poniższym oknie przeglądarki.



# W odpowiedzi jest wyświetlane wszystko, co zostało wygenerowane przed zgłoszeniem wyjątku SkipPageException

W odpowiedzi będzie zapisany cały kod, który został wygenerowany w metodzie doTag() przed zgłoszeniem wyjątku SkipPageException, jednak po zgłoszeniu tego wyjątku nie będzie przetwarzana ani dalsza część metody doTag(), ani dalsza część strony.

## W kodzie JSP

```
<%@ taglib prefix="mojeZnaczniki" uri="znacznikiProste" %>
```

```
<html><body>
```

```
 Zaraz wywołamy znacznik zgłaszający wyjątek SkipPageException

```

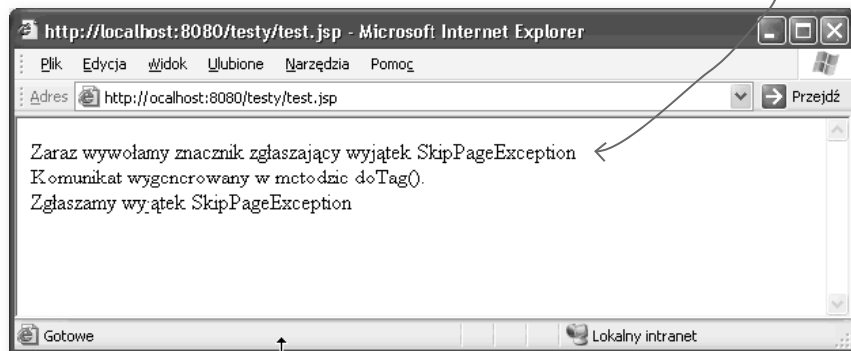
```
 <mojeZnaczniki:znacznikProsty6/>
```

```

Znowu jesteśmy na stronie po obsłudze znacznika niestandardowego.
```

```
</body></html>
```

*Ten kod nie zostanie wyświetlony!*



## W klasie obsługi znacznika

```
public void doTag() throws JspException, IOException {
 {
 getJspContext().getOut().print("Komunikat wygenerowany w metodzie doTag().
");
 getJspContext().getOut().print("Zgłaszamy wyjątek SkipPageException");
 if (cosNieZadzialalo) {
 throw new SkipPageException ();
 }
 }
}
```

## Ale co się stanie, jeśli znacznik zostanie umieszczony w dołączanej stronie?



### Zaostrz ołówki

Przeanalizuj poniższe fragmenty kodu i spróbuj określić, co się stanie w momencie próby wyświetlenia strony StronaA.

*Podpowiedź: Przeanalizuj dokumentację klasy `javax.servlet.jsp.SkipPageException`.*

Na poniższym rysunku napisz, co zostanie wyświetlone w przeglądarce:



### StronaA, do której jest dołączana StronaB

```
<html><body>
```

To jest strona (A), do której jest dołączana inna strona (B).

A teraz dołączamy:<br>

```
<jsp:include page="StronaB.jsp" />
```

<br>Strona A po zakończeniu dołączania...

```
</body></html>
```

### Kod JSP strony StronaB (dołączanej do StronaA), w której jest używany nasz znacznik

```
<%@ taglib prefix="mojeZnaczniki" uri="znacznikiProste" %>
```

To jest strona B, w której jest używany znacznik zgłaszający wyjątek `SkipPageException`.<br>

Teraz używany znacznika:...<br>

```
<mojeZnaczniki:znacznikProsty6/>
```

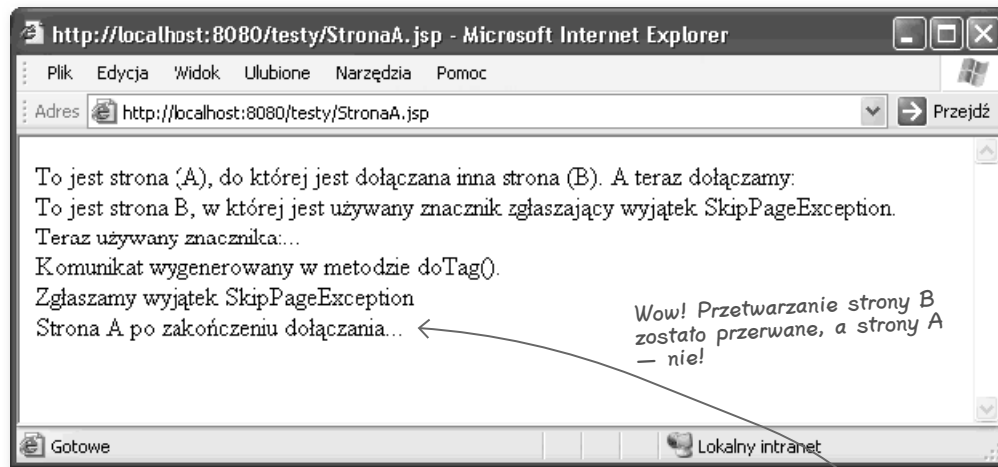
<br>Strona B po wywołaniu znacznika...

### Metoda `doTag()` klasy obsługi znacznika

```
public void doTag() throws JspException, IOException {
 getJspContext().getOut().print("Komunikat wygenerowany w metodzie doTag().
");
 getJspContext().getOut().print("Zgłaszamy wyjątek SkipPageException");
 throw new SkipPageException();
}
```

# Wyjątek SkipPageException przerywa przetwarzanie tylko tej strony, w której został umieszczony znacznik

Jeśli strona wywołująca znacznik jest dołączana w jakiejś innej stronie, przerwane przetwarzanie dotyczy tylko strony wywołującej (strony, w której kodzie umieszczono kłopotliwy znacznik). Nadrzędna strona, do której jest dołączana strona ze znacznikiem, jest przetwarzana bez względu na zgłoszenie wyjątku SkipPageException.



## StronaA, do której jest dołączana StronaB

```
<html><body>
 To jest strona (A), do której jest dołączana inna strona (B).
 A teraz dołączamy:

 <jsp:include page="StronaB.jsp" />

Strona A po zakończeniu dołączania...
</body></html>
```

*Czy zaskoczyło Cię wyświetlenie tego wiersza ze strony A?*

## Kod JSP strony StronaB (dołączanej do StronaA), w której jest używany nasz znacznik

```
<%@ taglib prefix="mojeZnaczniki" uri="znacznikiProste" %>
To jest strona B, w której jest używany znacznik zgłaszający wyjątek SkipPageException.

Teraz używany znacznika:...

<mojeZnaczniki:znacznikProsty6/>
```

**<br>Strona B po wywołaniu znacznika...**

*Ten wiersz, zgodnie z naszymi przewidywaniami, nie został wyświetlony.*

## Metoda doTag() klasy obsługi znacznika

```
public void doTag() throws JspException, IOException {
 getJspContext().getOut().print("Komunikat wygenerowany w metodzie doTag().
");
 getJspContext().getOut().print("Zgłaszamy wyjątek SkipPageException");
 throw new SkipPageException();
}
```

*Zgłoszenie tego wyjątku powoduje przerwanie przetwarzania strony B, lecz nie strony A.*

## Nie ma niemądrych pytań

**❓ Co się dzieje z obiektem typu SimpleTag po zakończeniu wykonywania metody doTag()? Czy kontener przechowuje go gdzieś i używa ponownie w przyszłości?**

**🕒** Nie. Obiekty tego typu nigdy nie są używane wielokrotnie! Każdy obiekt obsługuje tylko jedno wywołanie znacznika. Dzięki temu nigdy nie trzeba przejmować się tym, czy, przykładowo, składowe obiektu będą miały poprawne wartości początkowe. Obiekt typu SimpleTag zawsze będzie inicjalizowany przed wywołaniem którejkolwiek z jego metod.

**❓ Czy metody ustawiające wartości atrybutów w obiekcie klasy SimpleTag muszą operować na danych, które można automatycznie konwertować na łańcuchy? Innymi słowy, czy musimy się posługiwać wyłącznie typami podstawowymi oraz wartościami typu String?**

**🕒** Chyba ktoś nie uważał kilka stron wcześniej. Przecież na wejściu metody ustawiającej jednej z takich klas przekazywaliśmy tablicę filmów. A zatem odpowiedź na to pytanie brzmi — „nie”. Z drugiej strony, jeśli atrybut (który można postrzegać jak *właściwość*, jeśli klasę obsługi znacznika potraktujemy jako komponent JavaBean) nie jest ani egzemplarzem typu prostego, ani łańcuchem, elementowi `<rtexprval>` w deskrytorze TLD najlepiej jest przypisać wartość `true`. Takie ustawienie jest niezbędne, jeśli planujemy przypisywanie atrybutowi wartości, których nie da się wyrazić w formie łańcuchowej. Innymi słowy, przekazanie

znacznikowi obiektu klasy `Pies` byłoby niemożliwe, gdybyśmy musieli się ograniczać do stałych łańcuchowych. Jeśli jednak w roli wartości atrybutu będzie można stosować wyrażenia, wystarczy, że zwracane przez nie wartości będą obiektami typów zgodnych z typem argumentu odpowiedniej metody ustawiającej.

**❓ Jeśli deklaracja znacznika dopuszcza możliwość podania w nim zawartości, lecz w kodzie JSP znacznik jest pusty (gdyż nie ma możliwości zadeklarowania, że zawartość znacznika jest niezbędną), to czy będzie wywoływana metoda setJspBody() klasy obsługi tego znacznika?**

**🕒** Nie! Metoda `setJspBody()` jest wywoływana wyłącznie, gdy są spełnione dwa poniższe warunki:

1. W deskrytorze TLD nie zadeklarowano, że znacznik ma być pusty.
2. W wywołaniu znacznika podano jego zawartość.

To oznacza, że nawet wtedy, gdy w deskrytorze zadeklarujemy, że znacznik może zawierać jakieś ciało, metoda `setJspBody()` nie zostanie wywołana, jeśli wywołanie znacznika przyjmie jedną z poniższych postaci:

`<test:bar />` (znacznik pusty)

`<test:bar></test:bar>` (brak zawartości znacznika)

### KLUCZOWE ZAGADNIENIA



- Pliki znaczników implementują możliwości funkcjonalne znacznika przy wykorzystaniu *pliku*, natomiast znaczniki niestandardowe — przy wykorzystaniu *klasy* obsługi znacznika napisanej w języku Java.
- Istnieją dwa rodzaje klas obsługi znaczników niestandardowych: **klasyczne** i **proste** (przy czym proste klasy oraz pliki znaczników wprowadzono w specyfikacji JSP 2.0).
- Aby stworzyć klasę obsługi znacznika prostego, należy opracować klasę dziedziczącą po **SimpleTagSupport** (która z kolei implementuje interfejs **SimpleTag**).
- Wdrożenie klasy obsługi znacznika prostego wymaga utworzenia deskryptora TLD opisującego znacznik przy użyciu elementu `<tag>`, tego samego, który jest używany w JSTL oraz wszelkich innych bibliotekach znaczników.
- Aby w znaczniku prostym można było umieszczać zawartość, trzeba się upewnić, że w elemencie `<tag>` deskryptora element `<body-content>` nie ma wartości `empty`. W celu przetworzenia zawartości znacznika należy wywołać metodę **getJspBody().invoke()**.
- Klasa **SimpleTagSupport** zawiera implementację wszystkich metod interfejsu **SimpleTag** oraz trzy dodatkowe metody pomocnicze, w tym **getJspBody()**, której można użyć do uzyskania dostępu do zawartości (ciała) znacznika.
- Cykl życia znacznika prostego: **Ponieważ znaczniki proste nigdy nie są wielokrotnie używane**, każde wywołanie znacznika powoduje utworzenie nowego obiektu obsługi i wywołanie jego metody **setJspContext()**. Jeśli znacznik został umieszczony i jest wywoływany wewnątrz innego znacznika, to zostanie wywołana jego metoda **setParent()**. Jeśli w wywołaniu znacznika zostały podane jakieś atrybuty, to wywołane zostaną odpowiednie metody „set” jego klasy obsługi. Jeśli znacznik ma jakąś zawartość (zakładając przy tym, że deklaracja znacznika podana w deskrytorze dopuszcza taką możliwość), wywoływana jest metoda **setJspBody()**. Ostatnim krokiem jest wywołanie metody **doTag()**, a po jej zakończeniu obiekt jest usuwany.
- **Metoda setJspBody() zostanie wywołana wyłącznie w sytuacji, gdy znacznik faktycznie ma zawartość.** Jeśli znacznik nie będzie mieć zawartości, czyli jeśli zostanie zapisany jako pusty — `<moje:znacznik />` — lub nie zostanie podana jego zawartość — `<moje:znacznik></moje:znacznik>`, to metoda **setJspBody()** NIE zostanie wywołana. Pamiętaj, jeśli znacznik ma mieć zawartość, to deskryptor TLD musi zawierać informację na ten temat, a element `<body-content>` nie może zawierać wartości `empty`.
- W metodzie **doTag()** znacznika prostego można określać atrybuty wykorzystywane następnie w zawartości znacznika. W tym celu należy się posłużyć metodą **getJspContext().setAttribute()**, a następnie wywołać metodę **getJspBody().invoke()**.
- **Metoda doTag() deklaruje wyjątki JspException oraz IOException**, zatem z poziomu tej metody można wywoływać metody obiektu **JspWriter** bez konieczności umieszczania tych wywołań w blokach `try-catch`.
- Istnieje możliwość wielokrotnego przetwarzania zawartości znacznika prostego — w tym celu wystarczy w pętli wywoływać metodę **getJspBody().invoke()**.
- Jeśli znacznik ma atrybut, to atrybut ten należy zadeklarować w deskrytorze przy użyciu elementu `<attribute>`, a **w klasie obsługi znacznika napisać odpowiednią metodę set**. W momencie wywoływania znacznika metoda **set** zostanie wywołana przed metodą **doTag()**.
- Metoda **getJspBody()** zwraca obiekt typu **JspFragment**, który udostępnia dwie metody: **invoke(java.io.Writer)** oraz **getJspContext()**. Druga z nich zwraca kontekst strony (obiekt typu **JspContext**), za pośrednictwem którego można uzyskać dostęp do obiektu **PageContext** (on z kolei zapewnia dostęp do zmiennych domyślnych oraz atrybutów).
- Przekazanie wartości `null` w wywołaniu metody **invoke()** powoduje, że wygenerowane wyniki zostaną zapisane w odpowiedzi; jeśli jednak chcemy uzyskać bezpośredni dostęp do zawartości znacznika, możemy w wywołaniu tej metody przekazać inny obiekt typu **Writer**.
- Aby przerwać przetwarzanie bieżącej strony, wystarczy zgłosić wyjątek **SkipPageException**. Jeśli strona zawierająca przetwarzany znacznik została dołączona do innej strony, to przetwarzanie tej „dołączającej” strony nie zostanie przerwane.

To cudowne, że twórcy specyfikacji JSP udostępnili nam znaczniki proste i pliki znaczników, ale... hmm... szkoda, że zrobili to po tym, jak moja firma stworzyła około 10 milionów znaczników niestandardowych, posługując się modelem klasycznym...



### Wciąż musimy znać klasyczne klasy obsługi znaczników niestandardowych

Możesz mieć szczęście. Może Twoja firma używa technologii JSP 2.0, dzięki czemu już od samego początku będziesz mógł posługiwać się plikami znaczników oraz prostymi klasami obsługi znaczników niestandardowych.

*Hipotetycznie* to jest możliwe.

Jednak zapewne tak się nie stanie. Najprawdopodobniej pracujesz, lub będziesz pracować, w firmie, która korzysta z JSP w wersji wcześniejszej niż 2.0 i w której klasy obsługi znaczników niestandardowych są tworzone w modelu klasycznym.

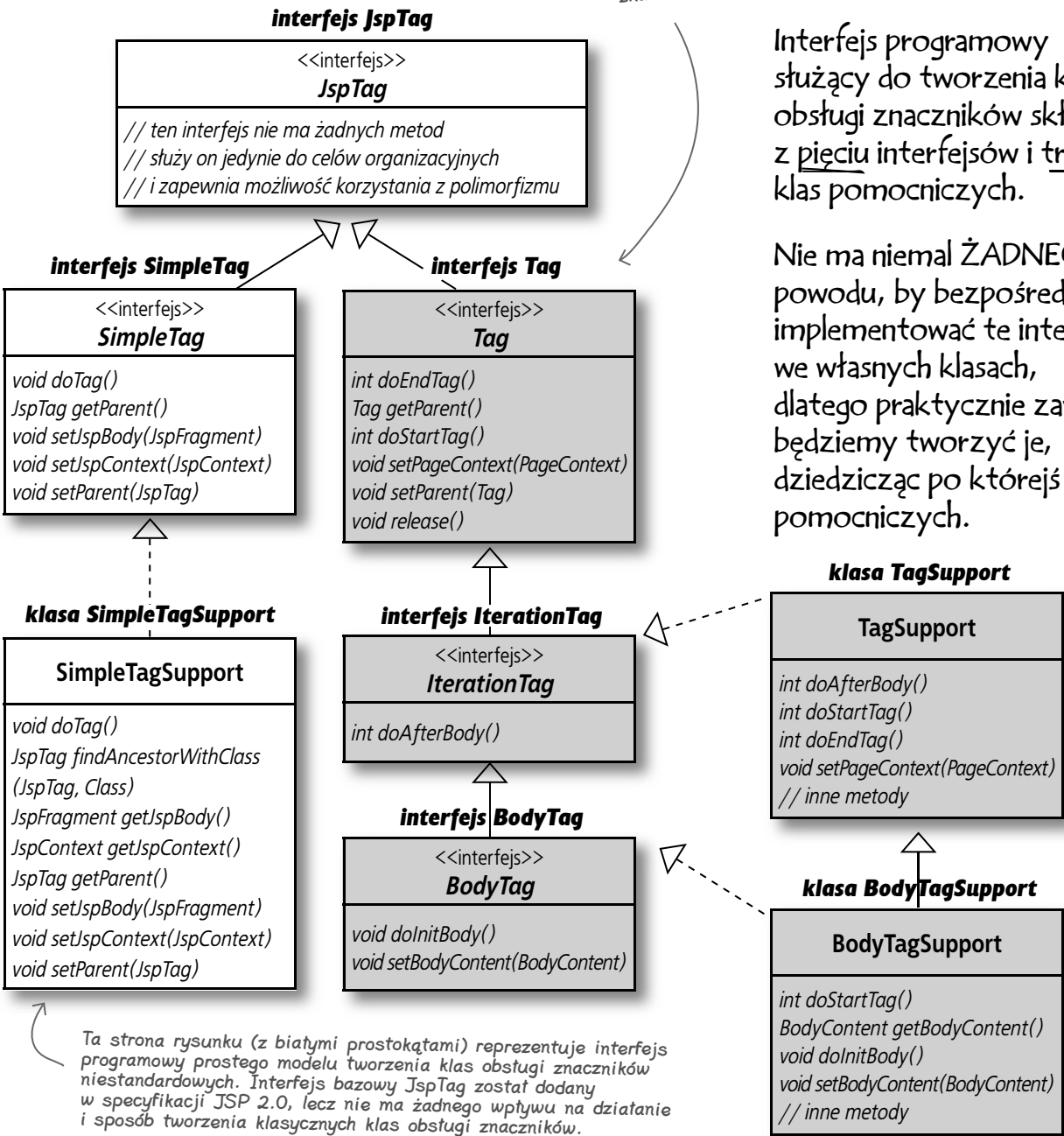
Prawdopodobnie musisz znać ten model na tyle, by przynajmniej być w stanie przeanalizować kod takich klas. Być może będziesz musiał utrzymywać te klasy lub modyfikować i poprawiać ich kod.

Niemniej jednak, nawet jeśli nie musisz ani analizować, ani tworzyć klasycznych klas obsługi znaczników niestandardowych, to wciąż są one (bardzo pobieżnie) uwzględniane podczas egzaminów. Możesz jednak być wdzięczny losowi — we wcześniejszych wersjach egzaminu można było znaleźć co najmniej siedem lub osiem pytań dotyczących klas obsługi znaczników tego typu. Obecnie nie pojawiają się więcej niż dwa pytania na ich temat.

# Interfejs programowy klas obsługi znaczników

Na poniższym rysunku wszystkie klasy i interfejsy należące do klasycznego modelu tworzenia klas obsługi znaczników niestandardowych zostały oznaczone kolorem szarym.

Ta strona rysunku (z szarymi prostokątami) zawiera klasy i interfejsy wchodzące w skład klasycznego modelu tworzenia klas obsługi znaczników niestandardowych.



Interfejs programowy służący do tworzenia klas obsługi znaczników składa się z pięciu interfejsów i trzech klas pomocniczych.

Nie ma niemal **ŻADNEGO** powodu, by bezpośrednio implementować te interfejsy we własnych klasach, dlatego praktycznie zawsze będziemy tworzyć je, dziedzicząc po którejs z klas pomocniczych.

## Bardzo prosta klasa obsługi klasycznego znacznika niestandardowego

Przedstawiony tu przykład jest tak prosty, że niemal nie różni się od metody `doTag()` implementowanej w klasach obsługi znaczników niestandardowych tworzonych w modelu prostym. W rzeczywistości różnice, jakie pojawiają się w klasach tworzonych w modelu klasycznym, staną się uciążliwe dopiero podczas obsługi zawartości znacznika (ale do tego momentu musisz jeszcze chwilę poczekać).

### Kod JSP wywołujący klasyczny znacznik

```
<%@ taglib prefix="moje" uri="znacznikiKlasyczne" %>
<html></body>
 Pierwszy znacznik klasyczny:

 <moje:znacznikKlasyczny1 />
</body></html>
```

Ten znacznik jest obsługiwany przez klasę stworzoną w modelu klasycznym, jednak w kodzie JSP wygląda on tak jak każdy inny znacznik.

### Element `<tag>` deskryptora TLD opisujący klasyczny znacznik

```
<tag>
 <description>Skrajnie uproszczony przykład użycia znacznika klasycznego</description>
 <name>znacznikKlasyczny1</name>
 <tag-class>tmp.ZnacznikKlasyczny1</tag-class>
 <body-content>empty</body-content>
</tag>
```

Na tej podstawie nie można stwierdzić, czy ten znacznik jest obsługiwany przez klasę napisaną w modelu klasycznym, chyba że wiemy, iż klasa `tmp.ZnacznikKlasyczny1` implementuje interfejs `Tag`, a nie `SimpleTag`. Moglibyśmy całkowicie zmienić kod klasy `tmp.znacznikKlasyczny1` tak, by implementowała interfejs `SimpleTag`, a dla deskryptora i tak nie miałyby to żadnego znaczenia.

### Klasa obsługi znacznika w modelu klasycznym

```
package tmp;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;

public class ZnacznikKlasyczny1 extends TagSupport
{
 public int doStartTag() throws JspException {
 JspWriter out = pageContext.getOut();
 try {
 out.println("Wyniki klasycznej klasy obsługi znacznika.");
 } catch (IOException ex) {
 throw new JspException("IOException - " + ex.toString());
 }
 return SKIP_BODY;
 }
}
```

Dziedzicząc po klasie `TagSupport`, implementujemy zarówno interfejs `Tag`, jak i `IterationTag`. W tym przykładzie nadpiszemy tylko jedną metodę — `doStartTag()`.

Metody deklarują wyjątek `JspException`, lecz nie `IOException`! (Metoda `doTag()` interfejsu `SimpleTag` deklaruje także wyjątek `IOException`).

Znaczniki klasyczne dziedziczą po klasie `TagSupport` zmienną składową nazwaną `pageContext` (co odróżnia je od klas tworzonych w modelu prostym, które mogą uzyskiwać kontekst przy użyciu metody `getJspContext()` należącej do klasy `SimpleTagSupport`).

W tym przypadku musimy użyć bloku `try-catch`, gdyż nie możemy zadeklarować wyjątku typu `IOException`.

Metoda musi zwrócić wartość całkowitą, na podstawie której kontener określi, jakie czynności należy wykonać później. W dalszej części rozdziału poświęcimy temu zagadnieniu jeszcze wiele uwagi...

## Klasyczna klasa obsługi znacznika zawierająca DWIE metody

W przykładzie przedstawionym na tej stronie nadpiszemy dwie metody — `doStartTag()` oraz `doEndTag()` — choć wykonywane przez nie operacje można by z powodzeniem zrealizować, używając tylko pierwszej z nich. Chodzi o to, byś wiedział, że metoda `doEndTag()` jest wywoływana *po* przetworzeniu zawartości znacznika. W poniższym przykładzie pominęliśmy deskryptor, gdyż oprócz nazwy niczym się on nie różni od poprzedniego. Deklaracja znacznika zakłada brak atrybutów i zawartości.

### Kod JSP wywołujący klasyczny znacznik

```
<%@ taglib prefix="moje" uri="znacznikiKlasyczne" %>
<html></body>
 Drugi znacznik klasyczny:

 <moje:znacznikKlasyczny2 />
</body></html>
```

Od tej pory nie będziemy już prezentować ani deklaracji pakietu, ani instrukcji importu, chyba że coś się w nich zmieni.

### Klasa obsługi znacznika w modelu klasycznym

```
public class ZnacznikKlasyczny2 extends TagSupport {
 JspWriter out;

 public int doStartTag() throws JspException {
 out = pageContext.getOut();

 try {
 out.println("w metodzie doStartTag()");
 } catch (IOException ex) {
 throw new JspException("IOException - " + ex.toString());
 }

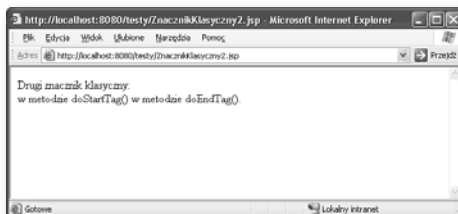
 return SKIP_BODY;
 }

 public int doEndTag() throws JspException {
 try {
 out.println("w metodzie doEndTag().");
 } catch (IOException ex) {
 throw new JspException("IOException - " + ex.toString());
 }

 return EVAL_PAGE;
 }
}
```

Ta wartość wynikowa przekazuje kontenerowi wiadomość: „Nawet jeśli znacznik ma jakąś zawartość, to nie przetwarzaj jej — przejdź prosto do metody `doEndTag()`”.

Ta wartość wynikowa przekazuje następującą informację: „Przetwórz dalszą część strony” (w odróżnieniu od wartości `SKIP_PAGE`, której użycie miałoby podobny efekt do zgłoszenia wyjątku `SkipPageException` w klasie obsługi stworzonej w modelu prostym).



## Kiedy znacznik ma zawartość. Porównanie obu modeli klas obsługi znaczników

Teraz pojawiają się najpoważniejsze różnice pomiędzy klasycznym a prostym modelem klas obsługi znaczników niestandardowych. Pamiętaj, że w przypadku klas tworzonych w modelu prostym zawartość znacznika jest przetwarzana wtedy, kiedy tego chcemy (jeśli w ogóle) — wystarczy wywołać metodę `invoke()` obiektu `JspFragment` reprezentującego zawartość znacznika. Jednak w modelu klasycznym **zawartość znacznika jest przetwarzana po wywołaniu metody `doStartTag()` i przed wywołaniem metody `doEndTag()`**! Oba przedstawione poniżej przykłady generują identyczne wyniki.

### Kod JSP wywołujący znacznik

```
<%@ taglib prefix="moje" uri="znacznikiKlasyczne" %>
<html></body>
 <moje:znacznikProstyZZawartoscia>
 Zawartość znacznika
 </moje:znacznikProstyZZawartoscia>
</body></html>
```

### Klasa obsługi znacznika napisana w modelu prostym

```
// pakiet i instrukcje importu
public class TestZnacznikaProstego extends SimpleTagSupport {
 public void doTag() throws JspException, IOException {
 getJspContext().getOut().print("Przed zawartością.");
 getJspBody().invoke(null); ← To wywołanie spowoduje przetworzenie
 getJspContext().getOut().print("Po zawartości."); zawartości (ciała) znacznika.
 }
}
```

### Klasa obsługi znacznika napisana w modelu klasycznym

```
// deklaracja pakietu i instrukcje importu
public class TestZnacznikaKlasycznego extends TagSupport {
 JspWriter out;

 public int doStartTag() throws JspException {
 out = pageContext.getOut();

 try {
 out.println("Przed zawartością");
 } catch (IOException ex) {
 throw new JspException("IOException - " + ex.toString());
 }

 return EVAL_BODY_INCLUDE; ← W modelu klasycznym zwrócenie TEJ
 wartości powoduje przetworzenie
 zawartości znacznika!
 }

 public int doEndTag() throws JspException {
 try {
 out.println("Po zawartości.");
 } catch (IOException ex) {
 throw new JspException("IOException - " + ex.toString());
 }

 return EVAL_PAGE;
 }
}
```

Ale jak można iteracyjnie przetwarzać zawartość znacznika? Wygląda na to, że metoda `doStartTag()` jest wywoływana za wcześnie, a metoda `doEndTag()` za późno, a poza tym nie ma żadnego sposobu, by zażądać ponownego przetworzenia zawartości...



### Znacznik prosty

```
// pakiet i instrukcje importu
public class TestZnacznikaProstego extends SimpleTagSupport {
 public void doTag() throws JspException, IOException {
 for (int i = 0; i < 3; i++) {
 getJspBody().invoke(null);
 }
 }
}
```

*Iteracyjne przetwarzanie zawartości (ciała) znacznika prostego jest bardzo łatwe — wystarczy w metodzie `doTag()` cyklicznie wywoływać metodę `invoke()` obiektu `JspFragment` reprezentującego tę zawartość.*

### Znacznik klasyczny

```
// pakiet i instrukcje importu
public class TestZnacznikaKlasycznego extends TagSupport {
 public int doStartTag() throws JspException {
 return EVAL_BODY_INCLUDE;
 }

 public int doEndTag() throws JspException {
 return EVAL_PAGE;
 }
}
```

*Ale gdzie umieścić pętlę przetwarzającą ciało znacznika, skoro proces przetwarzania ma miejsce gdzieś pomiędzy wywołaniami metod, a nie w wywołaniu metody (jak w przypadku metody `doTag()`)?*

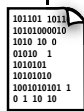
## Znaczniki klasyczne cechują się innym cyklem życia

Znaczniki proste są... proste — stworzenie takiego znacznika sprowadza się do zaimplementowania metody `doTag()`. Jednak w modelu znaczników klasycznych istnieją dwie metody — `doStartTag()` oraz `doEndTag()`. Takie rozwiązanie prowadzi do dość interesującego problemu — kiedy i jak jest przetwarzana zawartość znacznika? Nie ma metody `doBody()`, choć istnieje metoda **`doAfterBody()`** wywoływana *po* przetworzeniu zawartości znacznika i przed wywołaniem metody `doEndTag()`.

### Kontener



### Klasa obsługi znacznika klasycznego



KlasaObslugi.class

### Obiekt obsługi znacznika



Wczytanie klasy.

Utworzenie obiektu  
(przy użyciu konstruktora bezargumentowego).

Wywołanie metody `setPageContext(PageContext)`.

Jeśli znacznik jest zagnieżdżony  
(umieszczony wewnątrz innego znacznika),  
zostaje wywołana metoda `setParent(Tag)`.

Jeśli znacznik ma atrybuty, wywoływane są  
odpowiednie metody „set”.

Wywołanie metody `doStartTag()`.

Jeśli w deklaracji znacznika NIE określono, że ma  
być pusty, I jeśli w kodzie strony znacznik NIE  
zawiera ciała, I jeśli metoda `doStartTag()` zwraca  
wartość `EVAL_BODY_INCLUDE`, następuje  
przetworzenie zawartości tego znacznika.

Jeśli zawartość znacznika została przetworzona,  
wywoływana jest metoda `doAfterBody()`.

Wywołanie metody `doEndTag()`.

Klasa wczytywana jest w momencie  
pierwszego wywołania znacznika,  
jednak kontener (w zależności od  
okoliczności) może wielokrotnie  
wykorzystywać ten sam obiekt obsługi  
znacznika klasycznego.

Wywołanie tej metody przekazuje  
referencję do obiektu `PageContext`.

Znacznik zagnieżdżony może  
komunikować się z innymi znacznikami,  
wewnątrz których został umieszczony.

Jeśli w znaczniku zostały podane jakieś  
atrybuty, to dla każdego z nich wywoływana  
jest odpowiednia metoda `set` (podobnie jak  
w przypadku znaczników niestandardowych  
tworzonych w modelu prostym).

Zawartość znacznika jest przetwarzana  
po wywołaniu metody `doStartTag()` oraz  
przed wywołaniem metody `doEndTag()`.

Metoda `doAfterBody()` pozwala na wykonywanie  
pewnych operacji *PO* przetworzeniu zawartości  
znacznika i w odróżnieniu od innych metod  
może być wywoływana więcej niż raz.

Metoda `doEndTag()` zawsze jest  
wywoływana tylko raz, albo po  
wywołaniu metody `doStartTag()`, albo  
po wywołaniu metody `doAfterBody()`.

# Cykl życia znacznika klasycznego zależy od zwracanych wartości

Metody `doStartTag()` oraz `doEndTag()` zwracają wartości typu `int`. Na ich podstawie kontener określa, co należy dalej zrobić. W przypadku metody `doStartTag()` na podstawie tej wartości kontener znajduje odpowiedź na pytanie: „Czy powinienem przetwarzać zawartość znacznika?” (oczywiście zakładając, że znacznik ma jakąś zawartość oraz że w deskrytorze został zadeklarowany jako znacznik, który takie ciało może zawierać).

W przypadku metody `doEndTag()` kontener odpowiada na pytanie: „Czy powinienem przetwarzać dalszą część strony, na której został umieszczony aktualnie przetwarzany znacznik?” Wartości zwracane przez te metody zostały zadeklarowane jako stałe w interfejsach `Tag` oraz `IterationTag`.

## Możliwe wartości wynikowe w przypadku dziedziczenia po klasie `TagSupport`

`doStartTag()`  
`SKIP_BODY`  
`EVAL_BODY_INCLUDE`

`doAfterBody()`  
`SKIP_BODY`  
`EVAL_BODY_AGAIN`

`doEndTag()`  
`SKIP_PAGE`  
`EVAL_PAGE`

To jest jedyna wartość wynikowa zadeklarowana w interfejsie `IterationTag` (pozostałe zostały zadeklarowane w interfejsie `Tag`).

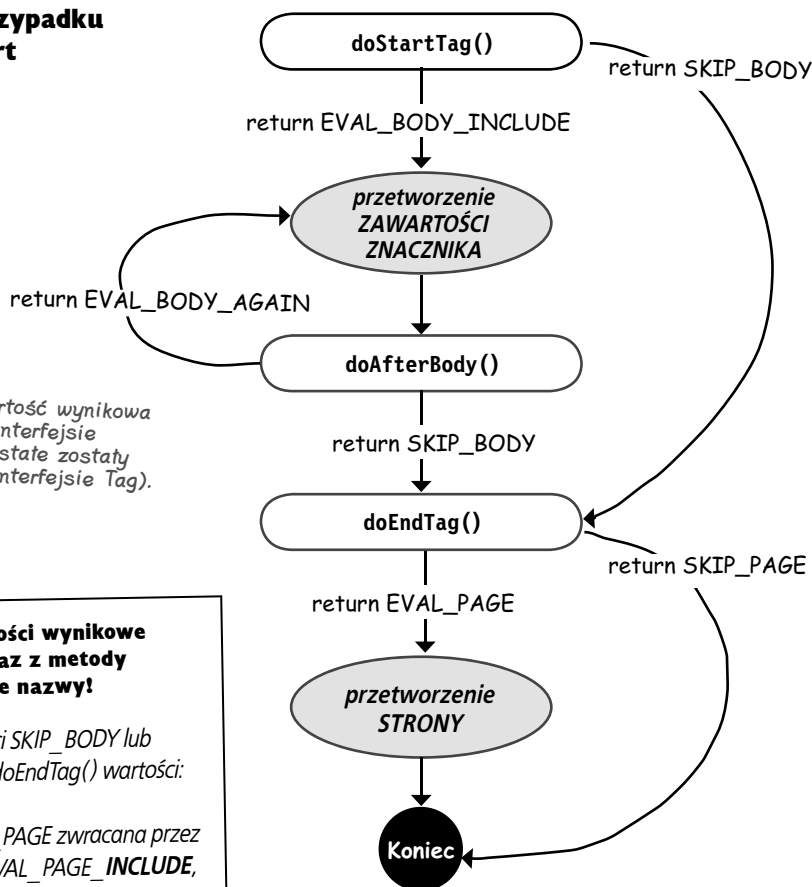


Oglądaj to!

**Stałe zwracane jako wartości wynikowe z metody `doStartTag()` oraz z metody `doEndTag()` mają niespójne nazwy!**

Metoda `doStartTag()` może zwracać wartości `SKIP_BODY` lub `EVAL_BODY_INCLUDE`, natomiast metoda `doEndTag()` wartości: `SKIP_PAGE` lub `EVAL_PAGE`.

Chcąc zachować spójność nazw, stała `EVAL_PAGE` zwracana przez metodę `doEndTag()` powinna nazywać się `EVAL_PAGE_INCLUDE`, gdyż wtedy odpowiadałaby stałej `EVAL_BODY_INCLUDE` zwracanej przez metodę `doStartTag()`. Twórcy omawianych interfejsów zdecydowali się jednak na inne rozwiązanie! Nie daj się zatem wprowadzić w błąd, kiedy na egzaminie zobaczysz poprawnie wyglądające (lecz błędne) nazwy stałych.

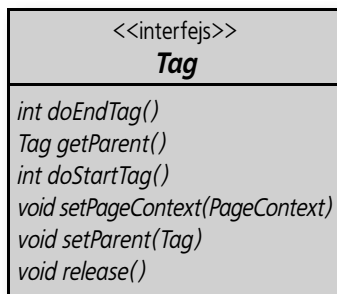


Zwrócenie wartości `SKIP_PAGE` z metody `doEndTag()` ma identyczny skutek co zgłoszenie wyjątku `SkipPageException` w metodzie `doTag()` klasy obsługi znacznika tworzonej w modelu prostym. Jeśli strona, na której został umieszczony znacznik, jest dotaczana do innej strony, przerwanie przetwarzania dotyczy tylko strony zawierającej znacznik.

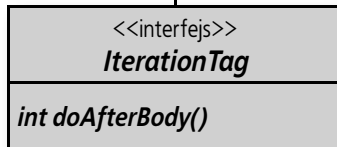
## Interfejs `IterationTag` pozwala na wielokrotne przetwarzanie zawartości znacznika

Tworząc klasę obsługi znacznika dziedziczącą po `TagSupport`, możemy korzystać ze wszystkich metod cyklu życia znacznika zdefiniowanych w interfejsie `Tag` oraz z jednej dodatkowej metody — `doAfterBody()` — wchodzącej w skład interfejsu `IterationTag`. Gdyby metoda `doAfterBody()` nie była dostępna, nie mielibyśmy możliwości iteracyjnego przetwarzania ciała znacznika, ponieważ w metodzie `doStartTag()` jest na to za wcześnie, a w metodzie `doEndTag()` za późno. Jednak wartość wynikowa zwracana przez metodę `doAfterBody()` informuje kontener, czy należy powtórzyć przetwarzanie zawartości (wartość `EVAL_BODY_AGAIN`) czy wywołać metodę `doEndTag()` (wartość `SKIP_BODY`).

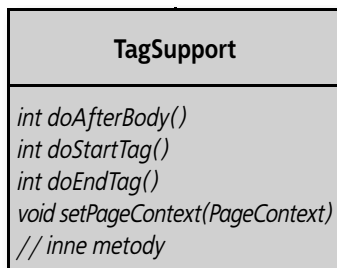
### interfejs `Tag`



### Interfejs `IterationTag`



### Klasa `TagSupport`



### Zaostrz ołówek

Spróbuj zaimplementować funkcjonalność metody `doTag()` poniższej klasy obsługi znacznika prostego. Przyjmij, że deskryptor TLD zezwala na zawieranie ciała przez znacznik.

```

public void doTag() throws JspException, IOException {
 String[] filmy = {"Spiderman", "Shrek", "Amelia"};
 for (int i = 0; i < filmy.length; i++) {
 getJspContext().setAttribute("film", filmy[i]);
 getJspBody().invoke(null);
 }
}

```

```

// deklaracja pakietu i instrukcje importu
public class MojZnacznikPowtarzany extends TagSupport {

```

```

 public int doStartTag() throws JspException {

```

```

 public int doAfterBody() throws JspException {

```

```

 public int doEndTag() throws JspException {

```

```

 }

```



### BĄDŹ kontenerem

Przeanalizuj przedstawiony poniżej poprawny kod klasy obsługi znacznika i spróbuj określić, czy wygeneruje wyniki pokazane na rysunku u dołu strony. Przyjmij, że znacznik jest wywoływany w sposób pokazany w przykładowym kodzie JSP zamieszczonym u dołu strony. Takie same wyniki ma generować znacznik prosty przedstawiony na poprzedniej stronie. Tak, odpowiedź na to pytanie zostanie zamieszczona w dalszej części rozdziału wraz z rozwiązaniem innego ćwiczenia...

### Klasa obsługi znacznika

*// deklaracja pakietu i instrukcje importu*

```
public class MojZnacznikPowtarzany extends TagSupport {
 String[] filmy = new String[] {"Spiderman", "Shrek", "Amelia"};
 int licznikFilmow;

 public int doStartTag() throws JspException {
 licznikFilmow = 0;
 return EVAL_BODY_INCLUDE;
 }

 public int doAfterBody() throws JspException {
 if (licznikFilmow < filmy.length) {
 pageContext.setAttribute("film", filmy[licznikFilmow]);
 licznikFilmow++;
 return EVAL_BODY_AGAIN;
 } else {
 return SKIP_BODY;
 }
 }

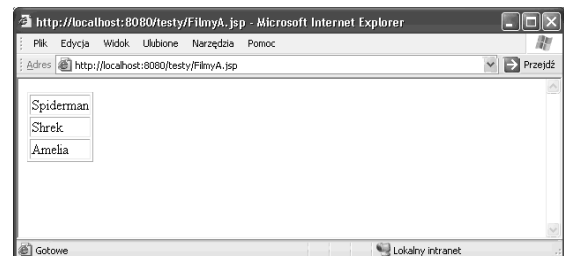
 public int doEndTag() throws JspException {
 return EVAL_PAGE;
 }
}
```

### Kod JSP wywołujący znacznik

```
<%@ taglib prefix="moje" uri="ZnacznikiKlasyczne" %>

<html><body>
 <table border="1">
 <moje:pokazFilmy>
 <tr><td>${film}</td></tr>
 </moje:pokazFilmy>
 </table>
</body></html>
```

### Pożądane wyniki



## Wartości domyślne zwracane przez metody klasy TagSupport

Jeśli tworząc własną klasę obsługi znacznika dziedziczącą po klasie TagSupport, nie decydujesz się na nadpisanie wszystkich metod cyklu życia znacznika, musisz mieć świadomość, jakie będą wartości domyślne zwracane przez oryginalne wersje tych metod. Klasa TagSupport zakłada, że znacznik nie zawiera ciała (stąd wartość SKIP\_BODY zwracana przez metodę doStartTag()), a jeśli już zawiera jakieś ciało, jest ono przetwarzane tylko raz (dlatego metoda doAfterBody() zwraca wartość SKIP\_BODY). Poza tym klasa TagSupport zakłada, że chcemy przetwarzać pozostałą część strony i dlatego metoda doEndTag() zwraca wartość EVAL\_PAGE.

### Domyślne wartości zwracane przez nienadpisane metody klasy TagSupport

doStartTag()

SKIP\_BODY

EVAL\_BODY\_INCLUDE

doAfterBody()

SKIP\_BODY

EVAL\_BODY\_AGAIN

doEndTag()

SKIP\_PAGE

EVAL\_PAGE

Klasa TagSupport zakłada, że znacznik nie ma zawartości, a jeśli ma zawartość i zawartość jest przetwarzana, to należy ją przetworzyć TYLKO RAZ.

Klasa TagSupport przyjmuje też, że zawsze chcemy przetwarzać dalszą część strony.



Oglądaj to!

**Metody doStartTag() oraz doEndTag() są wykonywane tylko raz.**

Znajomość cyklu życia naprawdę jest wymagana na egzaminie. Nie zapomnij, że metody doStartTag() oraz doEndTag() zawsze są wywoływane tylko raz, niezależnie od wszelkich innych wykonywanych operacji. Z kolei metoda doAfterBody() może zostać wywołana dowolną liczbą razy (w tym w ogóle), przy czym liczba tych wywołań zależy od wartości wynikowych zwróconych przez metodę doStartTag() oraz przez wcześniejsze wywołania metody doAfterBody().



Oglądaj to!

**Jeśli chcesz przetworzenia ciała znacznika, MUSISZ nadpisać metodę doStartTag()!**

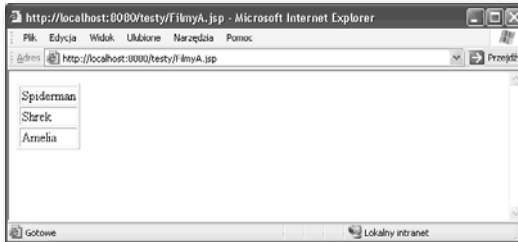
Zastanów się nad tym! Metoda doStartTag() domyślnie zwraca wartość SKIP\_BODY, zatem jeśli Twoja klasa dziedziczy po TagSupport, a chcesz przetworzenia ciała znacznika, musisz nadpisać metodę doStartTag() (choćby po to, by zwracała wartość EVAL\_BODY\_INCLUDE).

W tej sytuacji jest oczywiste, że jeśli chcesz przetworzyć zawartość znacznika więcej niż raz, musisz nadpisać także metodę doAfterBody(), która domyślnie zwraca wartość SKIP\_BODY.

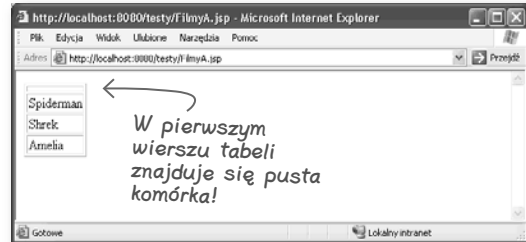


## BĄDŹ kontenerem — rozwiązanie ćwiczenia

### Pożądane wyniki



**Faktyczne wyniki (jeśli nie zostaną dodane dwa wiersze kodu, które w poniższym przykładzie zostały pokazane na czarnym tle).**



### Klasa obsługi znacznika

```
public class MojZnacznikPowtarzany extends TagSupport {
 String[] filmy = new String[] {"Spiderman", "Shrek", "Amelia"};
 int licznikFilmwow;

 public int doStartTag() throws JspException {
 licznikFilmwow = 0;

 pageContext.setAttribute("film", filmy[licznikFilmwow]);
 licznikFilmwow++;

 return EVAL_BODY_INCLUDE;
 }

 public int doAfterBody() throws JspException {
 if (licznikFilmwow < filmy.length) {
 pageContext.setAttribute("film", filmy[licznikFilmwow]);
 licznikFilmwow++;
 return EVAL_BODY_AGAIN;
 } else {
 return SKIP_BODY;
 }
 }

 public int doEndTag() throws JspException {
 return EVAL_PAGE;
 }
}
```

*Aby wyniki były zgodne z oczekiwaniami, MUSISZ dodać te dwa wiersze kodu.*

### Kod JSP wywołujący znacznik

```
<%@ taglib prefix="moje"
uri="ZnacznikiKlasyczne" %>
<html><body>
 <table border="1">
 <moje:pokazFilmy>
 <tr><td>${film}</td></tr>
 </moje:pokazFilmy>
 </table>
</body></html>
```

*Przedstawiona metoda doAfterBody() była co prawda poprawna, ale musimy pamiętać, że jej wywołanie następuje już PO jednym przetworzeniu ciała znacznika! Bez dwóch wierszy kodu dodanych w metodzie doStartTag(), podczas pierwszego przetwarzania zawartości znacznika nie jest określona wartość atrybutu film, przez co pierwsza komórka tabeli jest pusta.*

## Nie ma niemądrych pytań

**P.** To rozwiązanie nie wydaje się najlepsze, gdyż w metodach `doStartTag()` oraz `doAfterBody()` musimy umieścić ten sam fragment kodu.

**U.** Owszem, to prawda. W takim przypadku, jeśli dziedziczymy po klasie `TagSupport` i chcemy określić wartości atrybutów, które można wykorzystać przy przetwarzaniu zawartości znacznika, musimy je podać w metodzie `doStartTag()`. Nie możemy z tym czekać aż do wywołania metody `doAfterBody()`, gdyż jest ona wywoływana w chwili, kiedy zawartość znacznika zostanie już raz przetworzona.

Być może nie jest to bardzo eleganckie rozwiązanie. To właśnie dlatego prosty model klas obsługi znaczników jest znacznie lepszy. Oczywiście można napisać dodatkową metodę, na przykład `setFilm()`, i wywoływać ją w metodach `doStartTag()` oraz `doAfterBody()`. Niemniej jednak także i to rozwiązanie jest dziwaczne i nieeleganckie.

**P.** DLACZEGO wartość składowej `licznikFilmow` jest określana W metodzie `doStartTag()`? Czy nie można jej zainicjalizować w miejscu deklaracji?

**U.** Ha, właśnie! W odróżnieniu od obiektów obsługi znaczników tworzonych w modelu prostym, które nigdy nie są używane więcej niż raz, obiekty obsługi tworzone w modelu klasycznym mogą być przechowywane w puli i stosowane wielokrotnie. A to oznacza, że należy inicjalizować składowe na początku obsługi każdego nowego znacznika (czyli w metodzie `doStartTag()`). W przeciwnym razie pierwszy znacznik zostanie obsłużony poprawnie, lecz podczas obsługi każdego kolejnego składowa `licznikFilmow` będzie mieć poprzednią wartość (a nie wartość 0).



Oglądaj to!

**Kontener może wielokrotnie używać obiektu klasy obsługi znacznika stworzonej w modelu klasycznym!**

*Uważaj — to jedna z kluczowych różnic pomiędzy modelem klasycznym a modelem prostym, w którym obiekty obsługi znaczników NIGDY nie są używane wielokrotnie. To zaś oznacza, że należy bardzo uważnie inicjalizować i korzystać ze składowych — ich wartości należy ustawiać w metodzie `doStartTag()`. Interfejs `Tag` definiuje metodę `release()`, lecz jest ona wywoływana tylko w chwili, gdy kontener ma zamiar usunąć obiekt obsługi znacznika. Nie należy zatem traktować tej metody jako sposobu na przywrócenie początkowego stanu obiektu pomiędzy obsługą kolejnych znaczników!*

### Dobrze, przejdźmy do rzeczy...

Pamiętasz aplikację porad piwnych z rozdziału 3.? Spróbujmy ją trochę udoskonalić i zautomatyzować część formularza:

```
<form method="POST" action="WyborPiwa.do">
 <p>Wybierz właściwości piwa:</p>
```

Kolor:

```
<select name='kolor' size='1' >
 <option value='jasne'> jasne </option>
 <option value='bursztynowe'> bursztynowe </option>
 <option value='brązowe'> brązowe </option>
 <option value='ciemne'> ciemne </option>
</select>
```

Chcemy, aby zbiór opcji tego znacznika `<select>` pochodził z aplikacji.

```



```

```
<input type="SUBMIT">
```

```
</form>
```

Gdyby opcje były generowane automatycznie, można by łatwiej wprowadzać niezbędne zmiany w aplikacji bez konieczności modyfikowania kodu HTML. Chcielibyśmy więc, aby dostępne opcje były generowane na podstawie struktury Javy typu **List** utworzonej w kodzie aplikacji internetowej. Oto, jak powinien wyglądać odpowiedni znacznik niestandardowy:

```
<form method="POST" action="WyborPiwa.do">
 <p>Wybierz właściwości piwa:</p>
```

Kolor:

```
<formTags:select name='kolor' size='1'
 optionsList='${applicationScope.colorList}' />
```

Listę opcji generuje teraz nasz znacznik niestandardowy. Atrybuty nazwy i rozmiaru zawierają wartości przekazywane dalej w niezmienionej formie.

```



```

```
<input type="SUBMIT">
```

```
</form>
```

Użyty znacznik umożliwia naszej aplikacji modyfikowanie opcji bez konieczności trwałego kodowania danych biznesowych w formularzu języka HTML.



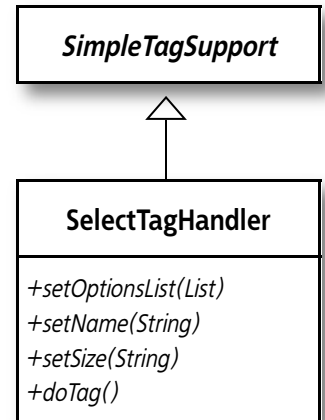
## Zaostrz ołówek

Twoim zadaniem (jeśli zdecydujesz się podjąć wyzwanie) jest dokończenie implementacji klasy obsługi znacznika **select** (słowo **select** pogrubione i podkreślone).

W pierwszej kolejności klasa obsługi powinna implementować po jednej metodzie ustawiającej dla każdego atrybutu tego znacznika; szkielet takiej implementacji przedstawiono poniżej:

```
package com.example.taglib;
// przyjmijmy, że tutaj umieszczono wszystkie niezbędne instrukcje importowania

public class SelectTagHandler extends SimpleTagSupport {
 // ustaw atrybut optionsList
```



```
// ustaw atrybut name
```

Spróbuj samodzielnie dopisać niezbędny kod w pustych miejscach pod komentarzami.

```
// ustaw atrybut size
```

Ciąg dalszy na następnej stronie



### Zaostrz ołówek

W kolejnym kroku musisz dokończyć implementację klasy obsługi znacznika niestandardowego **select**, definiując metodę **doTag()**. Dla ułatwienia zapisaliśmy już sygnaturę tej metody i kilka przydatnych komentarzy. Przyjrzyj się uważnie przedstawionemu na stronie 570 fragmentowi kodu HTML, który ma być generowany przez tę metodę.

```
// wygeneruj znaczniki <select> i <option>
public void doTag() throws JspException, IOException {
 PageContext pageContext = (PageContext) getJspContext();
 JspWriter out = pageContext.getOut();
 // Początek znacznika <select> HTML-a z niezbędnymi atrybutami tego języka
```

Musisz dopisać  
brakujący kod  
tutaj...  
tutaj...  
i tutaj.

```
// Wygeneruj znaczniki <option> na podstawie struktury optionsList
```

```
// Koniec znacznika </select> języka HTML
```

```
} // KONIEC metody doTag()
```

```
} // KONIEC klasy SelectTagHandler
```

Jeśli w klasie *SelectTagHandler*  
potrzebujesz dodatkowych  
stałych lub zmiennych, możesz je  
zdefiniować w tym miejscu.

Ciąg dalszy na następnej stronie →



## Zaostrz ołówek

Musimy teraz skonfigurować znacznik **select** w pliku deskryptora TLD. Powtarzalne, standardowe elementy tego deskryptora napisaliśmy już za Ciebie. Możesz się więc ograniczyć tylko do elementu deklarującego sam znacznik **select**, jego klasę obsługi i wszystkie niezbędne atrybuty.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
```

```
<!DOCTYPE taglib
```

```
 PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
```

```
 "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_2.dtd">
```

```
<taglib>
```

```
 <tlib-version>1.2</tlib-version>
```

```
 <jsp-version>1.2</jsp-version>
```

```
 <short-name>Biblioteka znaczników formularza</short-name>
```

```
 <uri>http://example.com/tags/forms</uri>
```

```
 <description>
```

Przykład biblioteki znaczników stosowanych w miejsce znaczników formularza HTML.

```
 </description>
```

```
<tag>
```

```
 <!-- Dodaj elementy deklarujące nazwę, klasę obsługi i typ ciała znacznika. -->
```

```
 <!-- Dodaj elementy właściwe dla atrybutu optionsList. -->
```

```
 <!-- Dodaj elementy właściwe dla atrybutu name. -->
```

```
 <!-- Dodaj elementy właściwe dla atrybutu size. -->
```

```
</tag>
```

```
</taglib>
```

Dopisz brakujący  
kod XML  
deskryptora TLD.



## Zaostrz ołówek — Rozwiązanie

Twoim zadaniem (jeśli przyjąłeś wyzwanie) było dokończenie implementacji klasy obsługi znacznika niestandardowego **select**. Nasza klasa musi implementować po jednej metodzie ustawiającej dla każdego z atrybutów tego znacznika. Klasa **SelectTagHandler** musi też implementować metodę **doTag()**.

```
package com.example.taglib;
// przyjmijmy, że tutaj umieszczono wszystkie niezbędne
// instrukcje importowania
```

```
public class SelectTagHandler extends SimpleTagSupport {
```

```
 private List optionsList;
 // ustaw atrybut optionsList
 public void setOptionsList(List value) {
 this.optionsList = value;
 }
```

← Metoda ustawiająca i zmienna  
egzemplarza dla atrybutu  
optionsList.

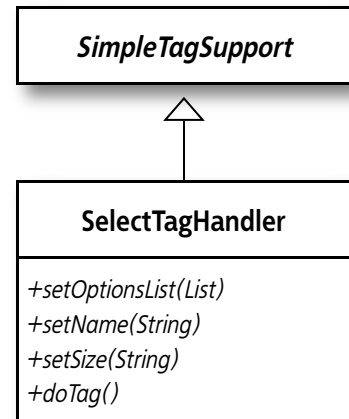
```
 private String name;
 // ustaw atrybut name
 public void setName(String value) {
 this.name = value;
 }
```

← Metoda ustawiająca i zmienna  
egzemplarza dla atrybutu name.

```
 private String size;
 // ustaw atrybut size
 public void setSize(String value) {
 this.size = value;
 }
```

← Metoda ustawiająca i zmienna  
egzemplarza dla atrybutu size.

```
 // dalsza część kodu klasy SelectTagHandler
}
```





## Zaostrz ołówek — Rozwiązanie

Twoim kolejnym zadaniem było uzupełnienie implementacji klasy obsługi znacznika **select** o metodę **doTag()**. Oto prawidłowy kod tej metody:

```
// wygeneruj znaczniki <select> i <option>
public void doTag() throws JspException, IOException {

 PageContext pageContext = (PageContext) getJspContext();

 JspWriter out = pageContext.getOut();

 // Początek znacznika <select> HTML-a z niezbędnymi atrybutami tego języka

 out.print("<select ");
 out.print(String.format(ATTR_TEMPLATE, "name", this.name));
 out.print(String.format(ATTR_TEMPLATE, "size", this.size));
 out.print('>');

 // Wygeneruj znaczniki <option> na podstawie struktury optionsList
 for (Object option : this.optionsList) {

 String optionTag = String.format(OPTION_TEMPLATE, option.toString());

 out.println(optionTag);

 }

 // Koniec znacznika </select> języka HTML
 out.println(" </select>");

} // KONIEC metody doTag()
```

Znacznik  
otwierający  
<select> HTML-a  
zawiera atrybuty  
name i size.

Obiekt optionsList wykorzystujemy  
do utworzenia znaczników <option>  
języka HTML.

I wreszcie klasa obsługi musi  
przekazać na wyjście znacznik  
zamykający </select> języka HTML.

```
private static final String ATTR_TEMPLATE = "%s='%s' ";
private static final String OPTION_TEMPLATE
 = " <option value='%1$s'> %1$s </option>";

} // KONIEC klasy SelectTagHandler
```

W naszej implementacji wykorzystano  
kilką statycznych łańcuchowych, które  
poprawiły czytelność przedstawionego  
kodu.



### Zaostrz ołówek — Rozwiązanie

W kolejnym kroku miałeś za zadanie skonfigurować znacznik **select** w pliku deskryptora TLD. Oto, jak powinien wyglądać prawidłowy deskryptor TLD deklarujący znacznik select, odpowiednią klasę obsługi i wszystkie atrybuty:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
```

```
<!DOCTYPE taglib
```

```
 PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
```

```
 "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_2.dtd">
```

```
<taglib>
```

```
 <tlib-version>1.2</tlib-version>
```

```
 <jsp-version>1.2</jsp-version>
```

```
 <short-name>Biblioteka znaczników formularza</short-name>
```

```
 <uri>http://example.com/tags/forms</uri>
```

```
 <description>
```

Przykład biblioteki znaczników stosowanych w miejsce znaczników formularza HTML.

```
 </description>
```

```
 <tag>
```

*Deklarujemy nazwę, klasę  
i typ ciała naszego znacznika.*

```
 <name>select</name>
```

```
 <tag-class>com.example.taglib.SelectTagHandler</tag-class>
```

```
 <body-content>empty</body-content>
```

```
 <attribute>
```

```
 <name>optionsList</name>
```

```
 <type>java.util.List</type>
```

```
 <required>true</required>
```

```
 <rtexprvalue>true</rtexprvalue>
```

```
 </attribute>
```

*Atrybut optionsList wymaga zadeklarowania typów danych. Atrybut optionsList dodatkowo musi obsługiwać wartości w formie wyrażeń przetwarzanych w czasie wykonywania.*



## Zaostrz ołówek — Rozwiązanie

```
<attribute>
 <name>name</name>
 <required>true</required>
</attribute>
```

```
<attribute>
 <name>size</name>
 <required>true</required>
</attribute>
```

```
</tag>
```

```
</taglib>
```

Deklaracje atrybutów `name` i `size` są nieporównanie prostsze, ponieważ oba atrybuty reprezentują domyślny typ danych (`String`).



## POTĘGA UMYSŁU

Jak myślisz, czy atrybuty `name` i `size` powinny akceptować zmienne wyznaczone w czasie wykonywania? Odpowiedź uzasadnij.

## Nasz dynamiczny znacznik `<select>` jest niekompletny...



Chwileczkę... skoro nasz znacznik wyboru ma udawać standardowy znacznik `<select>` języka HTML, musimy przecież uwzględnić wszystkie atrybuty tego znacznika, nie tylko atrybuty `name` i `size`.

Znacznik `<select>` języka HTML obsługuje dużo więcej atrybutów niż tylko `name` i `size`.

**Atrybuty podstawowe:** `id`, `class`, `style` i `title`

**Atrybuty umiędzynarodawiania:** `lang` i `dir`

**Atrybuty zdarzeń:** `onclick`, `ondblclick`, `onmouseup`, `onmousedown`, `onmouseover`, `onmousemove`, `onmouseout`, `onkeypress`, `onkeyup` i `onkeydown`

**Atrybuty formularza:** `name`, `disabled`, `multiple`, `size`, `tabindex`, `onfocus`, `onblur` i `onchange`

Za pomocą tych atrybutów można tworzyć także listy oraz menu.

Ludzie! Jak miałbym zaimplementować naprawdę wystrzałowe zachowania bez atrybutów zdarzeń?

Moja strona nikogo nie zachwyci, jeśli nie będę mogła korzystać ze stylów.

Nie wiążcie mi rąk... Muszę mieć możliwość stosowania zarówno list, jak i menu.



## Moglibyśmy dodać do naszego znacznika niestandardowego więcej atrybutów...



Nie ma się co przejmować,  
mogę to naprawić...  
to dla mnie żaden problem.

Wystarczy w klasie obsługi znacznika  
zaimplementować dodatkowe metody  
ustawiające atrybuty i uzupełnić deskryptor  
TLD o odpowiednie deklaracje.  
Raz, dwa i po sprawie.

Projekt Gary'ego jest bardzo prosty — musimy tylko dodać po jednej metodzie ustawiającej dla każdego z przekazywanych atrybutów HTML-a. Diagram UML klasy obsługi tego znacznika ze wszystkimi niezbędnymi metodami pokazano po prawej stronie.

Oto, jak powinna wyglądać zmodyfikowana klasa SelectTagHandler:

```
public class SelectTagHandler extends SimpleTagSupport {
 // atrybut znacznika (metody ustawiające i zmienne egzemplarza)
 public void setOptionsList(List value) {
 this.optionsList = value;
 }
 private List optionsList;

 public void setId(String id) {
 this.id = id;
 }
 private String id;

 public void setClass(String styleClass) {
 this.styleClass = styleClass;
 }
 private String styleClass;

 // dalsza część kodu na kolejnej stronie
```

To jedyny atrybut, który  
dodaaliśmy do znacznika  
select.

Pozostałe atrybuty znacznika są  
kierowane do przeglądarki internetowej  
w niezmienionej formie. Klasa obsługi  
ogranicza się do przekazywania ich do  
danych wynikowych znacznika <select>.

### SelectTagHandler

```
+setOptionsList(List)
+setId(String)
+setClass(String)
+setStyle(String)
+setTitle(String)
+setLang(String)
+setDir(String)
+setOnClick(String)
+setOndblclick(String)
+setOnmouseup(String)
+setOnmousedown(String)
+setOnmouseover(String)
+setOnmousemove(String)
+setOnmouseout(String)
+setOnkeypress(String)
+setOnkeydown(String)
+setOnkeyup(String)
+setName(String)
+setSize(String)
+setMultiple(String)
+setDisabled(String)
+setTabIndex(String)
+setOnfocus(String)
+setOnblur(String)
+setOnchange(String)
+doTag()
```

# Efekt obsługi dodatkowych atrybutów znacznika

```
public void setStyle(String style) {
 this.id = style;
}
private String style;

public void setTitle(String title) {
 this.id = title;
}
private String title;

public void setLang(String lang) {
 this.id = lang;
}
private String lang;

public void setDir(String dir) {
 this.id = dir;
}
private String dir;

public void setOnClick(String onclick) {
 this.id = onclick;
}
private String onclick;

public void setOndblclick(String ondblclick) {
 this.id = ondblclick;
}
private String ondblclick;

public void setOnmouseup(String onmouseup) {
 this.id = onmouseup;
}
private String onmouseup;

public void setOnmousedown(String onmousedown) {
 this.id = onmousedown;
}
private String onmousedown;

public void setOnmouseover(String onmouseover) {
 this.id = onmouseover;
}
private String onmouseover;

// dalsza część kodu na kolejnej stronie
```

To kolejne atrybuty znacznika <select> języka HTML. Mamy tutaj do czynienia przede wszystkim z atrybutami podstawowymi oraz kilkoma atrybutami obsługi zdarzeń.

Chwileczkę! Zakodowaliśmy zaledwie 11 z 24 atrybutów HTML-a. Kolejny zbiór metod ustawiających właściwych dla dalszych atrybutów znajdziesz na następnej stronie.

## Efekt obsługi dodatkowych atrybutów znacznika raz jeszcze

```

public void setOnmousemove(String onmousemove) {
 this.id = onmousemove;
}
private String onmousemove;

public void setOnmouseout(String onmouseout) {
 this.id = onmouseout;
}
private String onmouseout;

public void setOnkeypress(String onkeypress) {
 this.id = onkeypress;
}
private String onkeypress;

public void setOnkeydown(String onkeydown) {
 this.id = onkeydown;
}
private String onkeydown;

public void setOnkeyup(String onkeyup) {
 this.id = onkeyup;
}
private String onkeyup;

public void setOndblclick(String ondblclick) {
 this.id = ondblclick;
}
private String ondblclick;

public void setName(String name) {
 this.id = name;
}
private String name;

public void setSize(String size) {
 this.id = size;
}
private String size;

public void setMultiple(String multiple) {
 this.id = multiple;
}
private String multiple;

// jeszcze więcej kodu na kolejnej stronie

```

*Tak, pewnie się domyślasz  
— to nadal nie wszystkie  
metody ustawiające.*



## Mam już dość tych atrybutów znacznika!

```
public void setTabIndex(String tabindex) {
 this.tabindex = tabindex;
}
private String tabindex;

public void setOnfocus(String onfocus) {
 this.onfocus = onfocus;
}
private String onfocus;

public void setOnblur(String onblur) {
 this.onblur = onblur;
}
private String onblur;

public void setOnchange(String onchange) {
 this.onchange = onchange;
}
private String onchange;

// wygeneruj znaczniki <select> i <option>
public void doTag() throws JspException, IOException {
 PageContext pageContext = (PageContext) getJspContext();
 JspWriter out = pageContext.getOut();
 // Początek znacznika <select> HTML-a z niezbędnymi atrybutami tego języka
 out.print("<select ");
 // dodajemy atrybuty wymagane
 out.print(String.format(ATTR_TEMPLATE, "name", this.name));
 // dodajemy atrybuty opcjonalne
 if (this.id != null)
 out.print(String.format(ATTR_TEMPLATE, "id", this.id));
 if (this.styleClass != null)
 out.print(String.format(ATTR_TEMPLATE, "styleClass", this.styleClass));
 if (this.style != null)
 out.print(String.format(ATTR_TEMPLATE, "style", this.style));
 if (this.title != null)
 out.print(String.format(ATTR_TEMPLATE, "title", this.title));
 if (this.lang != null)
 out.print(String.format(ATTR_TEMPLATE, "lang", this.lang));
 if (this.dir != null)
 out.print(String.format(ATTR_TEMPLATE, "dir", this.dir));
```

... TAK, wreszcie wyczerpaliśmy pulę niezbędnych metod ustawiających wartości atrybutów.

Na tym jednak nie koniec. Klasa `SelectTagHandler` wymaga jeszcze więcej kodu. Metoda `doTag()` musi jeszcze zapisać w strumieniu odpowiedzi każdy ze standardowych atrybutów znacznika `<select>` języka HTML. Obsługa tych dodatkowych 17 atrybutów wymaga dalszych 34 wierszy kodu, czyli jeszcze przynajmniej półtorej strony. To całkiem sporo...



Masz rację. Przedstawione rozwiązanie rzeczywiście jest żałosne. Co więcej, wymaga pisania i utrzymywania ogromnej ilości kodu. A jeśli w dodatku zdecydujemy się opracować cały pakiet znaczników niestandardowych rozszerzających możliwości *innych* znaczników HTML-a?

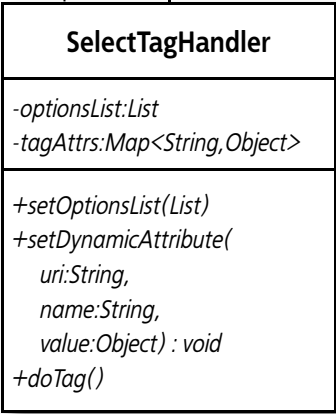
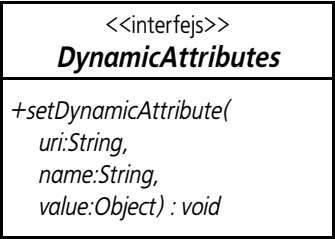
Klasy obsługi znacznika muszą implementować po jednej metodzie ustawiającej dla każdego atrybutu znacznika zadeklarowanego w deskrytorze TLD. Z drugiej strony, wspomniane metody nie podejmują żadnych interesujących działań. Wartości tych atrybutów są po prostu przekazywane do danych wyjściowych generowanych na potrzeby znacznika `<select>` języka HTML.

Można by zastosować pewną regułę projektową: „Zamykaj hermetycznie to, co się zmienia”.<sup>\*</sup> W tym przypadku *tym, co się zmienia*, jest zbiór opcjonalnych atrybutów znacznika języka HTML na poziomie klasy obsługi. Jednym z możliwych rozwiązań jest umieszczenie wszystkich tych atrybutów w tablicy mieszającej. Można w ten sposób wyodrębnić strukturę atrybutów w ramach obiektu znacznika, co jednak nie rozwiązuje problemu metod ustawiających. Ich wyeliminowanie jest niemożliwe, chyba że zmusimy silnik JSP do ustawiania atrybutów znacznika z użyciem uniwersalnego interfejsu.

<sup>\*</sup> Przytoczoną regułę projektową omówiono na stronie 250 książki *Head First Object-Oriented Analysis and Design*.<sup>1</sup>

Oczywiście nie mogliśmy sobie odmówić przyjemności bezwstydnego zareklamowania innej książki z cyklu *Head First*.

<sup>1</sup> Polskie wydanie: *Head First Object-Oriented Analysis and Design*. Edycja polska, Helion, 2008 — przyp. tłum.



Atrybuty dynamiczne najprawdopodobniej będziesz składował w mapie mieszającej.

Jedna metoda ustawiająca jest stosowana dla wszystkich atrybutów dynamicznych. Parametr *name* reprezentuje nazwę ustawianego atrybutu. Parametr *value* reprezentuje jego wartość. Parametr *uri* reprezentuje przestrzeń nazw XML-a definiującą dany atrybut. W normalnych okolicznościach można ten parametr po prostu zignorować.



**Relax**

Na egzaminie nie będzie sprawdzana znajomość sygnatury tej metody ani tym bardziej przeznaczenie parametru *uri*.

Nikt u licha nie wie, po co ten parametr.

## Kod klasy obsługi korzystającej z interfejsu `DynamicAttributes`

Sprawdźmy teraz, jak interfejs `DynamicAttributes` sprawdza się w praktycznych zastosowaniach. Klasa obsługi naszego znacznika niestandardowego musi oczywiście implementować interfejs `DynamicAttributes` z JSP API. Sam interfejs `DynamicAttributes` wymaga z kolei zaimplementowania metody `setDynamicAttribute()` odpowiedzialnej za składowanie par nazwa-wartość (reprezentujących atrybuty) w mapie mieszającej, czyli wprost idealnej strukturze danych dla tego rodzaju informacji.

```
package com.example.taglib;

import java.io.IOException;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import javax.servlet.jsp.JspException;
import javax.servlet.jsp.JspWriter;
import javax.servlet.jsp.PageContext;
import javax.servlet.jsp.tagext.DynamicAttributes;
import javax.servlet.jsp.tagext.SimpleTagSupport;

/**
 * Trzecia wersja znacznika select języka HTML wykorzystuje mechanizm atrybutów
 * dynamicznych JSP do składowania wszystkich przekazywanych dalej atrybutów
 * HTML-a w jednej strukturze mapy mieszającej.
 */
public class SelectTagHandler
 extends SimpleTagSupport
 implements DynamicAttributes {

 // ustaw atrybut optionsList
 public void setOptionsList(List value) {
 this.optionsList = value;
 }
 private List optionsList;

 // ustaw atrybut name
 public void setName(String value) {
 this.name = value;
 }
 private String name;

 // ustaw wszystkie pozostałe (dynamiczne) atrybuty
 public void setDynamicAttribute(String uri, String name, Object value) {
 tagAttrs.put(name, value);
 }
 private Map<String, Object> tagAttrs = new HashMap<String, Object>();
```

*Klasa obsługi naszego znacznika musi implementować interfejs `DynamicAttributes`.*

*Działanie metody ustawiającej z reguły ogranicza się do umieszczania poszczególnych par nazwa-wartość w mapie mieszającej.*

### Pozostały kod klasy obsługi znacznika

Pozostaje nam już tylko wprowadzenie pewnych modyfikacji w metodzie `doTag()`. Jediną istotną zmianą jest fakt składowania atrybutów standardowego znacznika `<select>` języka HTML w strukturze mapy mieszającej. Metoda `doTag()` musi iteracyjnie przeszukać elementy tej mapy i na tej podstawie wygenerować atrybuty HTML-a przekazywane następnie do strumienia odpowiedzi. Pozostałe elementy metody `doTag()` pozostają niezmienione.

To proste, prawda?

```
// wygeneruj znaczniki <select> i <option>
public void doTag() throws JspException, IOException {
 PageContext pageContext = (PageContext) getJspContext();
 JspWriter out = pageContext.getOut();

 // Początek znacznika <select> HTML-a z niezbędnymi atrybutami tego języka
 out.print("<select ");

 // dodaj atrybuty wymagane
 out.print(String.format(ATTR_TEMPLATE, "name", this.name));

 // dodaj atrybuty dynamiczne
 for (String attrName : tagAttrs.keySet()) {
 String attrDefinition
 = String.format(ATTR_TEMPLATE,
 attrName, tagAttrs.get(attrName));
 out.print(attrDefinition);
 }
 out.print('>');

 // Wygeneruj znaczniki <option> na podstawie struktury optionsList
 for (Object option : this.optionsList) {
 String optionTag = String.format(OPTION_TEMPLATE, option.toString());
 out.println(optionTag);
 }

 // Koniec znacznika </select> języka HTML
 out.println(" </select>");
} // KONIEC metody doTag()

private static final String ATTR_TEMPLATE = "%s='%s' ";
private static final String OPTION_TEMPLATE
 = " <option value='%1$s'> %1$s </option>";

} // KONIEC klasy SelectTagHandler
```

Uzyskujemy zbiór atrybutów na podstawie kluczy naszej mapy. Każdy klucz reprezentuje nazwę jednego z atrybutów dynamicznych.

Wartość danego atrybutu jest składowana w mapie. Metoda `get()` uzyskuje wartość właściwą dla klucza (czyli nazwy danego atrybutu).

## No dobrze, pozostało nam jeszcze trochę pracy konfiguracyjnej w deskrypcorze TLD

Ha! Nie sądziłeś chyba, że nasze nowe rozwiązanie ogranicza się tylko do kodu Javy, prawda? Musimy jeszcze uwzględnić element konfiguracji. Szczęśliwie mówimy o rozwiązaniach precyzyjnie opisanych w specyfikacji JSP, zatem niezbędne zmiany nie powinny nam sprawić kłopotu. Dodatkowy element nazwano `<dynamic-attributes>`:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
 PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
 "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_2.dtd">
```

```
<taglib>
```

```
 <tlib-version>1.2</tlib-version>
 <jsp-version>1.2</jsp-version>
 <short-name>Biblioteka znaczników formularza</short-name>
 <uri>http://example.com/tags/forms</uri>
 <description>
```

Przykład biblioteki znaczników stosowanych w miejsce znaczników formularza HTML.

```
</description>
```

```
<tag>
 <name>select</name>
 <tag-class>com.example.taglib.SelectTagHandler</tag-class>
 <body-content>empty</body-content>
 <description>
```

Ten znacznik konstruuje znacznik `<select>` stosowany w formularzach języka HTML. Znacznik dodatkowo generuje znaczniki `<option>` na podstawie zbioru elementów składowanych w liście przekazanej za pośrednictwem atrybutu `optionsList`.

```
</description>
 <attribute>
 <name>optionsList</name>
 <type>java.util.List</type>
 <required>true</required>
 <rtexprvalue>true</rtexprvalue>
 </attribute>
 <attribute>
 <name>name</name>
 <required>true</required>
 </attribute>
```

Nadal musimy zadeklarować wszystkie atrybuty wymagane. Dla każdego z tych atrybutów musi istnieć zdefiniowana wprost metoda ustawiająca w ramach klasy obsługi.

```
 <dynamic-attributes>true</dynamic-attributes>
```

```
</tag>
```

```
</taglib>
```

Ten element w zupełności wystarczy do zadeklarowania możliwości otrzymywania przez dany znacznik dowolnej liczby atrybutów dynamicznych.

### Nie ma niemądrych pytań

**P.** Stosowaliśmy model znaczników prostych. Czy przedstawione rozwiązanie można by zastosować także w modelu klasycznym?

**U.** Tak, interfejs `DynamicAttributes` może być implementowany przez znaczniki klasyczne dokładnie tak samo jak przez znaczniki proste. Co ciekawe, nawet konfiguracja w pliku TLD pozostaje niezmieniona.

**P.** Czy atrybutem dynamicznym można przypisywać wyrażenia przetwarzane w czasie wykonywania, np. wyrażenia języka EL lub konstrukcje `<%= %>`?

**U.** Oczywiście. Każdy atrybut dynamiczny domyślnie może obsługiwać wyrażenia języka EL i wyrażenia JSP w roli wartości atrybutów. Być może zauważyłeś, że nawet parametr `value` metody `setDynamicAttribute()` jest obiektem klasy `Object`, nie łańcuchem! To zaś oznacza, że w roli wartości można stosować dowolne obiekty Javy.

**P.** Co będzie, jeśli stanę przed koniecznością „przetworzenia” danych przekazanych za pośrednictwem atrybutu dynamicznego?

**U.** Zawsze możesz sprawdzić nazwę parametru i na tej podstawie zdecydować o ewentualnym przetwarzaniu i transformacji wartości odpowiedniego atrybutu. Jeśli jednak planujesz implementację tego rodzaju funkcjonalności, prawdopodobnie powinieneś wprost zdefiniować odpowiedni atrybut i podejmować stosowne działania w metodzie ustawiającej.

**P.** Co będzie, jeśli użytkownik mojego znacznika niestandardowego wpisze niepoprawną nazwę atrybutu?

**U.** To jedno z pytań za 100 punktów. Ponieważ nazwy atrybutów nie są deklarowane wprost w deskrytorze TLD, silnik JSP przekazuje je wszystkie klasie obsługi znacznika za pośrednictwem metody `setDynamicAttribute()`. Takie podejście powoduje, że autor strony JSP może popełnić błąd w pisowni standardowego atrybutu HTML-a i nigdy się o tym nie dowiedzieć — a przynajmniej do najbliższej nieudanej próby wywołania znacznika przez przeglądarkę. Okazuje się więc, że z pozoru karkołomne pierwsze rozwiązanie zaproponowane przez Gary’ego (z deklarowanymi wprost atrybutami oraz niezliczonymi metodami ustawiającymi i deklaracjami TLD) miało pewne zalety. Potrafisz wskazać inne zalety rozwiązania Gary’ego w porównaniu z rozwiązaniem Kima?



## WYTEŻ UMYSŁ

**Rozwiązanie Gary’ego polegało na jawnej obsłudze wszystkich atrybutów. W rozwiązaniu Kima większość atrybutów przekształcono w atrybuty dynamiczne. Oba podejścia mają swoje zalety i wady. Czy istnieje trzecia droga?**

## A co z plikami znaczników?

Także pliki znaczników mogą zawierać atrybuty dynamiczne. Chociaż w tym przypadku kluczowe elementy naszego mechanizmu pozostają niemal identyczne, warto pamiętać, że silnik JSP udostępnia plikom znaczników dodatkowy obiekt **Mapy**. Oznacza to, że możemy przeglądać i iteracyjnie przeszukiwać elementy mapy par atrybut-wartość za pomocą znacznika `forEach` biblioteki JSTL.

```
<%@ tag body-content='empty' dynamic-attributes='tagAttrs' %>
```

Wartością atrybutu `dynamic-attributes` jest zmienna zasięgu strony reprezentująca mapę mieszającą.

```
<%@ attribute name='optionsList' type='java.util.List'
 required='true' rtexprvalue='true' %>
```

```
<%@ attribute name='name' required='true' %>
```

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
```

```
<select name='${name}'
 <c:forEach var="attrEntry" items="${tagAttrs}">
 ${attrEntry.key}='${attrEntry.value}'
 </c:forEach>
>
```

Wykorzystujemy znacznik niestandardowy `forEach` biblioteki JSTL do iteracyjnego przetworzenia wszystkich elementów mapy atrybutów dynamicznych. Pamiętaj, że funkcję klucza w każdym z elementów tej mapy pełni nazwa atrybutu, a wartością tego elementu jest wartość reprezentowanego atrybutu.

```
<c:forEach var="option" items="${optionsList}">
 <option value='${option}'> ${option} </option>
</c:forEach>
```

```
</select>
```

### KLUCZOWE ZAGADNIENIA



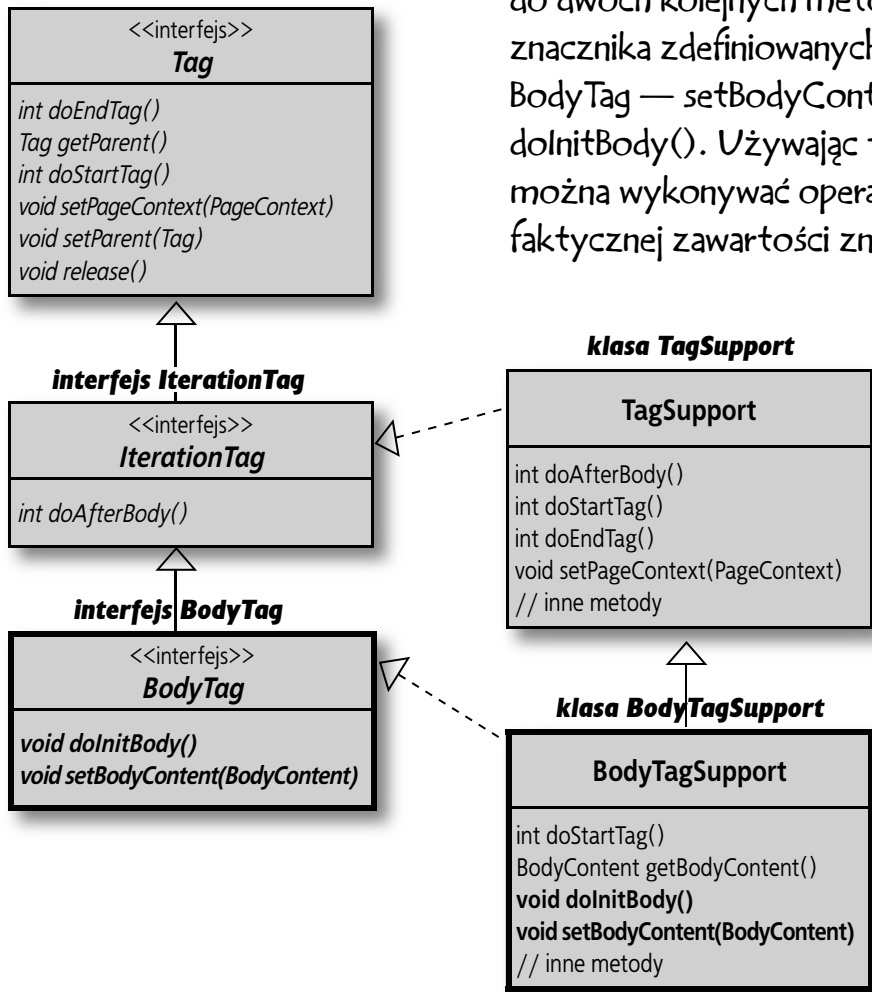
- Interfejs `DynamicAttributes` umożliwia klasie obsługi znacznika operowanie na dowolnej liczbie atrybutów.
- Deklaracja znacznika w deskrytorze TLD musi zawierać element `<dynamic-attributes>`.
- Dla atrybutów znacznika deklarowanych wprost muszą istnieć metody ustawiające.
- W typowych warunkach korzystamy w metodzie `setDynamicAttribute()` z mapy mieszającej do składowania po jednej parze nazwa-wartość dla każdego z atrybutów dynamicznych.
- Atrybuty dynamiczne można stosować także w plikach znaczników.
- W takim przypadku należy pamiętać o konieczności użycia atrybutu `dynamic-attributes` w dyrektywie tag.
- Wartość atrybutu `dynamic-attributes` reprezentuje mapę mieszającą atrybutów dynamicznych.
- Do iteracyjnego przeszukiwania tej mapy z reguły wykorzystuje się akcję niestandardową `forEach` biblioteki JSTL.

# A co zrobić, jeśli NAPRAWDĘ potrzebujemy dostępu do zawartości znacznika?

Sam zapewne dojdiesz do wniosku, że w większości przypadków metody interfejsów Tag oraz IterationTag udostępniane w klasie TagSupport są całkowicie wystarczające. Korzystając z trzech podstawowych metod — doStartTag(), doAfterBody() oraz doEndTag() — można zrobić niemal wszystko.

Z jednym wyjątkiem — nie dysponujemy dostępem do zawartości znacznika. Jeśli potrzebujemy dostępu do faktycznej zawartości znacznika, tak aby, na przykład, zastosować ją w wyrażeniu lub w pewien sposób zmodyfikować, to musimy stworzyć klasę dziedziczącą nie po klasie TagSupport, lecz BodyTagSupport; dzięki temu uzyskamy dostęp do metod interfejsu BodyTag.

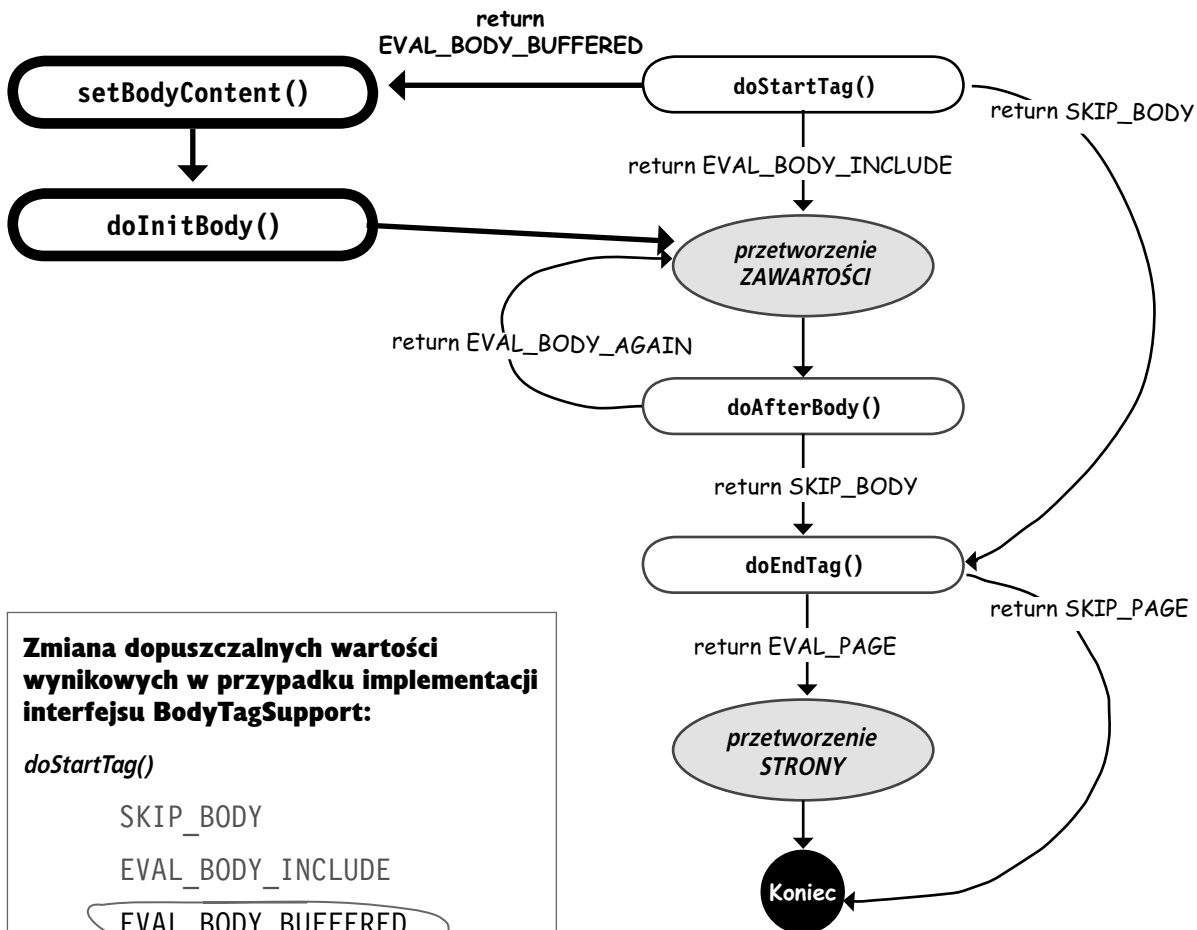
Tworząc klasę dziedziczącą po klasie BodyTagSupport, zyskujemy dostęp do dwóch kolejnych metod cyklu życia znacznika zdefiniowanych w interfejsie BodyTag — setBodyContent() oraz doInitBody(). Używając tych metod, można wykonywać operacje na faktycznej zawartości znacznika.



## Interfejs BodyTag udostępnia dwie kolejne metody

Implementując interfejs BodyTag (przez stworzenie klasy dziedziczącej po BodyTagSupport), zyskujemy możliwość korzystania z dwóch dodatkowych metod cyklu życia znacznika — `setBodyContent()` oraz `doInitBody()`. Możemy też korzystać z dodatkowej wartości wynikowej zwracanej przez metodę `doStartTag()` i reprezentowanej przez stałą `EVAL_BODY_BUFFERED`. Oznacza to, że w tym przypadku istnieją już trzy potencjalne wartości, które może zwracać metoda `doStartTag()` (w odróżnieniu od dwóch, którymi dysponowaliśmy w przypadku implementacji interfejsu TagSupport).

### Cykl życia znacznika implementującego interfejs BodyTag (bezpośrednio bądź poprzez dziedziczenie po klasie BodyTagSupport)



#### Zmiana dopuszczalnych wartości wynikowych w przypadku implementacji interfejsu BodyTagSupport:

`doStartTag()`

`SKIP_BODY`

`EVAL_BODY_INCLUDE`

`EVAL_BODY_BUFFERED`

Nowa wartość wynikowa, która w przypadku implementacji interfejsu BodyTagSupport jest wartością domyślną (w odróżnieniu od wartości SKIP\_BODY, która była wartością domyślną w przypadku implementacji interfejsu TagSupport).

# Dzięki interfejsowi BodyTag można buforować zawartość znacznika

Argument typu `BodyContent` przekazywany w wywołaniu metody `setBodyContent()` jest w istocie obiektem typu `java.io.Writer` (choć trzeba przyznać, że u programisty przywiązanego do idei programowania obiektowego takie rozwiązanie może budzić uzasadnione zaniepokojenie). Niemniej jednak, dzięki takiemu rozwiązaniu można przetworzyć zawartość znacznika, na przykład przekazując ją do innego strumienia wyjściowego lub samodzielnie ją odczytać i zmodyfikować.

**❓ Co się stanie, jeśli zwrócę wartość `EVAL_BODY_BUFFERED` w przypadku, gdy znacznik umieszczony w kodzie strony będzie pusty?**

**U.** Jeśli znacznik, który spowodował wykonywanie metod obiektu obsługi, jest pusty, to metody `setBodyContent()` oraz `doInitBody()` nie zostaną wywołane! Termin „pusty” oznacza w tym przypadku, że znacznik umieszczony w kodzie został zapisany jako znacznik pusty — `<moje:znacznik />`, bądź też, iż pomiędzy znacznikiem otwierającym i zamykającym nie została podana żadna zawartość — `<moje:znacznik></moje:znacznik>`.

Kontener wie, że tym razem znacznik nie ma zawartości, i po prostu przeskakuje do metody `doEndTag()`, więc zazwyczaj nie stanowi to żadnego problemu.

**Chyba że w deskrytorze TLD zadeklarowano, że znacznik ma mieć pustą zawartość.** Jeśli w TLD został umieszczony element `<body-content>empty</body-content>`, to nie masz wyjścia i NIE wolno Ci zwracać z metody `doStartTag()` wartości `EVAL_BODY_BUFFERED` ani `EVAL_BODY_INCLUDE`.

**❓ Co się dzieje z atrybutami znacznika w przypadku tworzenia klasy obsługi w modelu klasycznym? Czy są obsługiwane w taki sam sposób jak w przypadku stosowania modelu prostego?**

**U.** Owszem. W obu tych przypadkach na diagramach sekwencji obsługi znacznika znajduje się operacja polegająca na wywołaniu odpowiedniej metody `set` dla każdego z atrybutów podanych w znaczniku. W przypadku modelu prostego operacja ta jest wykonywana przed wywołaniem metody `doTag()`, natomiast w przypadku modelu klasycznego — przed wywołaniem metody `doStartTag()`. Innymi słowy, w obu modelach implementacji znaczników atrybuty są obsługiwane w identyczny sposób. Dotyczy to także sposobu deklaracji atrybutów w deskrytorze TLD.



Relax

Nie musisz znać wszelkich szczegółów związanych ze stosowaniem klasy `BodyTagSupport`.

Przystępując do egzaminu, musisz znać cykl życia klasy `BodyTagSupport` oraz różnice pomiędzy nią a klasą `TagSupport` (ta wiedza może Ci się przydać także w normalnej pracy). Musisz na przykład wiedzieć, że jeśli Twoja klasa NIE dziedziczy po `BodyTagSupport` ani NIE implementuje interfejsu `BodyTag`, metoda `doStartTag()` NIE może zwracać wartości `EVAL_BODY_BUFFERED`. Powinieneś także znać dwie nowe metody interfejsu `BodyTag`. Ale na tym koniec.



Oglądaj to!

**Jeśli w deskrytorze znacznik został zadeklarowany jako pusty, to metoda `doStartTag()` musi zwracać wartość `SKIP_BODY`!**

Choć ten fakt może być oczywisty, jednak oznacza on, iż należy zwrócić baczną uwagę, aby klasa obsługi znacznika oraz jego deskryptor były ze sobą zgodne. A zatem, jeśli w deskrytorze pojawi się element `<body-content>empty</body-content>`, to implementacja interfejsu `BodyTag` (lub dziedziczenie po klasie `BodyTagSupport`) NIE ma najmniejszego sensu. Oznacza to jednocześnie, że nie ma sensu implementowanie interfejsu `IterationTag`, choć jest on automatycznie implementowany w przypadku dziedziczenia po klasie `TagSupport`. Chodzi o to, iż jeśli w deskrytorze znacznik został zadeklarowany jako pusty, to metoda `doStartTag()` musi zwrócić wartość `SKIP_BODY`, nawet jeśli implementujesz interfejs `IterationTag` lub `BodyTag`.



Metody cyklu życia znaczników klasycznych

Uzupełnij poniższą tabelę. We wcześniejszej części rozdziału podaliśmy informacje niezbędne do poprawnego podania niemal wszystkich odpowiedzi, jednak w kilku przypadkach będziesz musiał zgadywać. (Nie odwracaj kartki!)

	BodyTagSupport	TagSupport
<b>Metoda doStartTag()</b> <i>Możliwe wartości wynikowe.</i>		
<i>Domyślna wartość zwracana przez implementację tej metody w klasie.</i>		
Liczba możliwych wywołań metody (w ramach obsługi jednego znacznika w kodzie strony JSP).		
<b>Metoda doAfterBody()</b> <i>Możliwe wartości wynikowe.</i>		
<i>Domyślna wartość zwracana przez implementację tej metody w klasie.</i>		
Liczba możliwych wywołań metody (w ramach obsługi jednego znacznika w kodzie strony JSP).		
<b>Metoda doEndTag()</b> <i>Możliwe wartości wynikowe.</i>		
<i>Domyślna wartość zwracana przez implementację tej metody w klasie.</i>		
Liczba możliwych wywołań metody (w ramach obsługi jednego znacznika w kodzie strony JSP).		
<b>Metody doInitBody() oraz setBodyContent()</b>  Czynniki decydujące o wywołaniach tych metod oraz liczba możliwych wywołań podczas obsługi jednego znacznika.		



Wartości wynikowe metod obsługi znaczników klasycznych

Przystępując do egzaminu, musisz być w stanie odpowiedzieć na wszystkie poniższe pytania!

ROZWIĄZANIE  
ĆWICZENIA

	BodyTagSupport	TagSupport
Metoda <b>doStartTag()</b> <i>Możliwe wartości wynikowe.</i>	SKIP_BODY EVAL_BODY_INCLUDE EVAL_BODY_BUFFERED	SKIP_BODY EVAL_BODY_INCLUDE
<i>Domyślna wartość zwracana przez implementację tej metody w klasie.</i>	EVAL_BODY_BUFFERED	SKIP_BODY
Liczba możliwych wywołań metody (w ramach obsługi jednego znacznika w kodzie strony JSP).	Dokładnie raz	Dokładnie raz
Metoda <b>doAfterBody()</b> <i>Możliwe wartości wynikowe.</i>	SKIP_BODY EVAL_BODY_AGAIN	SKIP_BODY EVAL_BODY_AGAIN
<i>Domyślna wartość zwracana przez implementację tej metody w klasie.</i>	SKIP_BODY	SKIP_BODY
Liczba możliwych wywołań metody (w ramach obsługi jednego znacznika w kodzie strony JSP).	Dowolną liczbę razy (w tym w ogóle)	Dowolną liczbę razy (w tym w ogóle)
Metoda <b>doEndTag()</b> <i>Możliwe wartości wynikowe.</i>	SKIP_PAGE EVAL_PAGE	SKIP_PAGE EVAL_PAGE
<i>Domyślna wartość zwracana przez implementację tej metody w klasie.</i>	EVAL_PAGE	EVAL_PAGE
Liczba możliwych wywołań metody (w ramach obsługi jednego znacznika w kodzie strony JSP).	Dokładnie raz	Dokładnie raz
Metody <b>doInitBody()</b> oraz <b>setBodyContent()</b>  Czynniki decydujące o wywołaniach tych metod oraz liczba możliwych wywołań podczas obsługi jednego znacznika.	Dokładnie raz i TYLKO wtedy, gdy metoda <b>doStartTag()</b> zwróci wartość <b>EVAL_BODY_BUFFERED</b>	NIGDY!

## A co, jeśli znaczniki współpracują ze sobą?

Wyobraźmy sobie pewien scenariusz... Dysponujemy znacznikiem `<moje:Menu>`, który tworzy pasek nawigacyjny. W tym znaczniku musimy jednak podać opcje menu. A zatem wewnątrz znacznika `<moje:Menu>` umieszczamy znaczniki `<moje:OpcjaMenu>`, przy czym znacznik `<moje:Menu>` musi się jakoś „dowiedzieć” (nie wiemy jeszcze, jak) o umieszczonych w nim znacznikach i używa ich do skonstruowania paska nawigacyjnego.

```
<moje:Menu>
```

```
 <moje:OpcjaMenu wartoscOpcji="Psy" />
```

```
 <moje:OpcjaMenu wartoscOpcji="Koty" />
```

```
 <moje:OpcjaMenu wartoscOpcji="Konie" />
```

```
</moje:Menu>
```

← Znacznik Menu będzie potrzebował dostępu do wartości atrybutów znaczników umieszczonych wewnątrz niego.

A zatem kluczowe pytanie, na jakie musimy znaleźć odpowiedź, brzmi: „w jaki sposób znaczniki komunikują się między sobą?”. Innymi słowy, w jaki sposób znacznik Menu (zewnątrzny) może odczytać wartości atrybutów znaczników OpcjaMenu (wewnętrznych)?

Znaczniki wewnętrzne są powszechnie wykorzystywane między innymi w bibliotece JSTL — doskonałym przykładem może być znacznik `<c:choose>` ze swoimi zagnieżdżonymi znacznikami `<c:when>` oraz `<c:otherwise>`. Może się zdarzyć, iż takich znaczników „współpracujących” (jak są one określane w specyfikacji, ang.: *cooperating tags*), będziesz także musiał używać we własnej pracy.

Na szczęście istnieje mechanizm pozwalający uzyskiwać dostęp do znaczników zewnętrznych oraz wewnętrznych niezależnie od głębokości zagnieżdżenia. Oznacza to, iż informacje ze znaczników zagnieżdżonych można pobierać nie tylko w znaczniku, bezpośrednio wewnątrz którego są one umieszczone, lecz w dowolnych znacznikach, które je zawierają.

### — Zaostrz ołówek

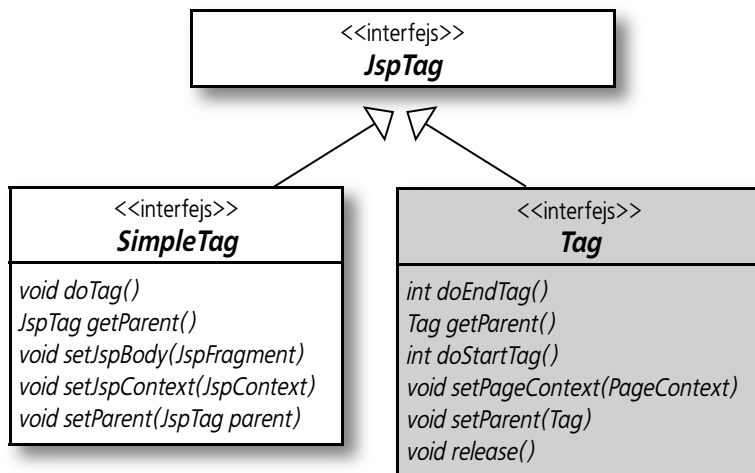


Przjrzyj się metodom zdefiniowanym w interfejsie `Tag`, przejrzyj przykładowe kody podane we wcześniejszej części rozdziału i spróbuj się zastanowić, w jaki sposób współpracujące ze sobą znaczniki mogą pobierać informacje na swój temat.

<<interfejs>> <b>Tag</b>
<pre>int doEndTag() Tag getParent() int doStartTag() void setPageContext(PageContext) void setParent(Tag) void release()</pre>

# Znacznik może odwołać się do znacznika zewnętrznego

Zarówno w interfejsie `SimpleTag`, jak i `Tag` została zdefiniowana metoda `getParent()`. Metoda `getParent()` interfejsu `Tag` zwraca obiekt typu `Tag`, natomiast metoda `getParent()` interfejsu `SimpleTag` — obiekt typu `JspTag`. Już za chwilę dowiesz się, jakie implikacje pociągają za sobą te typy obiektów zwracanych przez metody `getParent()`.



## Znacznik zagnieżdżony może się odwołać do swojego znacznika zewnętrznego (rodzica)

```
<moje:ZnacznikZewnetrzny>
 <moje:ZnacznikWewnetrzny />
</moje:ZnacznikZewnetrzny>
```

W tej relacji „ZnacznikZewnetrzny” jest znacznikiem macierzystym zawierającego „ZnacznikaWewnetrznego”.

## Uzyskiwanie referencji do znacznika zewnętrznego w modelu klasycznym

```
public int doStartTag() throws JspException {
 ZnacznikZewnetrzny zz = (ZnacznikZewnetrzny) getParent();
 // używamy znacznika zewnętrznego do wykonania jakichś operacji
 return EVAL_BODY_INCLUDE;
}
```

Nie zapomnij o odpowiednim rzutowaniu typów!

## Uzyskiwanie referencji do znacznika zewnętrznego w modelu prostym

```
public void doTag() throws JspException, IOException {
 ZnacznikZewnetrzny zz = (ZnacznikZewnetrzny) getParent();
 // używamy znacznika zewnętrznego do wykonania jakichś operacji
}
```

Ten wiersz jest identyczny jak w przypadku klasy obsługi znacznika tworzonej w modelu klasycznym.

Także i w tym przypadku nie można zapomnieć o rzutowaniu typów.

## Określanie poziomu zagnieżdżenia...

Wywołując metodę `getParent()` obiektu zwróconego przez poprzednie wywołanie tej metody, można przechodzić ku górze hierarchii znaczników. Jest to możliwe, ponieważ metoda `getParent()` zwraca albo obiekt reprezentujący inny znacznik (którego można użyć do kolejnego wywołania metody `getParent()`), albo wartość `null`.

### W kodzie JSP

```
<moje:ZnacznikNaPoziomie1>
 <moje:ZnacznikNaPoziomie2>
 <moje:ZnacznikNaPoziomie3 />
 </moje:ZnacznikNaPoziomie2>
</moje:ZnacznikNaPoziomie1>
```

### W klasie obsługi znacznika tworzonej w modelu klasycznym

```
package tmp;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;

public class ZnacznikZagniezdzony extends TagSupport {
 private int poziomZagniezdzenia;

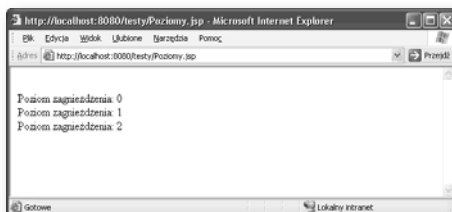
 public int doStartTag() throws JspException {
 poziomZagniezdzenia = 0;
 Tag zewn = getParent();
 while (zewn != null) {
 zewn = zewn.getParent();
 poziomZagniezdzenia++;
 }
 try {
 pageContext.getOut().println("
Poziom zagnieżdżenia: " + poziomZagniezdzenia);
 } catch (IOException ex) {
 throw new JspException("IOException - " + ex.toString());
 }
 return EVAL_BODY_INCLUDE;
 }
}
```

Wywołanie odziedziczonej metody `getParent()`.

Jeśli została zwrócona wartość `null`, oznacza to, że jesteśmy na najwyższym poziomie i znacznik nie ma żadnego znacznika zewnętrznego.

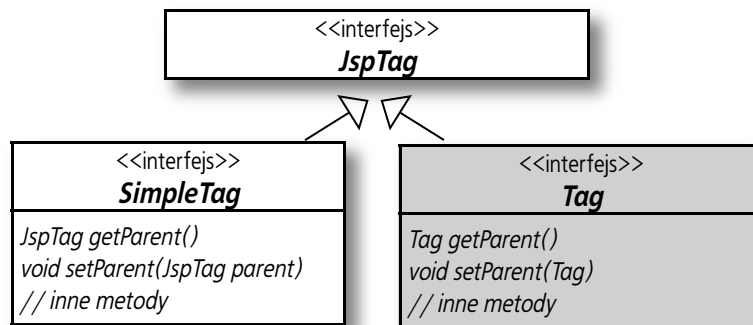
Jednak w przeciwnym razie pobieramy znacznik zewnętrzny pobranego znacznika i inkrementujemy zmienną określającą poziom zagnieżdżenia.

### Wyniki



## Znaczniki proste można umieszczać w znacznikach klasycznych

Takie rozwiązanie nie przysparza najmniejszych problemów, ponieważ metoda `getParent()` zdefiniowana w interfejsie `SimpleTag` zwraca obiekt typu `JspTag`, który obecnie jest interfejsem bazowym dla interfejsu implementowanego przez klasy obsługi znaczników tworzonych zarówno w modelu prostym, jak i klasycznym. W praktyce znaczniki *klasyczne* także mogą być umieszczane wewnątrz znaczników *prostych*, jednak rozwiązanie to wymaga zastosowania pewnej sztuczki, gdyż typu `SimpleTag` nie można rzutować na typ `Tag` zwracany przez metodę `getParent()` interfejsu `Tag`. Nie będziemy tu szczegółowo opisywać sposobu odwoływania się do znacznika prostego z poziomu umieszczonego wewnątrz niego znacznika klasycznego\*, jednak na potrzeby egzaminu (i praktycznych potrzeb pojawiających się przy tworzeniu aplikacji internetowych) wystarczy abyś wiedział, że metody `getParent()` można używać w celu odwołania się do znacznika klasycznego z poziomu umieszczonego wewnątrz niego innego znacznika klasycznego lub do odwołania się do znacznika dowolnego typu z poziomu umieszczonego wewnątrz niego znacznika prostego.



Dzięki zastosowaniu metody `getParent()` znaczniki klasyczne mogą uzyskiwać dostęp do macierzystych (zewnętrznych) znaczników klasycznych, natomiast znaczniki proste mogą uzyskiwać dostęp zarówno do macierzystych znaczników prostych, jak i macierzystych znaczników klasycznych.

### W kodzie JSP

```

<moje:KlasycznyZnacznikZewnetrzny nazwa="KlasycznyZnacznikZewnetrzny">
 <moje:ProstyZnacznikWewnetrzny />
</moje:KlasycznyZnacznikZewnetrzny>

```

Co zrobić, jeśli znacznik wewnętrzny (`ProstyZnacznikWewnetrzny`) chce uzyskać dostęp do atrybutu nazwa znacznika zewnętrznego?

### W klasie obsługi znacznika `ProstyZnacznikWewnetrzny`

```

public void doTag() throws JspException, IOException {
 MojZnacznikKlasyczny zewn = (MojZnacznikKlasyczny) getParent();
 getJspContext().getOut().print("Wartość atrybutu znacznika zewnętrznego: " + zewn.getNazwa());
}

```

Znacznik prosty może uzyskać dostęp do znacznika zewnętrznego obsługiwane w modelu klasycznym.

### W klasie obsługi znacznika `KlasycznyZnacznikZewnetrzny`

```

public class MojZnacznikKlasyczny extends TagSupport {
 private String nazwa;

 public void setNazwa(String nazwa) {
 this.nazwa = nazwa;
 }

 public String getNazwa() {
 return nazwa;
 }

 public int doStartTag() throws JspException {
 return EVAL_BODY_INCLUDE;
 }
}

```

Napisz metodę `get` dla atrybutu, tak by znacznik wewnętrzny mógł odczytać jego wartość.

Po pobraniu obiektu znacznika zewnętrznego można wywoływać jego metody, w końcu jest to zwyczajny obiekt Javy; a zatem można wywołać metodę w celu pobrania wartości atrybutu tego znacznika.

\* Jeśli naprawdę Cię to interesuje, zajrzyj do dokumentacji klasy `TagAdapter` w specyfikacji interfejsu API J2EE 1.4.

Jeśli ta metoda zwróci wartość `SKIP_BODY`, to znacznik wewnętrzny nigdy nie zostanie przetworzony!

## Można przemieszczać się w górze, lecz nie w dół...

Istnieje metoda `getParent()`, ale nie ma metody `getChild()`. Jednak w przykładzie, który przedstawiliśmy wcześniej, to zewnętrzny znacznik — `<moje:Menu>` — potrzebował informacji pobieranych z zagnieżdżonych znacznikach `<moje:OpcjaMenu>`. Co możemy zrobić w takiej sytuacji? W jaki sposób znacznik zewnętrzny może pobrać informacje o znaczniku wewnętrznym, skoro znacznik wewnętrzny może pobrać referencję do znacznika zewnętrznego, a pobranie referencji do znacznika wewnętrznego nie jest możliwe?



### — Zaostrz ołówek —



W jaki sposób znacznik zewnętrzny może odczytać wartości atrybutów jego znaczników wewnętrznych? Opisz, jak zaimplementowałbyś własny mechanizm współdziałania znaczników `Menu` oraz `OpcjaMenu`.

# Przekazywanie informacji ze znacznika wewnętrznego do zewnętrznego

Istnieją dwa podstawowe sposoby pozwalające na wzajemne współdziałanie znaczników:

1. Znacznik wewnętrzny potrzebuje informacji (na przykład wartości atrybutu) ze znacznika zewnętrznego.
2. Znacznik zewnętrzny potrzebuje informacji z każdego ze znaczników umieszczonych w jego wnętrzu.

We wcześniejszej części rozdziału pokazaliśmy już rozwiązanie pierwszego z powyższych scenariuszy — znacznik wewnętrzny może pobrać referencję do znacznika zewnętrznego przy użyciu metody `getParent()`, a następnie wywoływać metody zwrócone przez nią obiektu. Co się jednak dzieje, kiedy to znacznik zewnętrzny (macierzysty) potrzebuje informacji o swoich znacznikach potomnych? Otóż w takim przypadku musimy postąpić identycznie — czyli, jeśli znacznik zewnętrzny potrzebuje informacji o znaczniku wewnętrznym, to znacznik wewnętrzny musi je mu dostarczyć!

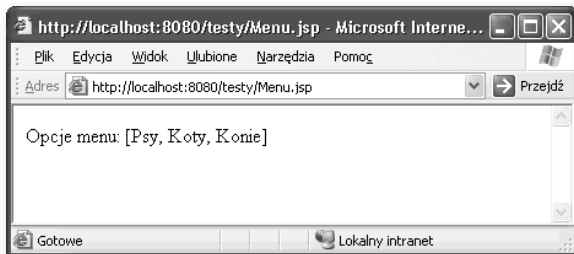
Ponieważ nie istnieje żaden automatyczny sposób uzyskiwania przez znacznik macierzysty informacji o jego znacznikach wewnętrznych, musimy zastosować te same metody co w przypadku pozyskiwania informacji o znaczniku zewnętrznym z poziomu znacznika wewnętrznego. Trzeba pobrać referencję do znacznika zewnętrznego i wywoływać jego metody. Jednak w tym przypadku zamiast metod `get` (zwracających wartości) zastosujemy ich odpowiedniki w formie metod ustawiających lub dodających (`set` lub `add`).

## W kodzie JSP

```
<%@ tablib prefix="moje" uri="MojeZnacznikiKlasyczne" %>
<html></body>
<moje:Menu>
 <moje:OpcjaMenu wartoscOpcji="Psy" />
 <moje:OpcjeMenu wartoscOpcji="Koty" />
 <moje:OpcjeMenu wartoscOpcji="Konie" />
</moje:Menu>

</body></html>
```

## Wyniki



*W tym przykładzie tak naprawdę NIE robimy niczego ciekawego z opcjami menu, a jedynie udowadniamy, iż udało się nam je pobrać. Można sobie jednak wyobrazić ich wykorzystanie na przykład do wygenerowania paska nawigacyjnego...*

## Klasy obsługi znaczników Menu oraz OpcjaMenu

### Znacznik wewnętrzny: OpcjaMenu

```
public class OpcjaMenu extends TagSupport {
 private String wartoscOpcji;

 public void setWartoscOpcji(String wartosc) {
 wartoscOpcji = wartosc;
 }

 public int doStartTag() throws JspException {
 return EVAL_BODY_INCLUDE;
 }

 public int doEndTag() throws JspException {
 Menu zewn = (Menu) getParent();
 zewn.dodajOpcjeMenu(wartoscOpcji);
 return EVAL_PAGE;
 }
}
```

W deskrytorze TLD znacznika OpcjaMenu zadeklarowano atrybut wartoscOpcji. To właśnie wartość tego atrybutu musimy przekazać do znacznika nadrzędnego...

Nic prostszego — pobieramy referencję do znacznika nadrzędnego i wywołujemy jego metodę dodajOpcjeMenu().

### Znacznik zewnętrzny: Menu

```
public class Menu extends TagSupport {
 private ArrayList opcje;

 public void dodajOpcjeMenu(String opcja) {
 opcje.add(opcja);
 }

 public void doStartTag() throws JspException {
 opcje = new ArrayList();

 return EVAL_BODY_INCLUDE;
 }

 public int doEndTag() throws JspException {
 try {
 pageContext.getOut().println("Opcje menu: " + opcje);
 } catch (Exception ex) {
 throw new JspException("Wyjątek: " + ex.toString());
 }
 // tu możesz sobie wyobrazić złożony kod generujący pasek nawigacyjny
 return EVAL_PAGE;
 }
}
```

To NIE jest metoda określająca wartość atrybutu! Ta metoda istnieje TYLKO po to, aby znaczniki wewnętrzne mogły poinformować znacznik zewnętrzny o wartości atrybutu. (Będzie ona wywoływana pomiędzy wywołaniami metod doStartTag() oraz doEndTag()).

Nie zapomnij wyzerować obiektu ArrayList w metodzie doStartTag(); pamiętaj, że kontener może wielokrotnie używać tego obiektu obsługi znacznika.

Jeśli metoda nie zwróci wartości EVAL\_BODY\_INCLUDE, to znaczniki wewnętrzne nigdy nie zostaną przetworzone!

# Odwoływanie się do dowolnego znacznika zewnętrznego

Istnieje jeszcze inny mechanizm, który możemy wykorzystywać do uzyskiwania dostępu do dowolnego znacznika w hierarchii znaczników zewnętrznych z pominięciem jednego lub kilku przodków bezpośrednich. Metoda, która to umożliwia, jest dostępna zarówno w klasie TagSupport, jak i SimpleTagSupport (choć obie wersje metody różnią się nieznacznie swoim działaniem) i nosi nazwę `findAncestorWithClass()`.

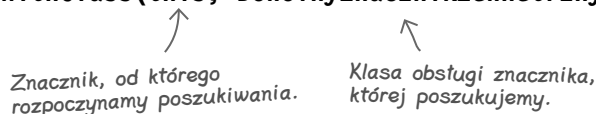
## Pobieranie najbliższego znacznika zewnętrznego przy użyciu metody `getParent()`

```
ZnacznikZewnetrzny zewn = (ZnacznikZewnetrzny) getParent();
```

## Pobieranie dowolnego znacznika nadrzędnego przy użyciu metody `findAncestorWithClass()`

```
DowolnyZnacznikZewnetrzny zewn = (DowolnyZnacznikZewnetrzny)
 findAncestorWithClass(this, DowolnyZnacznikZewnetrzny.class);
```

```
findAncestorWithClass(this, DowolnyZnacznikZewnetrzny.class);
```



Znacznik, od którego rozpoczynamy poszukiwania.

Klasa obsługi znacznika, której poszukujemy.

Kontener przeszukuje hierarchię znaczników aż do momentu odnalezienia znacznika obsługiwane przez wskazaną klasę. Metoda `findAncestorWithClass()` zawsze zwraca *pierwszy* odnaleziony znacznik, zatem nie ma możliwości zaznaczenia, że, na przykład, interesuje nas *drugi* z odnalezionych znaczników tego typu. A zatem, jeśli jesteś pewny, że interesuje Cię drugi znacznik podanego typu, który zostanie odnaleziony, to będziesz musiał odszukać pierwszy z nich, a następnie użyć *go* do wywołania metody `getParent()` lub ponownego wywołania metody `findAncestorWithClass()`.

Na egzaminie nie pojawią się pytania o jakiegokolwiek szczegóły związane z działaniem i stosowaniem metody `findAncestorWithTag()`. Wystarczy, że będziesz wiedział o jej istnieniu.



Ćwiczenie

Kluczowe różnice pomiędzy znacznikami prostymi i klasycznymi.

	Znaczniki proste	Znaczniki klasyczne
Interfejsy		
Klasy pomocnicze implementujące interfejsy		
Kluczowe metody cyklu życia, które MOŻEMY implementować we własnych klasach		
Sposób zapisu danych wyjściowych w odpowiedzi		
Sposób dostępu do zmiennych domyślnych i atrybutów z poziomu klas obsługi		
Sposób przetwarzania zawartości znacznika		
Sposób przerywania przetwarzania strony		



Ćwiczenie  
— rozwiązanie

Kluczowe różnice pomiędzy  
znacznikami prostymi i klasycznymi.

	Znaczniki proste	Znaczniki klasyczne
Interfejsy	<i>SimpleTag</i> (dziedziczy po <i>JspTag</i> )	<i>Tag</i> (dziedziczy po <i>JspTag</i> ) <i>IterationTag</i> (dziedziczy po <i>Tag</i> ) <i>BodyTag</i> (dziedziczy po <i>IterationTag</i> )
Klasy pomocnicze implementujące interfejsy	<i>SimpleTagSupport</i> (implementuje <i>SimpleTag</i> )	<i>TagSupport</i> (implementuje <i>IterationTag</i> ) <i>BodyTagSupport</i> (implementuje <i>BodyTag</i> )
Kluczowe metody cyklu życia, które MOŻEMY implementować we własnych klasach	<i>doTag()</i>	<i>doStartTag()</i> <i>doEndTag()</i> <i>doAfterBody()</i> (a w razie implementowania interfejsu <i>BodyTag</i> : <i>doInitBody()</i> i <i>setBodyContent()</i> ).
Sposób zapisu danych wyjściowych w odpowiedzi	<i>getJspContext().getOut().println</i> (blok try-catch nie jest konieczny, ponieważ metody interfejsu <i>SimpleTag</i> deklarują wyjątek <i>IOException</i> ).	<i>pageContext.getOut().println</i> (wywołanie umieszczone w bloku try-catch, gdyż metody klas obsługi znaczników tworzonych w modelu klasycznym NIE deklarują wyjątku <i>IOException</i> ).
Sposób dostępu do zmiennych domyślnych i atrybutów z poziomu klas obsługi	Za pomocą metody <i>getJspContext()</i> zwracającej obiekt <i>JspContext</i> (który zazwyczaj jest obiektem typu <i>PageContext</i> ).	Przy użyciu zmiennej domyślnej <i>pageContext</i> , a NIE metody (jak w przypadku interfejsu <i>SimpleTag</i> ).
Sposób przetwarzania zawartości znacznika	<i>getJspBody().invoke(null)</i>	W metodzie <i>doStartTag()</i> należy zwrócić wartość <i>EVAL_BODY_INCLUDE</i> lub <i>EVAL_BODY_BUFFERED</i> , jeśli klasa implementuje interfejs <i>BodyTag</i> .
Sposób przerywania przetwarzania strony	Zgłoszenie wyjątku <i>SkipPageException</i> .	Zwrócenie wartości <i>SKIP_PAGE</i> z metody <i>doEndTag()</i> .

# Stosowanie w klasach obsługi znaczników interfejsu API PageContext

Niniejsza strona zawiera wyłącznie powtórzenie informacji zamieszczonych w rozdziale 8., a powtarzamy je dlatego, że mają one kluczowe znaczenie dla klas obsługi znaczników niestandardowych. Musisz pamiętać, że takie klasy nie są ani serwetami, ani stronami JSP i jako takie domyślnie nie dysponują dostępem do różnych obiektów domyślnych. Niemniej jednak w klasach tych można uzyskać dostęp do obiektu typu PageContext, a za jego pośrednictwem do wszelkich innych obiektów, które mogą być nam potrzebne.

Trzeba pamiętać, iż w odróżnieniu od klas obsługi znaczników tworzonych w modelu klasycznym, które dysponują referencją do obiektu typu PageContext, klasy tworzone w modelu prostym dysponują referencją do obiektu JspContext, który zazwyczaj jest jednak obiektem typu PageContext. A zatem, jeśli w klasie obsługi znacznika tworzonej w modelu prostym będziesz musiał zastosować metody lub pola zdefiniowane w klasie PageContext, to konieczne będzie skonwertowanie posiadanej referencji na typ PageContext.

JspContext

getAttribute(String nazwa)  
getAttribute(String nazwa, int zakres)  
getAttributeNamesInScope(int zakres)  
findAttribute(String nazwa)  
getOut()  
  
// więcej metod, w tym metody  
// ustawiające i usuwające atrybuty  
// z innych zasięgów



PageContext

APPLICATION\_SCOPE  
PAGE\_SCOPE  
REQUEST\_SCOPE      pola statyczne  
SESSION\_SCOPE  
  
// inne pola  
  
-----  
getRequest()  
getServletConfig()      metody zwracające dowolne obiekty domyślne  
getServletContext()  
getSession()  
  
// inne metody



Oglądaj to!

**Jednoargumentowa metoda `getAttribute(String)` służy WYŁĄCZNIE do pobierania atrybutów zasięgu strony!**

W klasie PageContext dostępne są dwie przeciążone metody `getAttribute()`. Pierwsza z nich wymaga podania jednego argumentu typu String, a druga dwóch argumentów — typu String oraz typu int. Pierwsza wersja metody działa analogicznie jak pozostałe metody — zwraca atrybuty skojarzone z obiektem pageContext. Z kolei druga wersja metody pozwala na uzyskiwanie atrybutów dostępnych w dowolnym z czterech istniejących zasięgów.



Oglądaj to!

**Metoda `findAttribute()` poszukuje atrybutów w KAŻDYM z czterech zasięgów, począwszy od zasięgu strony — PAGE\_SCOPE.**

Możesz się spodziewać pytania na ten temat! Różnica pomiędzy sposobem działania metod `getAttribute(String)` oraz `findAttribute(String)` może być kolosalna — pierwsza z nich poszukuje WYŁĄCZNIE atrybutów zasięgu strony, natomiast druga operuje na wszystkich czterech zasięgach — odpowiednio strony, żądania, sesji i aplikacji. Metoda `findAttribute()` zwraca pierwszy odnaleziony atrybut.



## Zapamiętywanie sposobu stosowania plików znaczników

ODPOWIEDZI

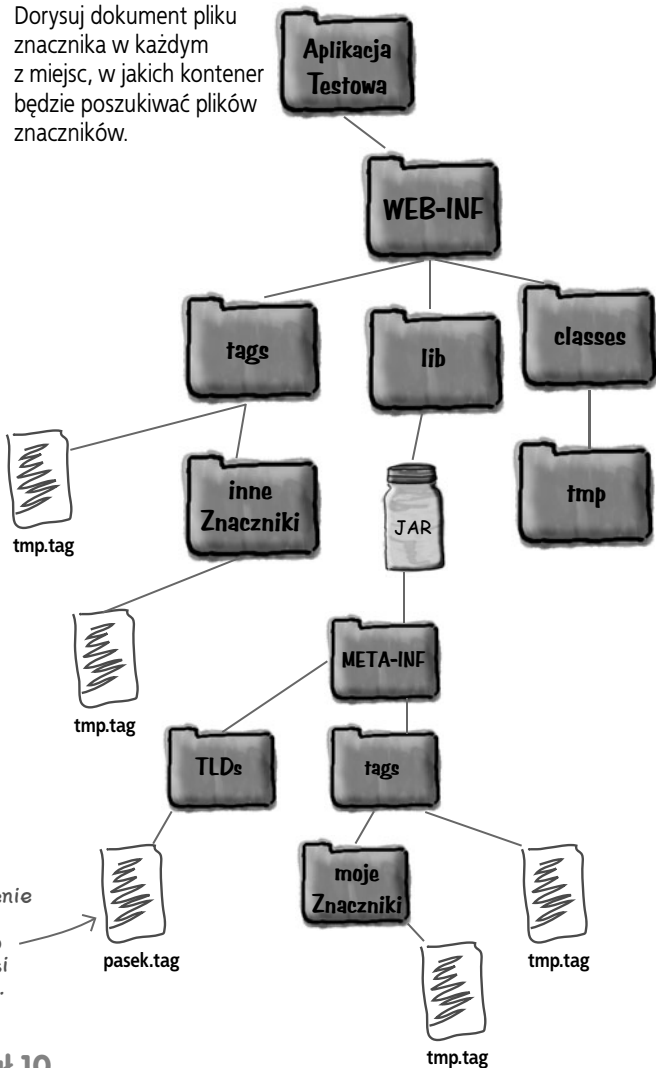
- 1 W poniższej ramce zapisz to, co należałoby umieścić w pliku znacznika, aby zadeklarować, że wymaga on podania jednego atrybutu o nazwie „tytuł”, którego wartością może być wyrażenie EL.

```
<%@ attribute name="tytuł" required="true" rtexprvalue="true" %>
```

- 2 W poniższej ramce zapisz deklarację informującą, że znacznik nie może mieć zawartości.

```
<%@ tag body-content="empty" %>
```

- 3 Dorysuj dokument pliku znacznika w każdym z miejsc, w jakich kontener będzie poszukiwać plików znaczników.



Bezpośrednio w katalogu *WEB-INF/tags*.  
W podkatalogu katalogu *WEB-INF/tags*.  
W katalogu *META-INF/tags* w ramach pliku JAR umieszczonego w katalogu *WEB-INF/lib*.  
W podkatalogu katalogu *META-INF/tags* w ramach pliku JAR umieszczonego w katalogu *WEB-INF/lib*.  
Jeśli plik znacznika umieszczono w archiwum JAR, MUSI dla niego istnieć odpowiedni deskryptor TLD.

Co prawda ćwiczenie nie dotyczyło bezpośrednio tego pliku, jednak musi się on tu znaleźć.



- 
- 1 W jaki sposób klasa obsługi znacznika może wymusić na kontenerze zignorowanie dalszej części strony JSP, w której umieszczono dany znacznik? (Należy zaznaczyć wszystkie poprawne opcje).
- ☐ A. Metoda **doEndTag()** powinna zwrócić wartość **Tag.SKIP\_BODY**.
  - ☐ B. Metoda **doEndTag()** powinna zwrócić wartość **Tag.SKIP\_PAGE**.
  - ☐ C. Metoda **doStartTag()** powinna zwrócić wartość **Tag.SKIP\_BODY**.
  - ☐ D. Metoda **doStartTag()** powinna zwrócić wartość **Tag.SKIP\_PAGE**.
- 
- 2 Które dyrektywy i (lub) standardowe akcje mogą być stosowane WYŁĄCZNIE w plikach znaczników? (Należy zaznaczyć wszystkie poprawne opcje).
- ☐ A. **tag**
  - ☐ B. **page**
  - ☐ C. **jsp:body**
  - ☐ D. **jsp:doBody**
  - ☐ E. **jsp:invoke**
  - ☐ F. **taglib**

- 2 Witryna internetowa prywatnej kliniki ukrywa wybrane elementy przed niezarejestrowanymi użytkownikami. W miejsce ukrywanych informacji powinien być wyświetlany komunikat zachęcający użytkowników do rejestracji. Przyjmijmy, że dysponujemy następującym fragmentem klasy obsługi znacznika prostego:

```
11. public int doTag() throws JspException, IOException {
12. String poziom =
13. (String) getJspContext().findAttribute("poziomKonta");
14. if ((poziom == null || "trial".equals(poziom))) {
15. String cena = "?"; // Musimy uzyskać parametr kontekstu
16. String komunikat = "Materiał tylko dla zarejestrowanych użytkowników.
+
17. "Zarejestruj się za jedyne "+cena+"!";
18. getJspContext().getOut().write(komunikat);
19. } else {
20. getJspBody().invoke(null);
21. }
22. }
```

W wierszu 15. należy uzyskać cenę rejestracji z parametru kontekstu nazwanego `cenaRejestracji`, ale klasa `JspContext` nie definiuje żadnych metod zwracających tego rodzaju parametry. Jak można rozwiązać ten problem?

- ☐ A. Uzyskując tę wartość za pomocą wywołania `pageContext.getServletContext().getInitParameter("cenaRejestracji");`.
- ☐ B. Rzutując obiekt typu `JspContext` na typ `PageContext`, aby można było użyć metod obiektu `PageContext` do uzyskania interesującego nas parametru kontekstu.
- ☐ C. Uzyskując tę wartość za pomocą wywołania `getJspContext().findAttribute("cenaRejestracji");`.
- ☐ D. Generując wyjątek, aby poinformować użytkownika o braku możliwości odnalezienia opłaty za rejestrację.
- ☐ E. Rozwiązanie tego problemu w modelu znaczników prostych jest niemożliwe. Należałoby więc użyć znacznika klasycznego.

4 Który z poniższych mechanizmów stosowanych w klasie obsługi znacznika tworzonej w modelu prostym nakazuje stronie JSP przerwanie dalszego przetwarzania?

- ☐ A. Zwrócenie wartości **SKIP\_PAGE** z metody **doTag()**.
- ☐ B. Zwrócenie wartości **SKIP\_PAGE** z metody **doEndTag()**.
- ☐ C. Zgłoszenie wyjątku **SkipPageException** w metodzie **doTag()**.
- ☐ D. Zgłoszenie wyjątku **SkipPageException** w metodzie **doEndTag()**.

5 Które z poniższych stwierdzeń dotyczących klasycznego modelu obsługi znaczników niestandardowych są prawdziwe? (Należy zaznaczyć wszystkie poprawne opcje).

- ☐ A. Interfejs **Tag** można stosować wyłącznie w przypadkach tworzenia znaczników pustych.
- ☐ B. Stała **SKIP\_PAGE** jest poprawną wartością, która może zostać zwrócona przez metodę **doEndTag()**.
- ☐ C. Stała **EVAL\_BODY\_BUFFERED** jest poprawną wartością, która może zostać zwrócona przez metodę **doAfterBody()**.
- ☐ D. Interfejs **Tag** definiuje jedynie dwie wartości, które mogą być zwracane przez metodę **doStartTag()**: **SKIP\_BODY** oraz **EVAL\_BODY**.
- ☐ E. Istnieją trzy interfejsy dla klas obsługi znaczników (**Tag**, **IterationTag** oraz **BodyTag**) i tylko dwie wbudowane klasy bazowe (**TagSupport** oraz **BodyTagSupport**).

- 6 Które z poniższych stwierdzeń dotyczących metody **findAncestorWithClass()** klasy **TagSupport** są prawdziwe? (Należy zaznaczyć wszystkie poprawne opcje).
- ☐ A. Wymaga podania jednego parametru, typu **Class**.
  - ☐ B. Jest to statyczna metoda klasy **TagSupport**.
  - ☐ C. Jest to metoda klasy **TagSupport**, lecz nie jest to metoda statyczna.
  - ☐ D. Metoda ta nie została zdefiniowana w żadnym ze standardowych interfejsów związanych z przetwarzaniem znaczników JSP.
  - ☐ E. Wymaga dwóch parametrów: typu **Tag** oraz **Class**.
  - ☐ F. Wymaga jednego parametru: typu **String**, który określa nazwę poszukiwanego znacznika.
  - ☐ G. Wymaga podania dwóch parametrów: typu **Tag** oraz **String**, przy czym ten drugi reprezentuje nazwę poszukiwanego znacznika.
- 
- 7 Które z poniższych warunków muszą być spełnione, aby było możliwe korzystanie z atrybutów dynamicznych w klasie obsługi znacznika prostego? (Należy zaznaczyć wszystkie poprawne opcje).
- ☐ A. W klasie obsługi znacznika NIE można zadeklarować żadnych statycznych atrybutów znacznika.
  - ☐ B. W deskrytorze TLD znacznika należy użyć elementu **<dynamic-attribute>**.
  - ☐ C. Klasa obsługi znacznika musi implementować interfejs **DynamicAttributes**.
  - ☐ D. Klasa obsługi znacznika powinna dziedziczyć po klasie **DynamicSimpleTagSupport**, która udostępnia domyślne mechanizmy obsługi dynamicznych atrybutów.
  - ☐ E. Naszego znacznika prostego NIE MOŻEMY używać łącznie ze standardową akcją **jsp:attribute**, ponieważ wspomniana akcja działa tylko z atrybutami statycznymi.

- 8 Które z poniższych stwierdzeń dotyczących plików znaczników są prawdziwe? (Należy zaznaczyć wszystkie poprawne opcje).
- ☐ A. Plik znacznika można umieścić w dowolnym podkatalogu katalogu **WEB-INF**.
  - ☐ B. Plik znacznika musi mieć rozszerzenie **.tag** lub **.tagx**.
  - ☐ C. Konieczne jest użycie deskryptora TLD w celu skojarzenia symbolicznej nazwy znacznika z faktycznym plikiem znacznika.
  - ☐ D. Pliku znacznika NIE można umieścić w pliku JAR znajdującym się w katalogu **WEB-INF/lib**.

9 Dysponujemy poniższym kodem:

```
10. public class ZnacznikBuf extends BodyTagSupport {
11. public int doStartTag() throws JspException {
12. // tu wstaw kod
13. }
14. }
```

Założ, że znacznik został poprawnie skonfigurowany w sposób pozwalający na umieszczanie w nim zawartości.

Która z poniższych instrukcji umieszczona w wierszu 12. spowoduje, że wykonanie kodu JSP: `<mojeZnaczniki:mojZnacznik>Zawartość</mojeZnaczniki:mojZnacznik>` wygeneruje łańcuch znaków **Zawartość**.

- ☐ A. `return SKIP_BODY;`
- ☐ B. `return EVAL_BODY_INCLUDE;`
- ☐ C. `return EVAL_BODY_BUFFERED;`
- ☐ D. `return BODY_CONTENT;`

---

**10** Które z poniższych stwierdzeń dotyczących metody **doAfterBody()** są prawdziwe? (Należy zaznaczyć wszystkie poprawne opcje).

- ☐ A. Metoda ta jest wywoływana wyłącznie dla znaczników, których klasa obsługi dziedziczy po klasie **TagSupport**.
- ☐ B. Metoda ta jest wywoływana wyłącznie dla znaczników, których klasa obsługi dziedziczy po klasie **IterationTagSupport**.
- ☐ C. Zakładając, że nie zostanie zgłoszony żaden wyjątek, to dla dowolnego znacznika, którego klasa obsługi implementuje interfejs **IterationTag**, metoda **doAfterBody()** jest wywoływana po metodzie **doStartTag()**.
- ☐ D. Zakładając, że nie zostanie zgłoszony żaden wyjątek, to dla dowolnego znacznika, którego klasa obsługi implementuje interfejs **IterationTag**, a metoda **doStartTag()** zwraca wartość **SKIP\_BODY**, metoda **doAfterBody()** jest wywoływana po metodzie **doStartTag()**.
- ☐ E. Zakładając, że nie zostanie zgłoszony żaden wyjątek, to dla dowolnego znacznika, którego klasa obsługi implementuje interfejs **IterationTag**, a metoda **doStartTag()** zwraca wartość **EVAL\_BODY\_INCLUDE**, metoda **doAfterBody()** jest wywoływana po metodzie **doStartTag()**.

---

**11** Dysponujemy poniższym kodem JSP:

1. `<%@ taglib prefix="moje" uri="/WEB-INF/mojeZnaczniki.tld" %>`
2. `<moje:znacznik1>`
3. `<%-- kod JSP --%>`
4. `</moje:znacznik1>`

Klasa obsługi znacznika nosi nazwę **ObslugaZnacznika1** i dziedziczy po klasie **TagSupport**.

Co się stanie, kiedy obiekt klasy **ObslugaZnacznika1** wywoła metodę **getParent()**? (Należy zaznaczyć wszystkie poprawne opcje).

- ☐ A. Zostanie zgłoszony wyjątek **JspException**.
- ☐ B. Zostanie zwrócona wartość **null**.
- ☐ C. Zostanie zgłoszony wyjątek **NullPointerException**.
- ☐ D. Zostanie zgłoszony wyjątek **IllegalStateException**.

**12** Które z poniższych stwierdzeń o cyklu życia znacznika prostego są prawdziwe? (Należy zaznaczyć wszystkie poprawne opcje).

- ☐ A. Po wywołaniu metody **doTag()** wywoływana jest metoda **release()**.
- ☐ B. Zawsze przed wywołaniem metody **doBody()** wywoływana jest metoda **setJspBody()**.
- ☐ C. Bezpośrednio przed określeniem wartości atrybutów znacznika wywoływane są metody **setParent()** oraz **setJspContext()**.
- ☐ D. Przed wywołaniem metody **doTag()** klasy obsługi kontener wywołuje obiekt **JspFragment** reprezentujący zawartość znacznika. Wynikowy obiekt, typu **BodyContent**, jest przekazywany do klasy obsługi znacznika przy użyciu metody **setJspBody()**.

**13** Dysponując poniższym kodem:

```

10. public class ZnacznikPrzykladowy extends TagSupport {
11. private String param;
12. public void setParam(String p) { param = p; }
13. public int doStartTag() throws JspException {
14. // tu wstaw kod
15. // dalszy kod metody
16. }
17. }
```

określ, która z poniższych instrukcji umieszczonych w wierszu 14. sprawi, że wartość atrybutu żądania **param** zostanie przypisana do zmiennej lokalnej **p**? (Należy zaznaczyć wszystkie poprawne opcje).

- ☐ A. **String p = findAttribute("param");**
- ☐ B. **String p = request.getAttribute("param");**
- ☐ C. **String p = pageContext.findAttribute("param");**
- ☐ D. **String p = getPageContext().findAttribute("param");**
- ☐ E. **String p = (String) pageContext.getRequest().getAttribute("param");**

**14** Które z poniższych wywołań są prawidłowymi wywołaniami metod obiektu **PageContext**? (Należy zaznaczyć wszystkie poprawne opcje).

- ☐ A. `getAttributeNames()`
- ☐ B. `getAttribute("klucz")`
- ☐ C. `findAttribute("klucz")`
- ☐ D. `getSessionAttribute()`
- ☐ E. `getAttributesScope("klucz")`
- ☐ F. `findAttribute("klucz", PageContext.SESSION_SCOPE)`
- ☐ G. `getAttribute("klucz", PageContext.SESSION_SCOPE)`

---

**15** Które z poniższych wywołań metod klasy **JspContext** jest najlepszym sposobem uzyskania wartości atrybutu zasięgu aplikacji?

- ☐ A. `getPageContext()`
- ☐ B. `getAttribute(String)`
- ☐ C. `findAttribute(String)`
- ☐ D. `getAttribute(String, int)`
- ☐ E. `getAttributesScope("klucz")`
- ☐ F. `getAttributeNamesInScope(int)`

**16** Wskaż najlepsze z zaproponowanych poniżej rozwiązań problemu poszukiwania atrybutu nieznanego zasięgu w klasie obsługi znacznika niestandardowego.

- ☐ A. Sprawdzenie wszystkich zasięgów za pomocą pojedynczego wywołania metody `pageContext.getAttribute(String)`.
- ☐ B. Sprawdzenie wszystkich zasięgów za pomocą pojedynczego wywołania metody `pageContext.findAttribute(String)`.
- ☐ C. Sprawdzenie wszystkich zasięgów za pomocą sekwencji wywołań metody `pageContext.getAttribute(String, int)`.
- ☐ D. Wywołanie metody `pageContext.getRequest().getAttribute(String)`, następnie wywołanie metody `pageContext.getSession().getAttribute(String)` i tak dalej.
- ☐ E. Żadne z powyższych rozwiązań na to nie pozwala.

**17** Dysponujemy znacznikiem **znacznikProsty** obsługiwanym przez klasę stworzoną w modelu prostym oraz znacznikiem **znacznikZlozony** obsługiwanym przez klasę stworzoną w modelu klasycznym. Deklaracje obu tych znaczników umieszczone w deskrytorze TLD zezwalają na umieszczenie zawartości w tych znacznikach.

Które z poniższych fragmentów kodu JSP są przykładami poprawnego użycia tych znaczników. (Należy zaznaczyć wszystkie poprawne opcje).

- ☐ A. `<moje:znacznikProsty>`  
`<moje:znacznikZlozony />`  
`</moje:znacznikProsty>`
- ☐ B. `<moje:znacznikProsty>`  
`<%= wyswietlTekst %>`  
`</moje:znacznikProsty>`
- ☐ C. `<moje:znacznikProsty>`  
`<%@ include file="/WEB-INF/web/common/naglowekMenu.html" %>`  
`</moje:znacznikProsty>`
- ☐ D. `<moje:znacznikProsty>`  
`<moje:znacznikZlozony>`  
`<% i++; %>`  
`</moje:znacznikZlozony>`  
`</moje:znacznikProsty>`

- 18** Które z poniższych stwierdzeń dotyczących modelu plików znaczników jest prawdziwe? Należy zaznaczyć wszystkie poprawne opcje).
- ☐ A. Dla każdego pliku znacznika należy podać niezbędne informacje w deskrytorze TLD.
  - ☐ B. Wszystkie dyrektywy, których można używać w plikach JSP, można także stosować w plikach znaczników.
  - ☐ C. Wszystkie dyrektywy, których można używać w plikach znaczników, można także stosować w plikach JSP.
  - ☐ D. Standardową akcję **<jsp:doBody>** można stosować wyłącznie w plikach znaczników.
  - ☐ E. Dozwolone rozszerzenia plików znaczników to: **.tag** oraz **.tagx**.
  - ☐ F. Dla każdego atrybutu zadeklarowanego i podanego w pliku znacznika kontener tworzy atrybut strony o identycznej nazwie.

- 19** Które z poniższych zapisów w plikach znaczników są poprawne? (Należy zaznaczyć wszystkie poprawne opcje).

- ☐ A. **<jsp:doBody />**
- ☐ B. **<jsp:invoke fragment="fragm" />**
- ☐ C. **<%@ page import="java.util.Date" %>**
- ☐ D. **<% variable name-given="data" variable-class="java.util.Date" %>**
- ☐ E. **<% attribute name="nazwa" value="pusty" type="java.lang.String" %>**

- 20** Które z poniższych wywołań umieszczone w klasie obsługi znacznika zwróci znacznik zewnętrzny? (Należy zaznaczyć wszystkie poprawne opcje).

- ☐ A. **getParent()**
- ☐ B. **getAncestor()**
- ☐ C. **findAncestor()**
- ☐ D. **getEnclosingTag()**

**21** Dysponując aplikacją o następującej strukturze:

`/WEB-INF/tags/ mojeZnaczniki /znacznik1.tag`

`/WEB-INF/tags /znacznik2.tag`

`/WEB-INF /znacznik3.tag`

`/znacznik4.tag`

określ, które ze znaczników mogą być użyte w poprawnej dyrektywie **taglib**? (Należy zaznaczyć wszystkie poprawne opcje).

- ☐ A. **znacznik1.tag**
- ☐ B. **znacznik2.tag**
- ☐ C. **znacznik3.tag**
- ☐ D. **znacznik4.tag**

**22** Hipotetyczna aplikacja internetowa obejmuje wiele formularzy wypełnianych i wysyłanych przez użytkowników. Strony z formularzami w żaden sposób nie informują użytkowników, które pola są wymagane. W pewnym momencie pracownicy biznesowi przedsiębiorstwa doszli do wniosku, że etykiety tekstowe właściwe dla pól wymaganych należy poprzedzić czerwonymi znakami gwiazdki, menedżer projektu upierał się, że należy te pola wyróżnić jasnoniebieskim kolorem tła, a pracownicy jeszcze innego działu zażądali dostosowania wyglądu stron do własnej aplikacji, gdzie etykiety pól wymaganych wyróżnia się pogrubieniem.

Mając na uwadze różne koncepcje identyfikacji pól wymaganych na stronie, wskaż najłatwiejsze w utrzymaniu zastosowanie znacznika niestandardowego, który pozwoli rozwiązać ten problem.

- ☐ A. `<cust:requiredIcon/>Imię: <input type="text" name="imie"/>`
- ☐ B. `<cust:textField label="Imię:" required="true"/>`
- ☐ C. `<cust:requiredField color="red" symbol="*" label="Imię:"/>`
- ☐ D. `<cust:required>`

Imię: `<input type="text" name="imie"/>`  
`</cust:required>`



# BAR KAWOWY

## Egzamin próbny — odpowiedzi

- 
- 1** W jaki sposób klasa obsługi znacznika może wymusić na kontenerze zignorowanie dalszej części strony JSP, w której umieszczono dany znacznik? (Należy zaznaczyć wszystkie poprawne opcje). *(Specyfikacja JSP 2.0 str. 2 – 56).*
- ☐ A. Metoda **doEndTag()** powinna zwrócić wartość **Tag.SKIP\_BODY**.  
– Odpowiedź A. nie jest prawidłowa, gdyż nie jest to poprawna wartość wynikowa metody doEndTag().
  - ☒ B. Metoda **doEndTag()** powinna zwrócić wartość **Tag.SKIP\_PAGE**.  
– Odpowiedź C nie jest prawidłowa, gdyż zwrócenie tej wartości powoduje jedynie pominięcie przetwarzania zawartości znacznika.
  - ☐ C. Metoda **doStartTag()** powinna zwrócić wartość **Tag.SKIP\_BODY**.  
– Odpowiedź D nie jest prawidłowa, gdyż podana wartość nie może być zwracana przez metodę doStartTag().
  - ☐ D. Metoda **doStartTag()** powinna zwrócić wartość **Tag.SKIP\_PAGE**.
- 
- 2** Które dyrektywy i (lub) standardowe akcje mogą być stosowane WYŁĄCZNIE w plikach znaczników? (Należy zaznaczyć wszystkie poprawne opcje). *(Specyfikacja JSP 2.0 8.5 (str. 1 – 179) JSP wersja 2.0 podrozdział 5.11 JSP wersja 2.0 podrozdział 5.12 JSP wersja 2.0 podrozdział 5.13).*
- ☒ A. **tag** – Odpowiedź A jest poprawna (strony 1 – 179).
  - ☐ B. **page** – Odpowiedź B jest błędna, gdyż dyrektywy page nigdy nie wolno używać w plikach znaczników (strony 1 – 179).
  - ☐ C. **jsp:body** – Odpowiedź C jest błędna, gdyż akcja jsp:body może być używana zarówno w plikach znaczników, JAK I w plikach JSP.
  - ☒ D. **jsp:doBody** – Odpowiedź D jest prawidłowa (strony 1 – 121).
  - ☒ E. **jsp:invoke** – Odpowiedź E jest prawidłowa (strony 1 – 119).
  - ☐ F. **taglib** – Odpowiedź F jest błędna, gdyż dyrektywa taglib może być używana ZARÓWNO w plikach znaczników, jak i w plikach JSP.

- 2 Witryna internetowa prywatnej kliniki ukrywa wybrane elementy przed niezarejestrowanymi użytkownikami. W miejsce ukrywanych informacji powinien być wyświetlany komunikat zachęcający użytkowników do rejestracji. Przyjmijmy, że dysponujemy następującym fragmentem klasy obsługi znacznika prostego:

```
11. public int doTag() throws JspException, IOException {
12. String poziom =
13. (String) getJspContext().findAttribute("poziomKonta");
14. if ((poziom == null || "trial".equals(poziom))) {
15. String cena = "?"; // Musimy uzyskać parametr kontekstu
16. String komunikat = "Materiał tylko dla zarejestrowanych użytkowników.
+
17. "Zarejestruj się za jedyne "+cena+"!";
18. getJspContext().getOut().write(komunikat);
19. } else {
20. getJspBody().invoke(null);
21. }
22. }
```

W wierszu 15. należy uzyskać cenę rejestracji z parametru kontekstu nazwanego `cenaRejestracji`, ale klasa `JspContext` nie definiuje żadnych metod zwracających tego rodzaju parametry. Jak można rozwiązać ten problem?

- ☐ A. Uzyskując tę wartość za pomocą wywołania `pageContext.getServletContext().getInitParameter("cenaRejestracji");`.
- ☒ B. Rzutując obiekt typu `JspContext` na typ `PageContext`, aby można było użyć metod obiektu `PageContext` do uzyskania interesującego nas parametru kontekstu.
- ☐ C. Uzyskując tę wartość za pomocą wywołania `getJspContext().findAttribute("cenaRejestracji");`.
- ☐ D. Generując wyjątek, aby poinformować użytkownika o braku możliwości odnalezienia opłaty za rejestrację.
- ☐ E. Rozwiązanie tego problemu w modelu znaczników prostych jest niemożliwe. Należałoby więc użyć znacznika klasycznego.

– Odpowiedź A jest błędna, ponieważ zmienna `pageContext` jest dostępna tylko dla znaczników klasycznych.

– Odpowiedź B jest poprawna. Co prawda nie wspomnieliśmy o tej ciekawej sztuczce i jej znajomość nie jest wymagana na egzaminie, jednak w rzeczywistych zastosowaniach taki zabieg może się okazać niezwykle przydatny.

– Odpowiedź C jest błędna, ponieważ nie interesuje nas atrybut, tylko parametr kontekstu.

– Odpowiedź D jest błędna. Nie poddawaj się tak łatwo! Wystarczy odrobina determinacji, aby znaleźć właściwe rozwiązanie!

– Odpowiedź E jest błędna. Rozwiązanie tego problemu jest trudne, ale nie niemożliwe.

- 4** Który z poniższych mechanizmów stosowanych w klasie obsługi znacznika tworzonej w modelu prostym nakaże stronie JSP przerwanie dalszego przetwarzania? (Specyfikacja JSP 2.0 podrozdział 13.6.1).
- ☐ A. Zwrócenie wartości **SKIP\_PAGE** z metody **doTag()**. – Odpowiedź A jest błędna, gdyż metoda **doTag()** nie zwraca żadnej wartości wynikowej.
  - ☐ B. Zwrócenie wartości **SKIP\_PAGE** z metody **doEndTag()**. – Odpowiedź B jest błędna, gdyż klasy obsługi znaczników tworzone w modelu prostym nie dysponują metodą **doEndTag()**.
  - ☒ C. Zgłoszenie wyjątku **SkipPageException** w metodzie **doTag()**.
  - ☐ D. Zgłoszenie wyjątku **SkipPageException** w metodzie **doEndTag()**. – Odpowiedź D jest błędna, gdyż klasy obsługi znaczników tworzone w modelu prostym nie dysponują metodą **doEndTag()**.
- 
- 5** Które z poniższych stwierdzeń dotyczących klasycznego modelu obsługi znaczników niestandardowych są prawdziwe? (Należy zaznaczyć wszystkie poprawne opcje). (Specyfikacja JSP 2.0 podrozdziały 13.1 oraz 13.2).
- ☐ A. Interfejs **Tag** można stosować wyłącznie w przypadkach tworzenia znaczników pustych. – Odpowiedź A jest błędna, ponieważ interfejs **Tag** obsługuje znaczniki zawierające ciało, ale nie zapewnia możliwości iteracyjnego przetwarzania ani nawet uzyskiwania dostępu do tego ciała.
  - ☒ B. Stała **SKIP\_PAGE** jest poprawną wartością, która może zostać zwrócona przez metodę **doEndTag()**.
  - ☐ C. Stała **EVAL\_BODY\_BUFFERED** jest poprawną wartością, która może zostać zwrócona przez metodę **doAfterBody()**. – Odpowiedź C jest błędna, gdyż metoda **doAfterBody()** może zwracać wyłącznie wartości **SKIP\_BODY** lub **EVAL\_BODY\_AGAIN**.
  - ☐ D. Interfejs **Tag** definiuje jedynie dwie wartości, które mogą być zwracane przez metodę **doStartTag()**: **SKIP\_BODY** oraz **EVAL\_BODY**. – Odpowiedź D jest błędna, gdyż metoda **doStartTag()** może zwracać wyłącznie wartości **SKIP\_BODY** lub **EVAL\_BODY\_INCLUDE**.
  - ☒ E. Istnieją trzy interfejsy dla klas obsługi znaczników (**Tag**, **IterationTag** oraz **BodyTag**) i tylko dwie wbudowane klasy bazowe (**TagSupport** oraz **BodyTagSupport**).

- 6 Które z poniższych stwierdzeń dotyczących metody **findAncestorWithClass()** klasy **TagSupport** są prawdziwe? (Należy zaznaczyć wszystkie poprawne opcje). (Specyfikacja JSP 2.0 str. 2 – 64).
- ☐ A. Wymaga podania jednego parametru, typu **Class**.
  - ☒ B. Jest to statyczna metoda klasy **TagSupport**.
  - ☐ C. Jest to metoda klasy **TagSupport**, lecz nie jest to metoda statyczna. – Odpowiedź C jest błędna, gdyż metoda **findAncestorWithClass** jest metodą statyczną.
  - ☒ D. Metoda ta nie została zdefiniowana w żadnym ze standardowych interfejsów związanych z przetwarzaniem znaczników JSP.
  - ☒ E. Wymaga dwóch parametrów: typu **Tag** oraz **Class**.
  - ☐ F. Wymaga jednego parametru: typu **String**, który określa nazwę poszukiwanego znacznika. – Odpowiedzi A oraz F są błędne, gdyż metoda ta wymaga podania dwóch parametrów.
  - ☐ G. Wymaga podania dwóch parametrów: typu **Tag** oraz **String**, przy czym ten drugi reprezentuje nazwę poszukiwanego znacznika. – Odpowiedź G jest błędna, gdyż drugi argument metody jest typu **Class**.

- 7 Które z poniższych warunków muszą być spełnione, aby było możliwe korzystanie z atrybutów dynamicznych w klasie obsługi znacznika prostego? (Należy zaznaczyć wszystkie poprawne opcje). (Specyfikacja JSP 2.0 podrozdział 13.3 str. 2 – 74, 75).
- ☐ A. W klasie obsługi znacznika NIE można zadeklarować żadnych statycznych atrybutów znacznika. – Odpowiedź A jest błędna, gdyż klasy obsługi tworzone w modelu prostym mogą obsługiwać zarówno atrybuty statyczne, jak i dynamiczne.
  - ☒ B. W deskrytorze TLD znacznika należy użyć elementu **<dynamic-attribute>**.
  - ☒ C. Klasa obsługi znacznika musi implementować interfejs **DynamicAttributes**. – Odpowiedź D jest błędna, ponieważ wbudowane interfejsy API w ogóle nie definiują takiej klasy pomocniczej.
  - ☐ D. Klasa obsługi znacznika powinna dziedziczyć po klasie **DynamicSimpleTagSupport**, która udostępnia domyślne mechanizmy obsługi dynamicznych atrybutów.
  - ☐ E. Naszego znacznika prostego NIE MOŻEMY używać łącznie ze standardową akcją **jsp:attribute**, ponieważ wspomniana akcja działa tylko z atrybutami statycznymi. – Odpowiedź E jest błędna, gdyż w znacznikach dynamicznych można używać akcji **jsp:attribute**.

- 8 Które z poniższych stwierdzeń dotyczących plików znaczników są prawdziwe? (JSP wersja 2.0 podrozdział 8.4).  
(Należy zaznaczyć wszystkie poprawne opcje).
- ☐ A. Plik znacznika można umieścić w dowolnym podkatalogu katalogu **WEB-INF**.  
– Odpowiedź A jest błędna, gdyż pliki znaczników należy umieszczać w katalogu **WEB-INF/tags**.
  - ☒ B. Plik znacznika musi mieć rozszerzenie **.tag** lub **.tagx**.  
– Odpowiedź B jest prawidłowa (strony 1 – 176, podrozdział 8.4.1).
  - ☐ C. Konieczne jest użycie deskryptora TLD w celu skojarzenia symbolicznej nazwy znacznika z faktycznym plikiem znacznika.  
– Odpowiedź C jest błędna, gdyż kontener poszukuje plików znaczników w kilku ściśle określonych miejscach. Konieczność kojarzenia nazwy i pliku znacznika jest opcjonalna i zależy od kontenera.
  - ☐ D. Pliku znacznika NIE można umieścić w pliku JAR znajdującym się w katalogu **WEB-INF/lib**.

- 9 Dysponujemy poniższym kodem: (Specyfikacja JSP 2.0 str. 2 – 68).

```
10. public class ZnacznikBuf extends BodyTagSupport {
11. public int doStartTag() throws JspException {
12. // tu wstaw kod
13. }
14. }
```

Założ, że znacznik został poprawnie skonfigurowany w sposób pozwalający na umieszczanie w nim zawartości.

Która z poniższych instrukcji umieszczona w wierszu 12. spowoduje, że wykonanie kodu JSP:  
`<mojeZnaczniki:mojZnacznik>Zawartość</mojeZnaczniki:mojZnacznik>` wygeneruje łańcuch znaków **Zawartość**.

- ☐ A. `return SKIP_BODY;`  
– Odpowiedź A jest błędna, gdyż zwrócenie tej wartości powoduje, że zawartość znacznika zostanie pominięta.
- ☒ B. `return EVAL_BODY_INCLUDE;`
- ☐ C. `return EVAL_BODY_BUFFERED;`  
– Odpowiedź C jest błędna, gdyż jej zwrócenie powoduje zapisanie zawartości znacznika w buforze, którego ten znacznik nie przetwarza.
- ☐ D. `return BODY_CONTENT;`  
– Odpowiedź D jest błędna, gdyż metoda `doStartTag()` nie może zwracać takiej wartości.

- 10** Które z poniższych stwierdzeń dotyczących metody **doAfterBody()** są prawdziwe? (Należy zaznaczyć wszystkie poprawne opcje). (Specyfikacja JSP 2.0 str. 1 – 152).
- ☐ A. Metoda ta jest wywoływana wyłącznie dla znaczników, których klasa obsługi dziedziczy po klasie **TagSupport**. – Odpowiedź A jest błędna, gdyż metodę **doAfterBody()** można wywoływać dla wszystkich znaczników, których klasy obsługi implementują interfejs **IterationTag**.
  - ☐ B. Metoda ta jest wywoływana wyłącznie dla znaczników, których klasa obsługi dziedziczy po klasie **IterationTagSupport**. – Odpowiedź B jest błędna – taka klasa nie istnieje.
  - ☐ C. Zakładając, że nie zostanie zgłoszony żaden wyjątek, to dla dowolnego znacznika, którego klasa obsługi implementuje interfejs **IterationTag**, metoda **doAfterBody()** jest wywoływana po metodzie **doStartTag()**. – Odpowiedzi C i D są błędne, gdyż metoda **doAfterBody()** jest wywoływana wyłącznie w sytuacji, gdy metoda **doStartTag()** zwróci wartość **EVAL\_BODY\_INCLUDE**.
  - ☐ D. Zakładając, że nie zostanie zgłoszony żaden wyjątek, to dla dowolnego znacznika, którego klasa obsługi implementuje interfejs **IterationTag**, a metoda **doStartTag()** zwraca wartość **SKIP\_BODY**, metoda **doAfterBody()** jest wywoływana po metodzie **doStartTag()**.
  - ☒ E. Zakładając, że nie zostanie zgłoszony żaden wyjątek, to dla dowolnego znacznika, którego klasa obsługi implementuje interfejs **IterationTag**, a metoda **doStartTag()** zwraca wartość **EVAL\_BODY\_INCLUDE**, metoda **doAfterBody()** jest wywoływana po metodzie **doStartTag()**.

- 11** Dysponujemy poniższym kodem JSP: (Specyfikacja JSP wersja 2.0 TagSupport API, str. 2 – 64).

```
1. <%@ taglib prefix="moje" uri="/WEB-INF/mojeZnaczniki.tld" %>
2. <moje:znacznik1>
3. <!-- kod JSP -->
4. </moje:znacznik1>
```

Klasa obsługi znacznika nosi nazwę **ObslugaZnacznika1** i dziedziczy po klasie **TagSupport**.

Co się stanie, kiedy obiekt klasy **ObslugaZnacznika1** wywoła metodę **getParent()**? (Należy zaznaczyć wszystkie poprawne opcje).

- ☐ A. Zostanie zgłoszony wyjątek **JspException**.
- ☒ B. Zostanie zwrócona wartość **null**. – Odpowiedź B jest prawidłowa. Metoda **getParent()** nie zgłasza żadnych wyjątków.
- ☐ C. Zostanie zgłoszony wyjątek **NullPointerException**.
- ☐ D. Zostanie zgłoszony wyjątek **IllegalStateException**.

- 12** Które z poniższych stwierdzeń o cyklu życia znacznika prostego są prawdziwe? (Należy zaznaczyć wszystkie poprawne opcje). (Specyfikacja JSP 2.0, sekcja 13.6, str. 2 – 80/83).
- ☐ A. Po wywołaniu metody **doTag()** wywoływana jest metoda **release()**. – Odpowiedź A jest błędna, gdyż znaczniki proste nie mają metody **release()**.
  - ☐ B. Zawsze przed wywołaniem metody **doBody()** wywoływana jest metoda **setJspBody()**. – Odpowiedź B jest błędna, ponieważ metoda **setJspBody()** w ogóle nie jest wywoływana dla pustego znacznika prostego.
  - ☒ C. Bezpośrednio przed określeniem wartości atrybutów znacznika wywoływane są metody **setParent()** oraz **setJspContext()**.
  - ☐ D. Przed wywołaniem metody **doTag()** klasy obsługi kontener wywołuje obiekt **JspFragment** reprezentujący zawartość znacznika. Wynikowy obiekt, typu **BodyContent**, jest przekazywany do klasy obsługi znacznika przy użyciu metody **setJspBody()**. – Odpowiedź D jest błędna, gdyż fragment jest wywoływany przez implementację metody **doTag()**, a NIE zanim zostanie wywołana metoda **doTag()**.

- 13** Dysponując poniższym kodem: (Specyfikacja JSP 2.0, str. 2 – 27).

```

10. public class ZnacznikPrzykladowy extends TagSupport {
11. private String param;
12. public void setParam(String p) { param = p; }
13. public int doStartTag() throws JspException {
14. // tu wstaw kod
15. // dalszy kod metody
16. }
17. }

```

określ, która z poniższych instrukcji umieszczonych w wierszu 14. sprawi, że wartość atrybutu żądania **param** zostanie przypisana do zmiennej lokalnej **p**? (Należy zaznaczyć wszystkie poprawne opcje).

- ☐ A. **String p = findAttribute("param");** – Odpowiedź A jest błędna, gdyż taka metoda nie istnieje.
- ☐ B. **String p = request.getAttribute("param");** – Odpowiedź B jest błędna, gdyż zmienna **request** nie jest dostępna.
- ☐ C. **String p = pageContext.findAttribute("param");** – Odpowiedź C jest błędna, gdyż atrybuty dostępne w zasięgu strony zostałyby odnalezione przed atrybutami zasięgu żądania.
- ☐ D. **String p = getPageContext().findAttribute("param");**
- ☒ E. **String p = (String) pageContext.getRequest().getAttribute("param");** – Odpowiedź D jest błędna, gdyż nie istnieje żadna metoda o nazwie **getPageContext()**.

**14** Które z poniższych wywołań są prawidłowymi wywołaniami metod obiektu **pageContext**? (Specyfikacja JSP 2.0, str. 2 – 23).  
(Należy zaznaczyć wszystkie poprawne opcje).

- ☐ A. `getAttributeNames()`
- ☒ B. `getAttribute("klucz")`
- ☒ C. `findAttribute("klucz")` – Odpowiedzi A oraz D są błędne, gdyż metody o takich nazwach nie istnieją.
- ☐ D. `getSessionAttribute()`
- ☒ E. `getAttributesScope("klucz")`
- ☐ F. `findAttribute("klucz", PageContext.SESSION_SCOPE)` – Odpowiedź F jest błędna, ponieważ metoda `findAttribute()` nie pobiera argumentu określającego zasięg.
- ☒ G. `getAttribute("klucz", PageContext.SESSION_SCOPE)`

**15** Które z poniższych wywołań metod klasy **JspContext** jest najlepszym sposobem uzyskania wartości atrybutu zasięgu aplikacji? (Specyfikacja JSP 2.0, str. 2 – 23).

- ☐ A. `getPageContext()` – Odpowiedź A jest błędna, gdyż taka metoda nie istnieje.
- ☐ B. `getAttribute(String)` – Odpowiedź B jest błędna, ponieważ użyta metoda przeszukuje tylko atrybuty zasięgu strony.
- ☐ C. `findAttribute(String)` – Odpowiedź C jest błędna, ponieważ reprezentuje mniej efektywne rozwiązanie niż wywołanie metody podanej w odpowiedzi D – metoda z odpowiedzi C w pierwszej kolejności przeszuka pozostałe trzy zasięgi.
- ☒ D. `getAttribute(String, int)`
- ☐ E. `getAttributesScope("klucz")` – Odpowiedź E jest błędna, ponieważ taka metoda w ogóle nie istnieje.
- ☐ F. `getAttributeNamesInScope(int)` – Odpowiedź F jest błędna, ponieważ przedstawione wywołanie jest tylko pierwszym krokiem dłuższego procesu, który i tak byłby nieporównanie mniej efektywny od wywołania z odpowiedzi D.

- 16** Wskaż najlepsze z zaproponowanych poniżej rozwiązań problemu poszukiwania atrybutu nieznanego zasięgu w klasie obsługi znacznika niestandardowego. (Specyfikacja JSP 2.0, str. 2 – 23).

- ☐ A. Sprawdzenie wszystkich zasięgów za pomocą pojedynczego wywołania metody `pageContext.getAttribute(String)`. – Odpowiedź A jest błędna, ponieważ użyta metoda przeszukuje tylko atrybuty zasięgu strony.
- ☒ B. Sprawdzenie wszystkich zasięgów za pomocą pojedynczego wywołania metody `pageContext.findAttribute(String)`. – Odpowiedzi C i D są błędne, gdyż ich zastosowanie byłoby mniej efektywnym rozwiązaniem niż wywołanie metody `findAttribute()`.
- ☐ C. Sprawdzenie wszystkich zasięgów za pomocą sekwencji wywołań metody `pageContext.getAttribute(String, int)`.
- ☐ D. Wywołanie metody `pageContext.getRequest().getAttribute(String)`, następnie wywołanie metody `pageContext.getSession().getAttribute(String)` i tak dalej.
- ☐ E. Żadne z powyższych rozwiązań na to nie pozwala.

- 17** Dysponujemy znacznikiem **znacznikProsty** obsługiwany przez klasę stworzoną w modelu prostym oraz znacznikiem **znacznikZlozony** obsługiwany przez klasę stworzoną w modelu klasycznym. Deklaracje obu tych znaczników umieszczone w deskrytorze TLD zezwalają na umieszczenie zawartości w tych znacznikach. (Specyfikacja JSP 2.0, podrozdział 7.1.6, str. 1 – 156).

Które z poniższych fragmentów kodu JSP są przykładami poprawnego użycia tych znaczników. (Należy zaznaczyć wszystkie poprawne opcje).

- ☒ A. `<moje:znacznikProsty>`  
`<moje:znacznikZlozony />`  
`</moje:znacznikProsty>` – Odpowiedź A jest poprawna. Wewnątrz znacznika prostego można umieścić znacznik klasyczny, o ile tylko nie zawiera on żadnego kodu skryptowego.
- ☐ B. `<moje:znacznikProsty>`  
`<%= wyswietlTeskt %>`  
`</moje:znacznikProsty>` – Odpowiedź B jest błędna, ponieważ ciała znaczników prostych nie mogą zawierać znaczników wyrażeń JSP.
- ☒ C. `<moje:znacznikProsty>`  
`<%@ include file="/WEB-INF/web/common/naglowekMenu.html" %>`  
`</moje:znacznikProsty>` – Odpowiedź C jest poprawna, ponieważ dyrektywa `include` jest przetwarzana, zanim zawartość znacznika prostego zostanie zamieniona na obiekt `JspFragment`; z drugiej strony, dołączana zawartość nie może zawierać kodu skryptowego (stąd decyzja o dołączeniu zwykłego pliku HTML).
- ☐ D. `<moje:znacznikProsty>`  
`<moje:znacznikZlozony>`  
`<% i++; %>`  
`</moje:znacznikZlozony>`  
`</moje:znacznikProsty>` – Odpowiedź D nie jest błędna ze względu na sposób użycia znacznika `znacznikZlozony` (podobnie jak w przypadku odpowiedzi A), lecz dlatego, iż wewnątrz tego znacznika został umieszczony kod skryptowy.

- 18 Które z poniższych stwierdzeń dotyczących modelu plików znaczników jest prawdziwe? (Specyfikacja JSP 2.0, str. 1 – 173).  
Należy zaznaczyć wszystkie poprawne opcje).
- ☐ A. Dla każdego pliku znacznika należy podać niezbędne informacje w deskryptorze TLD.
  - ☐ B. Wszystkie dyrektywy, których można używać w plikach JSP, można także stosować w plikach znaczników.
  - ☐ C. Wszystkie dyrektywy, których można używać w plikach znaczników, można także stosować w plikach JSP.
  - ☒ D. Standardową akcję **<jsp:doBody>** można stosować wyłącznie w plikach znaczników.
  - ☒ E. Dozwolone rozszerzenia plików znaczników to: **.tag** oraz **.tagx**.
  - ☒ F. Dla każdego atrybutu zadeklarowanego i podanego w pliku znacznika kontener tworzy atrybut strony o identycznej nazwie.
- Odpowiedź A jest błędna, gdyż możliwość użycia plików znaczników wiąże się z koniecznością umieszczenia ich w odpowiednim miejscu.  
– Odpowiedź B jest błędna, gdyż dyrektywy **page** nie można stosować w plikach znaczników.  
– Odpowiedź C jest błędna, gdyż dyrektyw **page**, **attribute** oraz **variable** nie można stosować w plikach JSP.

- 19 Które z poniższych zapisów w plikach znaczników są poprawne? (Specyfikacja JSP 2.0, str. 1 – 174).  
(Należy zaznaczyć wszystkie poprawne opcje).
- ☒ A. **<jsp:doBody />**
  - ☒ B. **<jsp:invoke fragment="fragm" />**
  - ☐ C. **<%@ page import="java.util.Date" %>**
  - ☒ D. **<% variable name-given="data" variable-class="java.util.Date" %>**
  - ☐ E. **<% attribute name="nazwa" value="pusty" type="java.lang.String" %>**
- Odpowiedź C jest błędna, gdyż dyrektywy **page** nie można używać w plikach znaczników.  
– Odpowiedź E jest błędna, gdyż dyrektywa **attribute** nie posiada atrybutu o nazwie **value**.

- 20 Które z poniższych wywołań umieszczone w klasie obsługi znacznika zwróci znacznik wewnętrzny? (Należy zaznaczyć wszystkie poprawne opcje). (Specyfikacja JSP 2.0, str. 2 – 53).
- ☒ A. **getParent()**
  - ☐ B. **getAncestor()**
  - ☐ C. **findAncestor()**
  - ☐ D. **getEnclosingTag()**
- Odpowiedź A jest poprawna, żadna z pozostałych metod nie istnieje.

21 Dysponując aplikacją o następującej strukturze:

(JSP 2.0, strony 1 – 176).

/WEB-INF/tags/ mojeZnaczniki/znacznik1.tag

/WEB-INF/tags/znacznik2.tag

/WEB-INF/znacznik3.tag

/znacznik4.tag

określ, które ze znaczników mogą być użyte w poprawnej dyrektywie **taglib?**  
(Należy zaznaczyć wszystkie poprawne opcje).

☒ A. **znacznik1.tag**

☒ B. **znacznik2.tag**

☐ C. **znacznik3.tag**

☐ D. **znacznik4.tag**

– Odpowiedzi C oraz D są błędne, ponieważ pliki znaczników należy umieszczać w katalogu WEB-INF/tags lub w jednym z jego podkatalogów.

22 Hipotetyczna aplikacja internetowa obejmuje wiele formularzy wypełnianych i wysyłanych przez użytkowników. Strony z formularzami w żaden sposób nie informują użytkowników, które pola są wymagane. W pewnym momencie pracownicy biznesowi przedsiębiorstwa doszli do wniosku, że etykiety tekstowe właściwe dla pól wymaganych należy poprzedzić czerwonymi znakami gwiazdki, menedżer projektu upierał się, że należy te pola wyróżnić jasnoniebieskim kolorem tła, a pracownicy jeszcze innego działu zażądali dostosowania wyglądu stron do własnej aplikacji, gdzie etykiety pól wymaganych wyróżnia się pogrubieniem.

(JSP 2.0, rozdział 7.).

Mając na uwadze różne koncepcje identyfikacji pól wymaganych na stronie, wskaż najłatwiejsze w utrzymaniu zastosowanie znacznika niestandardowego, który pozwoli rozwiązać ten problem.

☐ A. **<cust:requiredIcon/>Imię: <input type="text" name="imie"/>**

☒ B. **<cust:textField label="Imię:" required="true"/>**

☐ C. **<cust:requiredField color="red" symbol="\*" label="Imię:"/>**

☐ D. **<cust:required>**

**Imię: <input type="text" name="imie"/>**  
**</cust:required>**

– Odpowiedź D jest błędna, ponieważ wymagałaby od klasy implementującej nasz znacznik poddawania jego zawartości analizie składniowej i dalszego przetwarzania. Konserwacja takiego kodu byłaby koszmarem.

– Odpowiedź A byłaby prawidłowa, gdybyśmy mieli pewność, że wymagane pole zawsze będzie poprzedzane odpowiednim symbolem i że potencjalne zmiany będą się ograniczały tylko do wyboru tego symbolu. Z drugiej strony, nawet wówczas równie proste byłoby użycie znacznika img i ewentualna wymiana ikon .gif w katalogu obrazów.

– Odpowiedź B reprezentuje najbardziej elastyczne rozwiązanie. Nasz znacznik niestandardowy zyskuje pełną kontrolę nad procesem konstruowania i wyświetlania etykiet i pól tekstowych.

– Odpowiedź C jest błędna, ponieważ określanie koloru i symbolu w samym znaczniku nie gwarantuje należytej elastyczności — zmiana którejkolwiek z tych wartości wymagałaby bowiem odpowiednich aktualizacji wszystkich znaczników we wszystkich stronach JSP.

## 11. Wdrażanie aplikacji internetowych

# Jak wdrożyć aplikację internetową?



**W końcu Twoja aplikacja jest gotowa.** Strony zostały dopracowane w najdrobniejszych szczegółach, kod jest przetestowany i zoptymalizowany, a termin... minął dwa tygodnie temu. Ale gdzie to wszystko należy umieścić? Jest tyle różnych katalogów, tyle zasad. Jakie nazwy *powinieneś* nadać katalogom? Jakie nazwy nadałby *klient*? Do jakich zasobów tak naprawdę będzie się odwoływać klient i skąd kontener ma wiedzieć, gdzie ich należy szukać? W jaki sposób możesz się upewnić, że podczas przenoszenia całej aplikacji na inny komputer nie zapomnisz o jakimś katalogu? Co zrobić, jeśli klient skieruje żądania dotyczące nie konkretnego *pliku*, lecz całego *katalogu*? W jaki sposób w deskryptorze wdrożenia można skonfigurować strony zawierające informacje o napotykanym błędach i problemach, pliki powitalne oraz typy MIME? To wszystko nie jest aż tak złożone jak z pozoru mogłoby się wydawać...



### Wdrażanie aplikacji internetowych

- 2.1.** Skonstruuj strukturę plików i katalogów aplikacji internetowej, która może zawierać: (a) treści statyczne, (b) strony JSP, (c) klasy serwetów, (d) deskryptor wdrożenia, (e) biblioteki znaczników, (f) pliki JAR oraz (g) pliki klas Javy. Opisz, w jaki sposób można zabezpieczyć te zasoby przed nieupoważnionym dostępem przy wykorzystaniu protokołu HTTP.
  - 2.2.** Opisz przeznaczenie i semantykę każdego z wymienionych elementów deskryptora wdrożenia: `error-page`, `init-param`, `mime-mapping`, `servlet`, `servlet-class`, `servlet-mapping`, `servlet-name` oraz `welcome-file`.
  - 2.3.** Podaj poprawną postać każdego z poniższych elementów deskryptora wdrożenia: `error-page`, `init-param`, `mime-mapping`, `servlet`, `servlet-class`, `servlet-name` oraz `welcome-file`.
  - 2.4.** Opisz przeznaczenie plików WAR, ich zawartość oraz sposób, w jaki te pliki można tworzyć.
- 
- 6.3.** Napisz poprawny składniowo dokument JSP (z wykorzystaniem składni XML).

### Uwagi wyjaśniające:

*Ten cel egzaminacyjny był opisywany w różnych rozdziałach we wcześniejszej części książki, dlatego większość informacji na jego temat w niniejszym rozdziale służy albo powtórzeniu już przekazanej wiedzy, albo uzupełnieniu wcześniejszych wyjaśnień.*

*Cele egzaminacyjne 2.2 oraz 2.3 koncentrują się głównie na szczegółach związanych z tworzeniem znaczników XML umieszczanych w deskrytorze wdrożenia. Choć możliwe, że jest to najmniej interesująca część niniejszej książki (jak również egzaminu), to jednak większość informacji związanych z tymi zagadnieniami jest prosta i sprowadza się w zasadzie do zapamiętania używanych znaczników.*

*Tylko jedno zagadnienie — odwzorowanie serwetów — jest bardziej złożone, dlatego też poświęcimy mu najwięcej uwagi.*

*Zdecydowaliśmy się umieścić ten cel egzaminacyjny w niniejszym rozdziale z dwóch powodów: 1) większość zamieszczonych w nim informacji jest związana z XML-em oraz 2) nie chcieliśmy dodawać nowych informacji do rozdziałów poświęconych zagadnieniom JSP. Zdecydowaliśmy, że lepiej będzie, jeśli skoncentrujesz się na zagadnieniach związanych ze składnią i działaniem wszelkich innych aspektów JSP, niż gdybyś miał się dodatkowo przejmować, czy wygląda jak XML-owa wersja kodu JSP. Ale teraz, kiedy jesteś już... no wiesz — **ekspertem** ... sądzimy, że poradzisz sobie z tymi zagadnieniami.*

## Radość wdrażania

We wcześniejszych rozdziałach książki opisaliśmy już większość najbardziej interesujących i ciekawych zagadnień, teraz zaś nadszedł czas na umieszczenie aplikacji na serwerze.

W niniejszym rozdziale będziesz musiał skoncentrować się na trzech podstawowych zagadnieniach.

### ① **Gdzie twórca — czyli TY — umieszcza różne zasoby wchodzące w skład aplikacji?**

Gdzie należy umieszczać zasoby statyczne? Strony JSP? Pliki klas serwletów? Pliki klas komponentów JavaBean? Pliki klas obiektów nasłuchujących? Klasy obsługi znaczników? Deskryptory TLD? Pliki JAR? Deskryptor wdrożenia *web.xml*? Gdzie umieszczać zasoby, których kontener nie powinien udostępniać? (Innymi słowy, które fragmenty aplikacji są zabezpieczone przed bezpośrednim dostępem ze strony klienta?) Gdzie są umieszczane pliki „powitalne”?

### ② **W jakich miejscach KONTENER będzie poszukiwać różnych zasobów wchodzących w skład aplikacji?**

Gdzie kontener będzie poszukiwać dokumentów HTML żądanych przez klienta? Gdzie będą poszukiwane strony JSP, serwlety, a gdzie zasoby, którym nie odpowiadają pliki (na przykład `TestPiwa.do`)? Gdzie kontener będzie poszukiwał klas obsługi znaczników? Gdzie będą poszukiwane deskryptory TLD, pliki klas, plik JAR, deskryptor wdrożenia? Gdzie będą poszukiwane klasy, od których zależy poprawne działanie stworzonych serwletów? Gdzie kontener będzie poszukiwać plików „powitalnych”? (Oczywiście, kiedy poznasz odpowiedzi na te pytania, to pytania z poprzedniego punktu staną się dziecinnie proste).

### ③ **W jaki sposób KLIENT odwołuje się do różnych zasobów wchodzących w skład aplikacji?**

Co użytkownik musi wpisać w przeglądarce, aby wyświetlić dokument HTML? A co, jeśli chce wyświetlić stronę JSP albo serwlet? W jaki sposób należy odwoływać się do zasobu, który w rzeczywistości nie jest plikiem? W jakich miejscach aplikacji klient może żądać bezpośredniego dostępu do zasobów, a gdzie taki dostęp jest niedozwolony? Co się stanie, gdy użytkownik wpisze w przeglądarce adres katalogu, a nie pliku?

# Gdzie należy umieszczać poszczególne zasoby aplikacji internetowej?

W kilku wcześniejszych rozdziałach niniejszej książki przedstawialiśmy miejsca, w jakich muszą być umieszczane konkretne pliki. Na przykład w rozdziale poświęconym znacznikom niestandardowym dowiedziałeś się, że pliki znaczników muszą być umieszczane w katalogu *WEB-INF/tags* lub jego podkatalogach bądź też w pliku JAR przechowywanym w katalogu */META-INF/tags* lub jego podkatalogach. Jeśli plik znacznika zostanie umieszczony w jakimkolwiek innym miejscu, to kontener zignoruje go lub potraktuje jako zwyczajny plik statyczny, który można zwrócić do klienta.

Specyfikacja serwetów i stron JSP zawiera wiele szczegółowych zasad określających, gdzie należy umieszczać różne zasoby, a co więcej, konieczne jest opanowanie większości z tych zasad. Ponieważ w taki bądź inny sposób we wcześniejszych rozdziałach przedstawiliśmy już znaczną część tych informacji, zatem kilka pierwszych stron tego rozdziału potraktujemy jako swoisty test sprawdzający Twoją pamięć oraz wiadomości. Nie pomijaj ich! Potraktuj je jako ćwiczenie egzaminacyjne!

## Nie ma niemądrych pytań

**❓ Dlaczego powinienem wiedzieć, gdzie należy umieszczać zasoby różnego typu? Czy nie do tego są przeznaczone narzędzia do umieszczania aplikacji na serwerze? Albo nawet skrypty wykonywane przez program ANT?**

**U.** Jeśli masz szczęście, to możesz używać narzędzi służących do wdrażania aplikacji J2EE na serwerze, które pozwalają na podawanie informacji o aplikacji w sekwencji okien dialogowych. Kontener używa następnie tych informacji do stworzenia pliku XML zawierającego deskryptor wdrożenia (ang. *Deployment Descriptor*, czyli pliku *web.xml*), stworzenia odpowiedniej struktury katalogów oraz umieszczenia w nich plików. Niemniej jednak, nawet jeśli *masz* to szczęście, to czy nie uważasz, że powinieneś wiedzieć, co robi narzędzie, którego używasz? Być może będziesz musiał nieco zmienić rezultaty jego działania. Być może będziesz musiał rozwiązać jakieś problemy. Albo zmienić dostawcę używanych narzędzi programistycznych, przez co stracisz możliwość automatycznego umieszczania swoich aplikacji na serwerze.

Wielu programistów używa takich programów jak ANT, jednak nie oznacza to wcale, że nie musisz wiedzieć, co ten program robi.

**❓ Ale właśnie udało mi się znaleźć w Internecie skrypt programu ANT, który potrafi to wszystko zrobić za mnie.**

**U.** I świetnie! Jednak i tak musisz wiedzieć, co się dzieje z Twoją aplikacją. Jeśli całkowicie zdasz się na łaskę i niełaskę używanego narzędzia, to znajdziesz się w poważnych tarapatkach, kiedy coś pójdzie nie tak jak trzeba. Znajomość struktury aplikacji internetowej można porównać z umiejętnością zmiany opony w samochodzie — prawdopodobnie nigdy nie będziesz musiał z niej korzystać, jeśli jednak jest trzecia w nocy, a Ty znajdujesz się gdzieś, nie wiadomo gdzie, to dobrze jest mieć świadomość, że w razie czego *możesz* to zrobić.

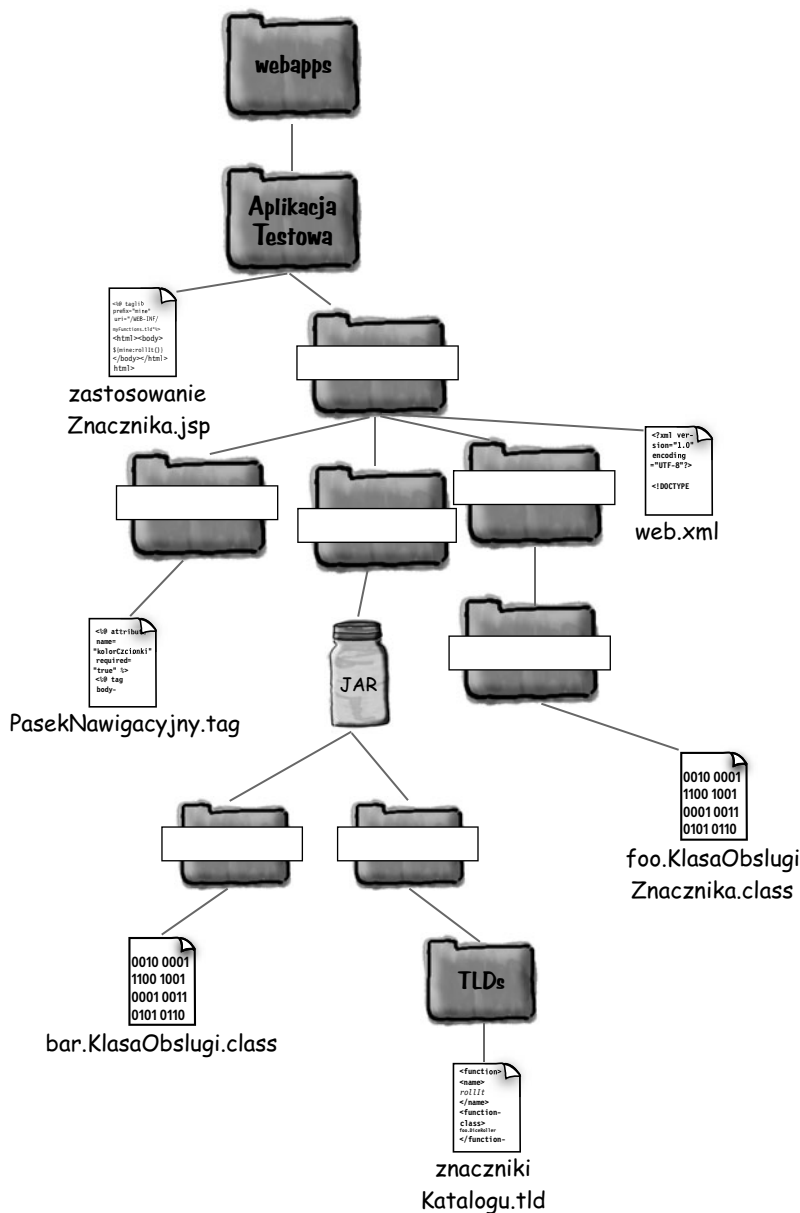
I jeszcze informacja dla osób podchodzących do egzaminu — Wy nie macie żadnego wyboru. Niemal wszystkie informacje podane w tym rozdziale mogą się pojawić na egzaminie.



## Zaostrz ołówek

## Podaj nazwy katalogów

Na poniższym rysunku podaj poprawne nazwy katalogów, opierając się na nazwach plików, które są w tych katalogach umieszczone. Wszystkie informacje niezbędne do podania poprawnych odpowiedzi zostały już przedstawione we wcześniejszych rozdziałach, jednak nie przejmuj się, jeśli wszystkiego nie zapamiętałeś. To właśnie w *niniejszym* rozdziale powinieneś sobie *utrwalić* tę wiedzę.



## Ćwiczenie z wdrażania aplikacji



### Zaostrz ołówek

### Narysuj strukturę katalogów i plików

Przeanalizuj zamieszczony poniżej opis aplikacji i narysuj strukturę katalogów, w których można ją umieścić. Nie zapomnij także o narysowaniu plików. Struktura aplikacji może mieć kilka różnych postaci; sugerujemy, żebyś wybrał najprostszą z nich (czyli zawierającą najmniejszą ilość katalogów).

**Nazwa aplikacji:** Randki

**Zawartość statyczna i strony JSP:** powitanie.html, logowanie.jsp, szukaj.jsp

**Serwlety:** randki.Rejestracja.class, randki.Szukaj.class

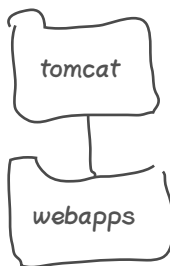
**Klasy obsługi znaczników niestandardowych:** klasyZnacznikow.ZnacznikPierwszy.class

**TLD:** ZnacznikiRandki.tld

**Komponenty JavaBean:** randki.Klient.class

**Deskryptor wdrożenia:** web.xml

**Pomocnicze pliki JAR:** RandkiJar.jar

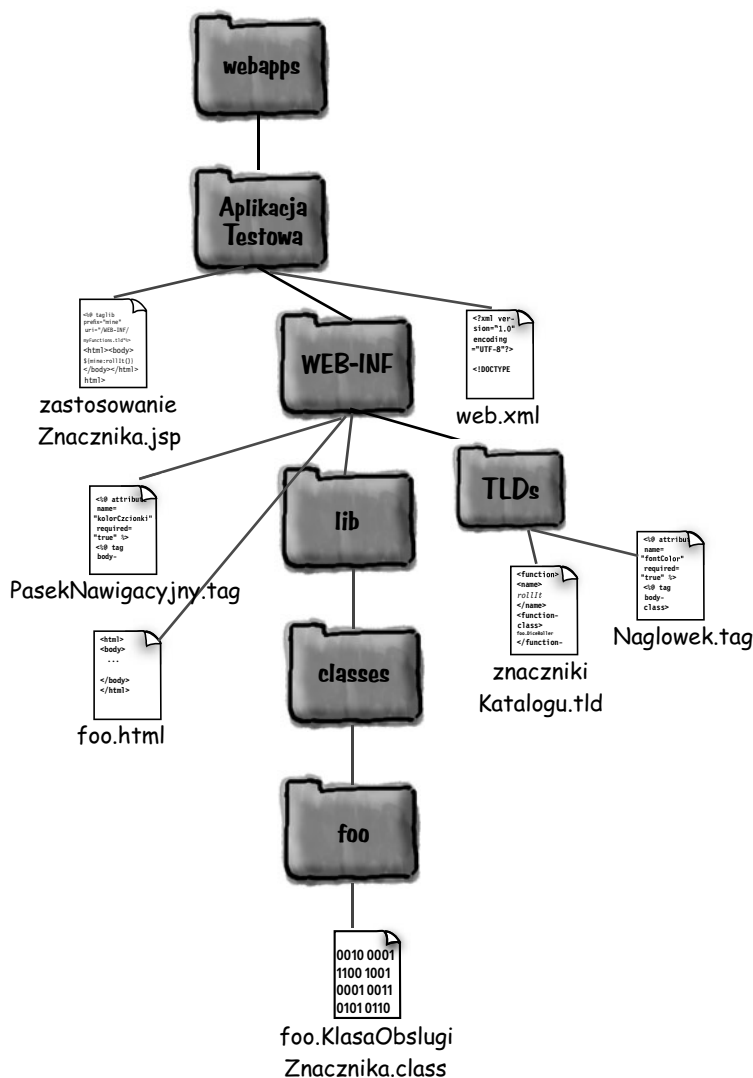


# BĄDŹ kontenerem



Struktura aplikacji przedstawiona na poniższym rysunku jest błędna. Dlaczego? Z kilku powodów, które sprawiają, że przedstawiona struktura nie odpowiada wymogom określonym w specyfikacji serwletów i JSP w zakresie katalogów dla poszczególnych składników aplikacji internetowej. Przyjmij, że wszystkie przedstawione pliki mają poprawne nazwy i rozszerzenia.

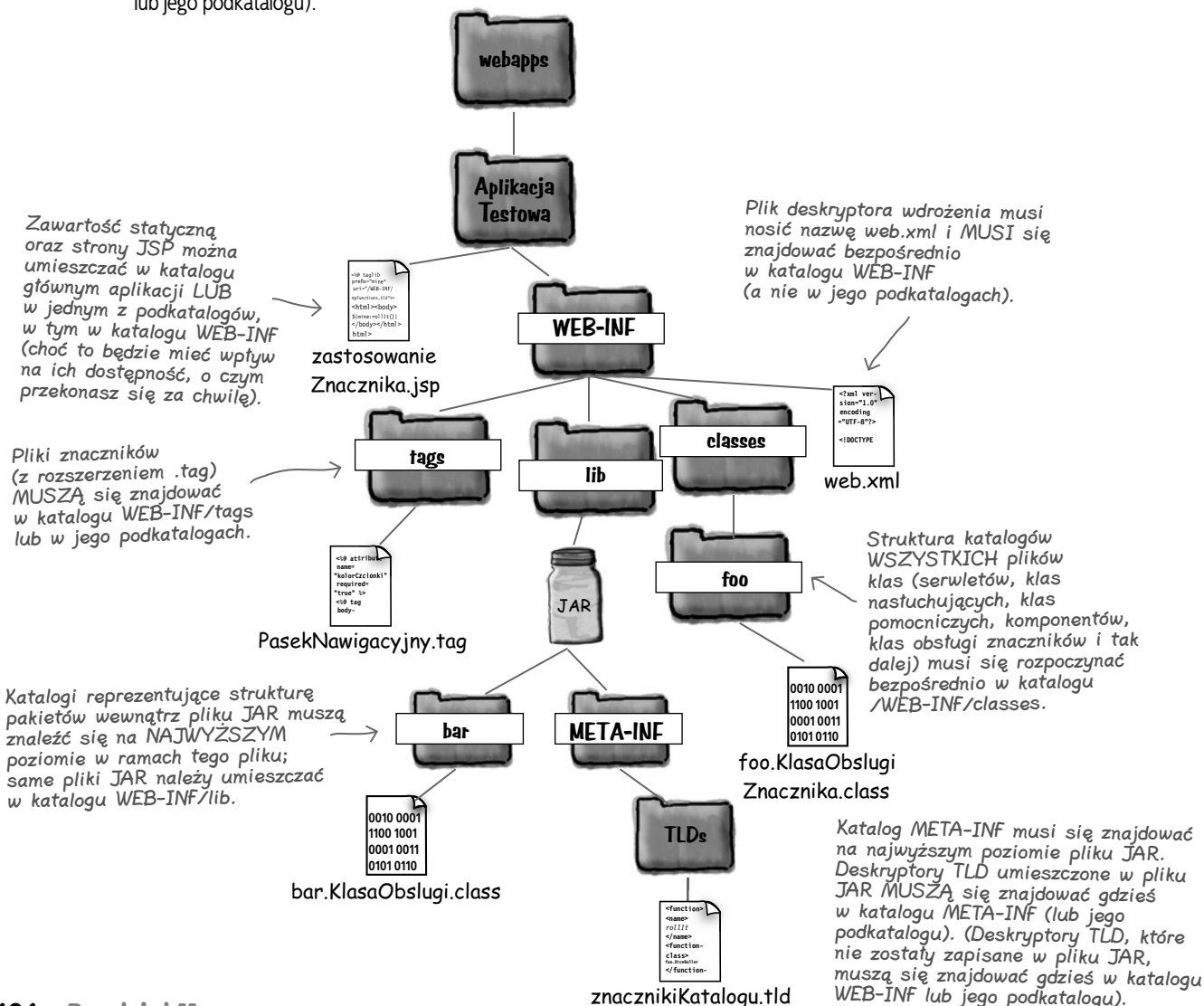
Podaj wszystkie nieprawidłowości struktury przedstawionej na rysunku:



## Zaostrz ołówek

### Podaj nazwy katalogów

Aby pomyślnie wdrożyć aplikację internetową na serwerze, MUSISZ nadać jej strukturę przedstawioną na poniższym rysunku. Katalog *WEB-INF* musi się znajdować bezpośrednio w kontekście aplikacji (w tym przypadku jest nim katalog *Aplikacja Testowa*). Katalog *classes* musi się znajdować bezpośrednio w katalogu *WEB-INF*. Katalogi tworzące strukturę pakietów muszą się znajdować bezpośrednio wewnątrz katalogu *classes*. Katalog *lib* musi się znajdować bezpośrednio w katalogu *WEB-INF*, a pliki JAR należy umieszczać bezpośrednio w katalogu *lib*. Katalog *META-INF* powinno się umieszczać na najwyższym poziomie w ramach plików JAR, a pliki TLD przechowywane w archiwum JAR należy umieszczać w katalogu *META-INF* lub w jednym z jego podkatalogów (przy czym nie ma wymogu, by katalog ten nosił nazwę *TLDs*). Pliki TLD, które nie są przechowywane w archiwum JAR, należy umieszczać gdzieś w katalogu *WEB-INF* lub jego podkatalogach. Pliki znaczników (z rozszerzeniami *.tag* lub *.tagx*) muszą być umieszczone gdzieś w katalogu *WEB-INF/tags* lub jego podkatalogach (chyba że są zapisane w pliku JAR — w takim przypadku muszą się znajdować w katalogu *META-INF/tags* lub jego podkatalogu).





## Zaostrz ołówek

ODPOWIEDZI

## Narysuj strukturę katalogów i plików

Jedyne zmiany, jakie można wprowadzić na tym rysunku, są: 1) zawartość statyczna oraz strony JSP mogą się znaleźć w podkatalogu umieszczonym wewnątrz katalogu *Randki* lub mogą zostać ukryte w katalogu *WEB-INF*, 2) plik *ZnacznikiRandki.tld* może być umieszczony w jednym z podkatalogów katalogu *WEB-INF*.

**Nazwa aplikacji:** Randki

**Zawartość statyczna i strony JSP:** powitanie.html, logowanie.jsp, szukaj.jsp

**Serwlety:** randki.Rejestracja.class, randki.Szukaj.class

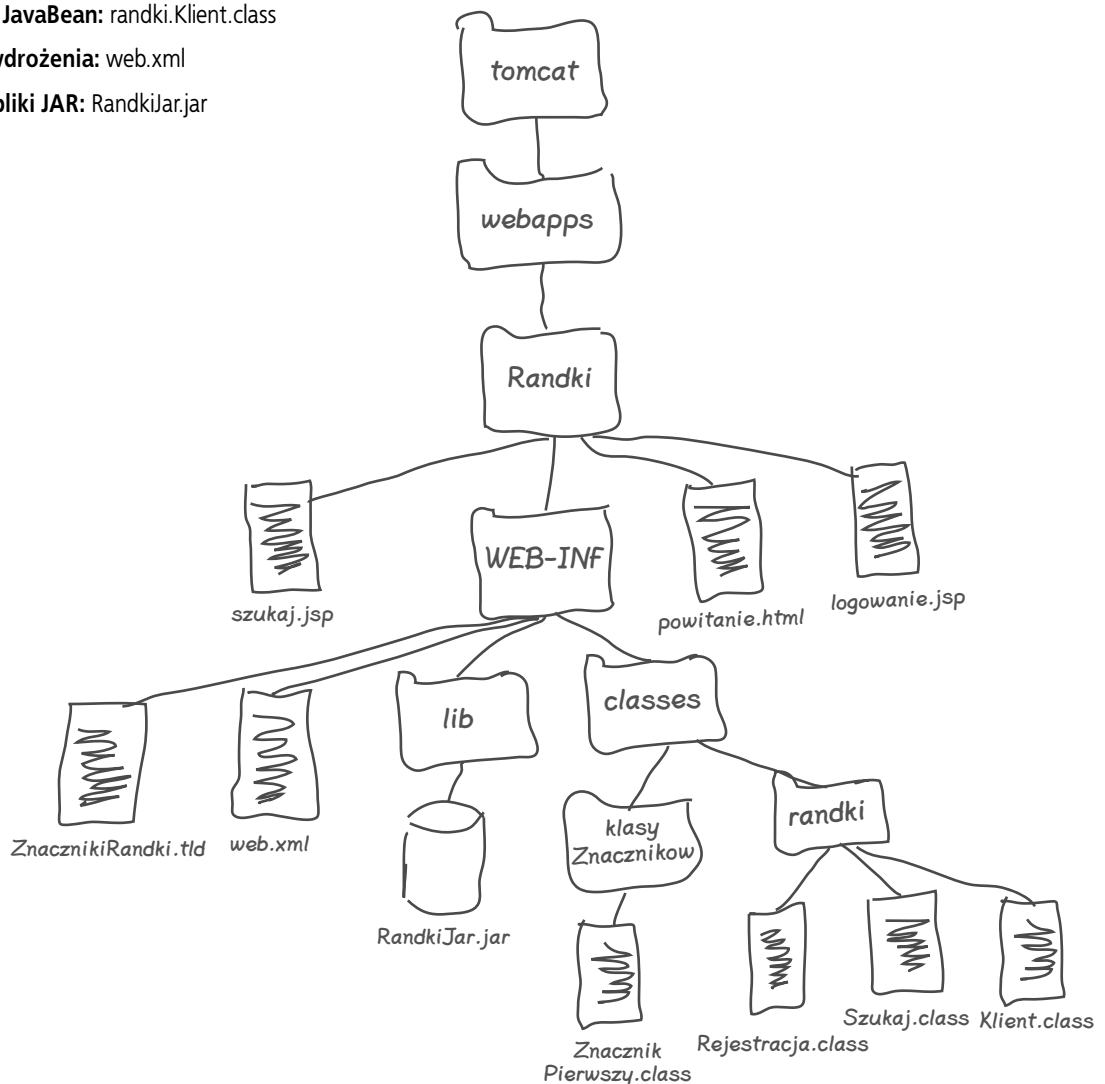
**Klasy obsługi znaczników niestandardowych:** klasyZnacznikow.ZnacznikPierwszy.class

**TLD:** ZnacznikiRandki.tld

**Komponenty JavaBean:** randki.Klient.class

**Deskryptor wdrożenia:** web.xml

**Pomocnicze pliki JAR:** RandkiJar.jar

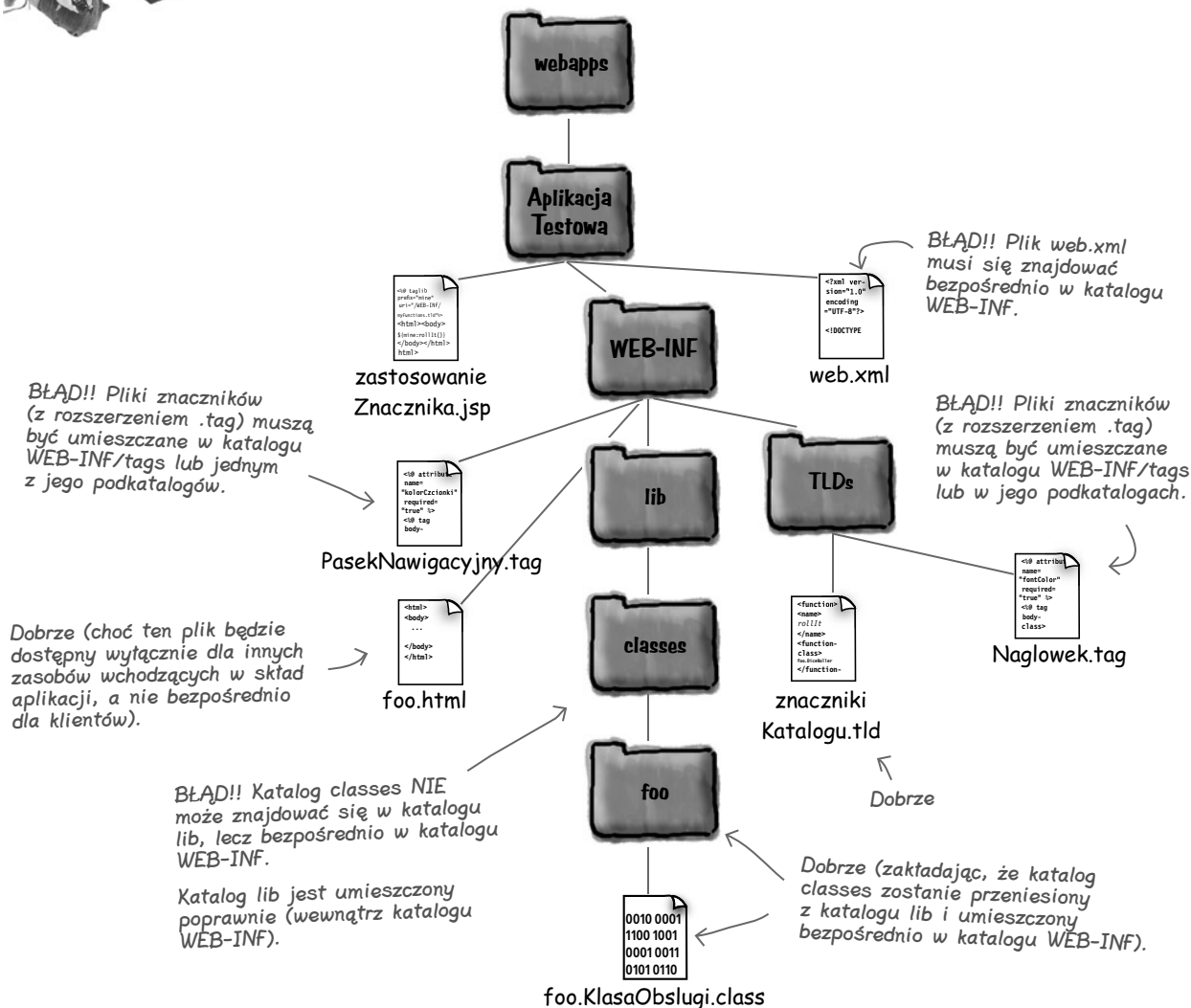




## BĄDŹ kontenerem

### Odpowiedzi

Struktura przedstawiona na poniższym rysunku zawiera kilka błędów!



Och... gdyby tylko istniała  
możliwość wdrażania całej aplikacji  
w pliku JAR, tak żebym mogła ją  
przenosić w jednym pliku,  
zamiast w formie całej masy plików  
i katalogów...



### Ona tak naprawdę chciałaby używać pliku WAR

Struktura katalogów aplikacji internetowych faktycznie jest rozbudowana, a wszystkie wchodzące w jej skład pliki muszą być umieszczane w ściśle określonych miejscach. Przenoszenie aplikacji internetowej może być bardzo uciążliwe.

Istnieje jednak pewne rozwiązanie — pliki WAR. Ich nazwa to skrót od słów **Web AR**chive (archiwum internetowe). Jeśli nazwa ta wydaje Ci się równie podejrzana co JAR (skrót od **Java AR**chive), to zapewne dlatego, iż plik WAR *jest* w rzeczywistości plikiem JAR. To po prostu plik JAR z innym rozszerzeniem — *.war*.

### Pliki WAR

Plik WAR jest kopią struktury aplikacji internetowej w miłej, przenośnej i skompresowanej formie (gdyż w rzeczywistości jest to plik JAR). Aby go utworzyć, należy spakować całą aplikację przy użyciu programu JAR (pomijając przy tym katalog kontekstu — czyli katalog *nadrzędny* katalogu *WEB-INF*) i zmienić rozszerzenie uzyskanego pliku na *.war*. Jednak takie rozwiązanie przysparza pewnego problemu — otóż jeśli w pliku WAR nie umieścimy katalogu aplikacji (na przykład *AplikacjaPiwna*), to w jaki sposób kontener będzie w stanie określić jej nazwę (kontekst)?

To zależy od używanego kontenera. **W przypadku Tomcata nazwą aplikacji staje się nazwa pliku WAR!** Załóżmy, że korzystamy z kontenera Tomcat i wdrażamy na nim aplikację w standardowej strukturze katalogów umieszczonej w katalogu *tomcat/webapps/AplikacjaPiwna*. Aby wdrożyć tę samą aplikację w formie pliku WAR, należy spakować wszystko, co znajduje się w katalogu *AplikacjaPiwna* (z pominięciem samego katalogu *AplikacjaPiwna*), a nazwę uzyskanego w ten sposób pliku zmienić na ***AplikacjaPiwna.war***. Następnie można umieścić plik *AplikacjaPiwna.war* w katalogu *tomcat/webapps*. I to wszystko. Tomcat rozpakuje plik WAR i utworzy katalog kontekstu, nadając mu nazwę pliku WAR. Pamiętaj jednak, że kontener, którego używasz, może obsługiwać wdrażanie plików WAR i określanie nazw aplikacji w inny sposób. Dla nas najważniejsze jest to, czego wymaga specyfikacja, a stwierdza ona, iż niemal nie ma żadnych różnic pomiędzy aplikacjami wdrażanymi w formie plików WAR oraz w formie struktury niezależnych katalogów i plików! Innymi słowy, w obu przypadkach konieczny będzie katalog *WEB-INF*, plik *web.xml* i tak dalej. Obowiązują wszystkie zasady, których znajomość sprawdziliśmy na kilku poprzednich stronach tego rozdziału.

**To niemal wszystko.** Jest jednak pewna rzecz, którą możesz zrobić jedynie w przypadku wdrażania aplikacji w formie pliku WAR. Chodzi o **zadeklarowanie zależności bibliotek**.

W ramach pliku WAR (w pliku *META-INF/MANIFEST.MF*) można dodatkowo zadeklarować wzajemne zależności pomiędzy bibliotekami, aby kontener mógł sprawdzić, czy w momencie *wdrażania aplikacji* dysponuje wszystkimi pakietami i klasami, od których zależy jej poprawne funkcjonowanie. Oznacza to, że nie musimy już czekać na otrzymanie żądania konkretnego zasobu, by przekonać się, że cała aplikacja nie działa wskutek braku jednej z niezbędnych klas.



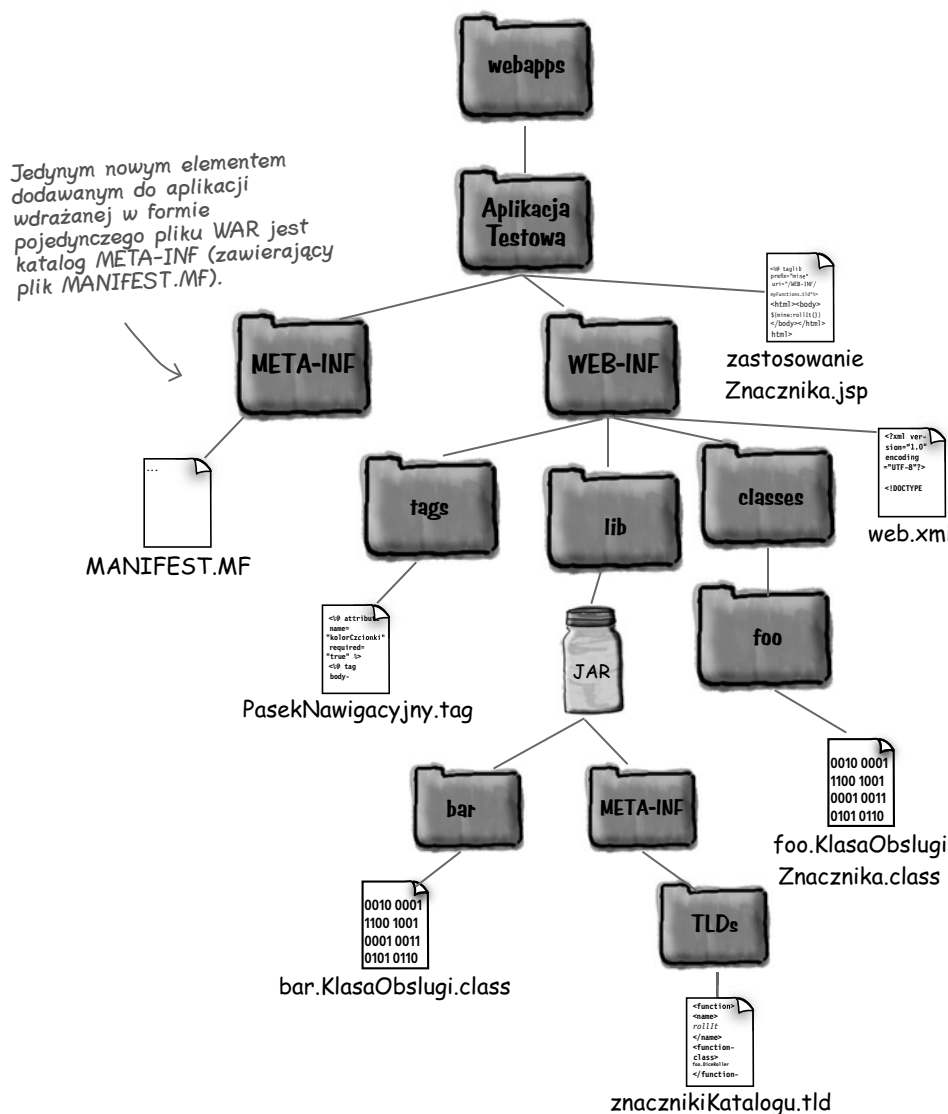
Oglądaj to!

**Nie daj się zaskoczyć pytaniom dotyczącym plików WAR...  
żadne reguły nie ulegają zmianom!**

Krótkie pytanie: Czy wciąż potrzebujemy pliku *web.xml*, jeśli aplikacja jest umieszczana na serwerze w formie pliku WAR? Oczywiście. Czy wciąż potrzebujemy katalogu *WEB-INF*, jeśli aplikacja jest wdrażana na serwerze w formie pliku WAR? Oczywiście. Czy pliki klas wciąż trzeba umieszczać w katalogu *classes* stanowiącym podkatalog katalogu *WEB-INF*? Oczywiście. Pewnie już rozumiesz. Sam fakt umieszczenia aplikacji w pliku WAR nie powoduje zmiany jakichkolwiek zasad! Jedyna poważna różnica polega na tym, że plik WAR będzie zawierać katalog *META-INF* (oprócz katalogu *WEB-INF*).

## Jak wygląda plik WAR umieszczony na serwerze?

Kiedy wdrażamy aplikację internetową w kontenerze Tomcat przez umieszczenie pliku WAR w katalogu *webapps*, nasz plik jest rozpakowywany, a jego zawartość jest umieszczana w nowym katalogu kontekstu (w przypadku aplikacji przedstawionej na poniższym rysunku będzie to katalog **Aplikacja Testowa**); jedynym nowym elementem aplikacji będzie katalog **META-INF** (zawierający plik **MANIFEST.MF**) umieszczony w głównym katalogu aplikacji. Prawdopodobnie nigdy nie będziesz samodzielnie umieszczał plików w tym katalogu, a zatem to, czy aplikacja została wdrożona przy wykorzystaniu pliku WAR, nie będzie miało dla Ciebie większego znaczenia; oczywiście jeśli nie będziesz musiał określać zależności bibliotek w pliku **MANIFEST.MF**.



## Bezpośrednie udostępnianie zawartości statycznej oraz stron JSP

Tworząc aplikację internetową, możesz zdecydować, czy wchodzące w jej skład statyczne pliki HTML oraz strony JSP będą dostępne bezpośrednio spoza tej aplikacji. Przez *dostęp bezpośredni* rozumiemy możliwość wpisania adresu zasobu w przeglądarce WWW i zwrócenia tego zasobu przez serwer. Można temu zapobiec, umieszczając te pliki w katalogu *WEB-INF* lub w katalogu *META-INF* (jeśli aplikacja jest wdrażana na serwerze w formie pliku WAR).

Ta ścieżka jest dostępna bezpośrednio w ramach aplikacji internetowej.

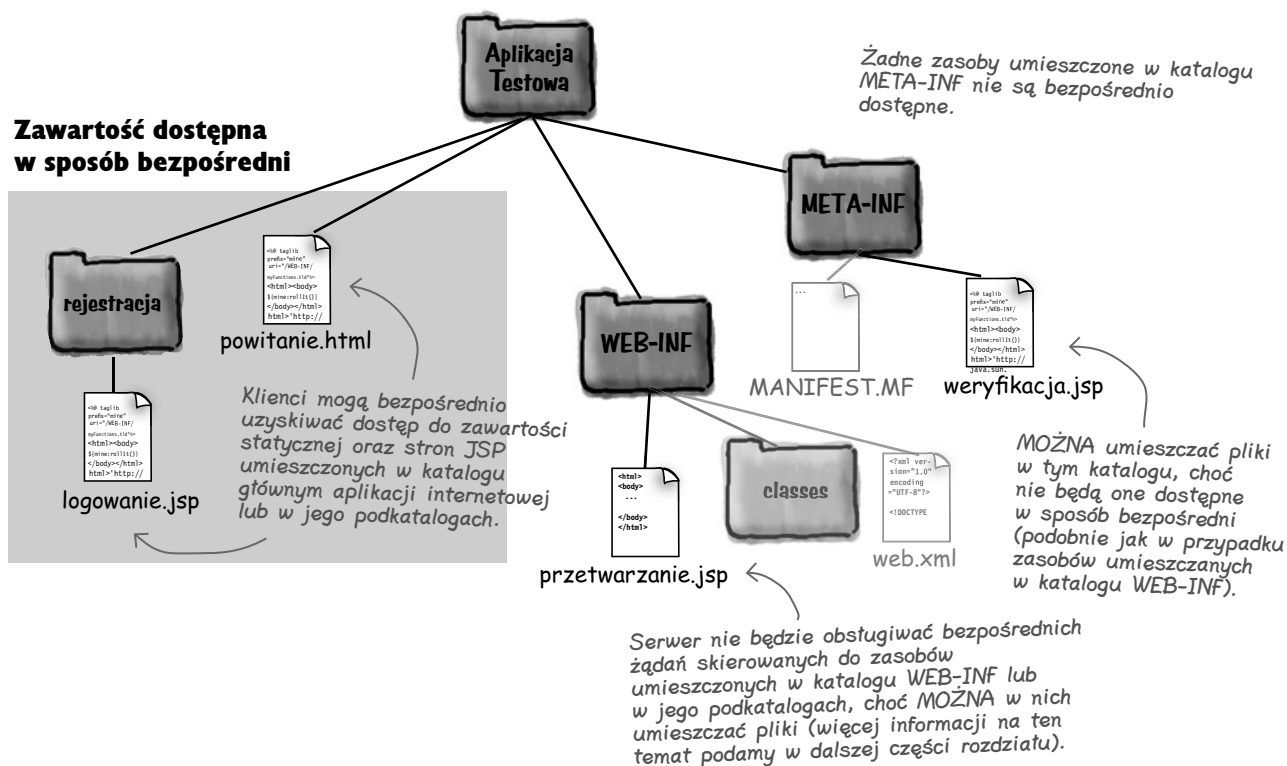
**Żądanie poprawne**

`http://www.aplikacja.com/AplikacjaTestowa/rejestracja/logowanie.jsp`

Nie! Nie ma możliwości bezpośredniego odwrotania się do zasobów umieszczonych w katalogu *WEB-INF*.

**Żądanie błędne (zwróci błąd numer 404 — plik nie został odnaleziony).**

`http://www.aplikacja.com/AplikacjaTestowa/WEB-INF/przetwarzanie.jsp`



**Jeśli serwer odbierze żądanie dotyczące jakiegokolwiek zasobu umieszczonego wewnątrz katalogów *WEB-INF* oraz *META-INF*, to kontener MUSI zwrócić odpowiedź 404 — NIE ZNALEZIONO.**

## Nie ma niemądrych pytań

**P. Skoro pliki umieszczone w katalogach WEB-INF i META-INF nie są udostępniane, to po co je w nich zapisywać??!**

**U.** Zastanów się. W Javie możesz tworzyć klasy i składowe z domyślnym poziomem dostępu (czyli w ramach pakietu), nieprawdaż? Choć takie klasy i składowe nie są dostępne „publicznie”, mogą być używane przez inne klasy i składowe, do których każdy może uzyskać dostęp. To samo dotyczy plików statycznych i stron JSP, do których nie można się odwoływać bezpośrednio. Umieszczając je w katalogu *WEB-INF* (lub *META-INF* w przypadku wdrażania aplikacji w postaci pliku WAR), chronimy te zasoby przed dostępem bezpośrednim, a jednocześnie zapewniamy innym elementom aplikacji możliwość ich wykorzystywania. Na przykład możemy zezwolić na dołączanie pewnego pliku lub przekierowanie żądania do niego, mając jednocześnie pewność, że żaden klient nie będzie mógł odwołać się do niego w sposób bezpośredni. Chcąc zabezpieczać zasoby przed dostępem bezpośrednim, będziesz je najprawdopodobniej umieszczał w katalogu *WEB-INF*, a nie *META-INF*, jednak przystępując do egzaminu, musisz wiedzieć, że oba te katalogi są traktowane w identyczny sposób.

**P. A co z katalogiem META-INF umieszczonym wewnątrz pliku JAR, który jest przechowywany w katalogu WEB-INF/lib? Czy jest on chroniony w taki sam sposób jak katalog META-INF w pliku WAR?**

**U.** Hm... tak. Jednak w tym przypadku fakt umieszczenia zawartości w katalogu *META-INF* nie ma kluczowego znaczenia. Liczy się to, iż plik JAR jest umieszczony w katalogu *lib*, który znajduje się w katalogu *WEB-INF*. A jak wiadomo, wszystkie zasoby umieszczone w katalogu *WEB-INF* są chronione i nie można się do nich odwoływać w sposób bezpośredni! A zatem nie ma większego znaczenia, w którym konkretnie miejscu znajduje się interesujący nas plik. Jeśli tylko jest przechowywany w katalogu *WEB-INF* lub w jednym z jego podkatalogów, to będzie chroniony. Stwierdzając, iż katalog *META-INF* jest chroniony, mamy na myśli katalog *META-INF* umieszczony w pliku WAR, gdyż katalogi o tych samych nazwach umieszczone w plikach JAR są chronione dlatego, że znajdują się w jednym z podkatalogów katalogu *WEB-INF* (*WEB-INF/lib*).

**P. Na jednej z poprzednich stron wspominaliście o umieszczaniu w pliku MANIFEST.MF informacji o wzajemnych zależnościach bibliotek. Czy podawanie tych informacji jest niezbędne? Czy wszystkie zasoby umieszczone w plikach JAR zapisanych w katalogu WEB-INF/lib oraz zasoby z katalogu WEB-INF/classes nie są automatycznie umieszczane na ścieżce klas danej aplikacji?**

**U.** Owszem, wszystkie klasy umieszczone w katalogu *WEB-INF/classes* oraz w plikach JAR umieszczonych w katalogu *WEB-INF/lib* są dostępne automatycznie, a korzystanie z nich nie wymaga żadnych

dotychczasowych operacji. One po prostu są dostępne. Może się jednak zdarzyć, że Twoja aplikacja będzie korzystała z pewnych dodatkowych pakietów dostępnych w ścieżce klas samego kontenera. Może być także tak, że aplikacja będzie wymagać zastosowania konkretnej wersji biblioteki. Plik *MANIFEST.MF* daje nam możliwość przekazania kontenerowi informacji o dodatkowych bibliotekach, do których musimy mieć dostęp. Jeśli kontener nie będzie w stanie ich udostępnić, to nie pozwoli na pomyślne zakończenie wdrażania aplikacji. Takie rozwiązanie jest o wiele lepsze od sytuacji, w której można rozmieścić aplikację, a dopiero potem, w chwili przesłania żądania, występują jakieś poważne (bądź, co jest znacznie gorsze — subtelne) błędy.

**P. W jaki sposób kontener uzyskuje dostęp do zawartości plików JAR umieszczonych w katalogu WEB-INF/lib?**

**U.** Kontener automatycznie umieszcza pliki JAR w swojej ścieżce klas, a zatem wszystkie klasy serwetów, komponentów itp. są dostępne dokładnie w taki sam sposób, jak gdyby zostały bezpośrednio umieszczone w strukturze podkatalogów katalogu *WEB-INF/classes*. Innymi słowy, to, czy klasy znajdują się w pliku JAR, nie ma znaczenia, o ile zostały one umieszczone w odpowiednich miejscach.

Pamiętaj jednak, że kontener zawsze będzie poszukiwał klas najpierw w katalogu *WEB-INF/classes*, a dopiero *potem* w plikach JAR.

**P. No dobrze, to wyjaśnia sprawy związane z plikami klasowymi, ale co z wszelkimi pozostałymi typami plików? Co mam zrobić, jeśli muszę skorzystać z pliku tekstowego zapisanego w archiwum JAR umieszczonym w katalogu WEB-INF/lib?**

**U.** To zupełnie inna sprawa. Jeśli kod działający w ramach aplikacji internetowej musi uzyskać dostęp do zasobów (plików tekstowych, obrazów JPEG itp.) umieszczonych w plikach JAR, konieczne jest zastosowanie metod `getResource()` lub `getResourceAsStream()` obiektu ładowania klas — są to zwyczajne metody dostępne także w J2SE.

Być może przypominasz sobie obie wspomniane wcześniej metody (`getResource()` oraz `getResourceAsStream()`), gdyż są one także dostępne w interfejsie `ServletContext`. Różnica pomiędzy nimi polega na tym, że metody dostępne w interfejsie `ServletContext` operują jedynie na zasobach, które co prawda wchodzą w skład aplikacji, ale nie zostały umieszczone w plikach JAR. (Podchodząc do egzaminu wystarczy, abyś wiedział, że możesz korzystać ze standardowych mechanizmów dostępnych w J2SE w celu pobierania zasobów przechowywanych w plikach JAR; nie musisz jednak znać żadnych dodatkowych szczegółów).

## Jak NAPRAWDĘ działają odwzorowania serwletów?

Widziałeś już przykłady odwzorowań serwletów zamieszczone w deskryptorach wdrożenia przedstawionych we wcześniejszych rozdziałach, w tym także „miniporadniku MVC” w rozdziale 3.

Każde odwzorowanie serwletu składa się z dwóch części — elementu `<servlet>` oraz elementu `<servlet-mapping>`. Pierwszy z nich określa nazwę oraz klasę serwletu, natomiast drugi — wzorec adresu URL odpowiadający nazwie serwletu zdefiniowanej w innym miejscu deskryptora wdrożenia.

```
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
 version="2.4">
```

```
<servlet>
```

```
 <servlet-name>Piwo</servlet-name>
```

```
 <servlet-class>com.przyklad.WyborPiwa</servlet-class>
```

```
</servlet>
```

```
<servlet-mapping>
```

```
 <servlet-name>Piwo</servlet-name>
```

```
 <url-pattern>/Piwo/WyborPiwa.do</url-pattern>
```

```
</servlet-mapping>
```

```
</web-app>
```

*Ta nazwa jest przeznaczona głównie do wykorzystania w innych fragmentach deskryptora wdrożenia. Klient NIE będzie o niej wiedział.*

*Kiedy zostanie odebrane żądanie w tej formie, kontener odnajdzie element `<servlet>` z odpowiednią wartością elementu `<servlet-name>` i na tej podstawie określi klasę odpowiedzialną za obsługę żądania.*

Jeśli zostanie odebrane żądanie zasobu `/Piwo/WyborPiwa.do`, ostatecznie trafi ono do serwletu nazwanego `Piwo`.



web.xml

A... widzę, że istnieje element `<servlet>` z odpowiednią wartością elementu `<servlet-name>` — `Piwo` — ten element zawiera informację o klasie, której należy użyć do obsługi żądania.



Kontener

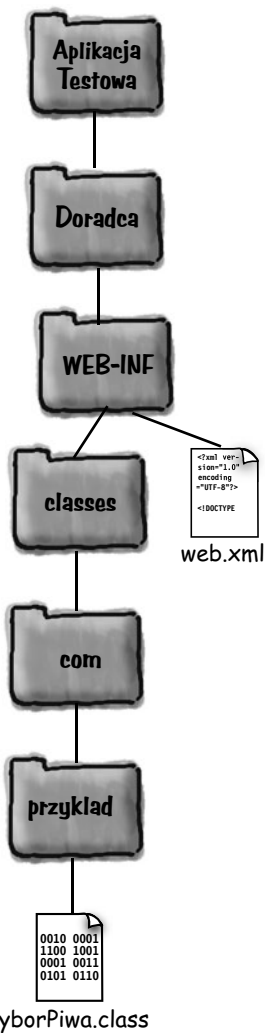
Ale nie widzę  
w strukturze aplikacji  
żadnego katalogu o nazwie  
Piwo ani pliku o nazwie  
WyborPiwa.do

To jest katalog główny  
(katalog kontekstu) tej  
aplikacji internetowej.

Ta ścieżka istnieje  
wyłącznie w deskrypcorze  
wdrożenia.

<http://www.aplikacja.com/Doradca/Piwo/WyborPiwa.do>

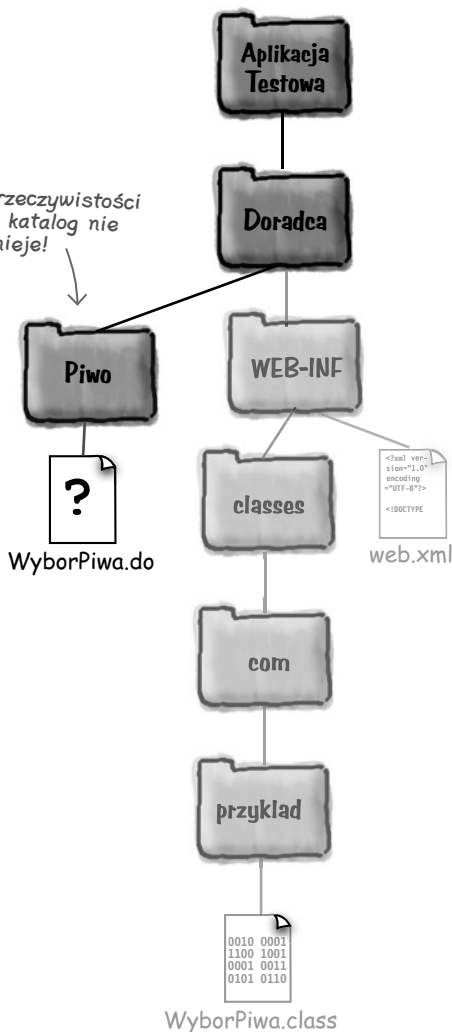
## FAKTYCZNA (fizyczna) struktura katalogów



To jest serwet  
obsługujący żądania  
kierowane do  
/Piwo/WyborPiwa.do

## WIRTUALNA (logiczna) struktura katalogów

W rzeczywistości  
ten katalog nie  
istnieje!



# Odwzorowania serwletów mogą „oszukiwać”

Wzorzec URL podawany w odwzorowaniu serwletu może być całkowicie dowolny. Wymyślony. Udawany. To tylko logiczna nazwa, którą mają się posługiwać klienci naszej aplikacji. Klienci nie muszą dysponować żadnymi informacjami na temat *rzeczywistej* fizycznej struktury aplikacji.

Odwzorowania serwletów powodują, że dysponujemy dwiema strukturami tej samej aplikacji — *rzeczywistą* strukturą fizycznych katalogów i plików, w których są umieszczane zasoby tworzące aplikację, oraz strukturą *wirtualną* (logiczną).

### TRZY typy elementów <url-pattern>

#### ① Dopasowanie DOKŁADNE

```
<url-pattern>/Piwo/WyborPiwa.do</url-pattern>
```

Wzorzec MUSI rozpoczynać się od ukośnika (/).  
Wzorzec może mieć rozszerzenie, choć nie jest to wymagane.

#### ② Dopasowanie KATALOGU

```
<url-pattern>/Piwo/*</url-pattern>
```

Wzorzec MUSI rozpoczynać się od ukośnika (/).  
To może być katalog rzeczywisty lub wirtualny.  
Na końcu zawsze należy umieścić znaki ukośnika i gwiazdki (/\*)

#### ③ Dopasowanie ROZSZERZENIA

```
<url-pattern>*.do</url-pattern>
```

Wzorzec MUSI zaczynać się od gwiazdki (\*) (w tym przypadku nie może zaczynać się od znaku ukośnika).  
Po znaku gwiazdki KONIECZNIE należy zapisać kropkę i rozszerzenie (.do, .jsp lub inne).

Struktura wirtualna (logiczna) istnieje tylko dlatego, iż TWIERDZISZ, że istnieje!

Wzorce URL podane w deskrypcorze wdrożenia odpowiadają jedynie elementom <servlet-name> podanym w tym samym deskrypcorze.

Kluczowe znaczenie dla odwzorowania serwletów mają elementy <servlet-name> — to one kojarzą wzorzec żądania (<url-pattern>) z faktyczną klasą serwletu.

Zagadnienie kluczowe: klienci odwołują się do serwletu, stosując żądanie w formie określonej w elemencie <url-pattern>, a NIE w elemencie <servlet-name> czy <servlet-class>!

## BĄDŹ kontenerem



Na podstawie odwzorowań serwetów zdefiniowanych w poniższym deskrypcorze wdrożenia oraz żądań przesyłanych przez klienta spróbuj określić, który z serwetów zostanie wybrany przez kontener. Takie same pytania pojawią się na prawdziwym egzaminie!

### Odwzorowania:

```
<servlet>
 <servlet-name>Pierwszy</servlet-name>
 <servlet-class>tmp.WdrozenieTestPierwszy</servlet-class>
</servlet>
<servlet-mapping>
 <servlet-name>Pierwszy</servlet-name>
 <url-pattern>*.do</url-pattern>
</servlet-mapping>

<servlet>
 <servlet-name>Drugi</servlet-name>
 <servlet-class>tmp.WdrozenieTestDrugi</servlet-class>
</servlet>
<servlet-mapping>
 <servlet-name>Drugi</servlet-name>
 <url-pattern>/tmpObsluga/bar</url-pattern>
</servlet-mapping>

<servlet>
 <servlet-name>Trzeci</servlet-name>
 <servlet-class>tmp.WdrozenieTestTrzeci</servlet-class>
</servlet>
<servlet-mapping>
 <servlet-name>Trzeci</servlet-name>
 <url-pattern>/tmpObsluga/*</url-pattern>
</servlet-mapping>
```

### Kluczowe reguły związane z odwzorowywaniem serwetów

- 1) Kontener poszukuje serwetów w kolejności przedstawionej na poprzedniej stronie. Innymi słowy, w pierwszej kolejności stara się znaleźć dopasowanie dokładne. Jeśli nie uda się znaleźć serwetu *dokładnie* pasującego do nadesłanego żądania, kontener stara się dopasować *katalog*, a jeśli i to się nie uda, spróbuje dopasować *rozszerzenie*.
- 2) Jeśli żądanie pasuje do większej liczby wzorców `<url-pattern>`, kontener wybierze najdłuższy z nich. Innymi słowy, nadesłane żądanie `/tmp/bar/mojeSprawy.do` zostanie skojarzone ze wzorcem `<url-pattern>` w formie `/tmp/bar/*`, nawet jeśli pasuje także do wzorca `/tmp/*`. Zawsze uwzględniane jest najbardziej *precyzyjne* dopasowanie.

### Żądania:

`http://localhost:8080/TestOdwzorowania/niebieski.do`

**Serwet wybrany przez kontener:**

`http://localhost:8080/TestOdwzorowania/tmpObsluga/bar`

**Serwet wybrany przez kontener:**

`http://localhost:8080/TestOdwzorowania/tmpObsluga/bar/niebieski.do`

**Serwet wybrany przez kontener:**

`http://localhost:8080/TestOdwzorowania/tmpObsluga/niebieski.do`

**Serwet wybrany przez kontener:**

`http://localhost:8080/TestOdwzorowania/fred/niebieski.do`

**Serwet wybrany przez kontener:**

`http://localhost:8080/TestOdwzorowania/tmpObsluga`

**Serwet wybrany przez kontener:**

`http://localhost:8080/TestOdwzorowania/tmpObsluga/bar/tmp.fo`

**Serwet wybrany przez kontener:**

`http://localhost:8080/TestOdwzorowania/fred/niebieski.fo`

**Serwet wybrany przez kontener:**



# BĄDŹ kontenerem

## Odpowiedzi

**Odwzorowania:**

```
<servlet>
 <servlet-name>Pierwszy</servlet-name>
 <servlet-class>tmp.WdrozenieTestPierwszy</servlet-class>
</servlet>
<servlet-mapping>
 <servlet-name>Pierwszy</servlet-name>
 <url-pattern>*.do</url-pattern>
</servlet-mapping>

<servlet>
 <servlet-name>Drugi</servlet-name>
 <servlet-class>tmp.WdrozenieTestDrugi</servlet-class>
</servlet>
<servlet-mapping>
 <servlet-name>Drugi</servlet-name>
 <url-pattern>/tmpObsluga/bar</url-pattern>
</servlet-mapping>

<servlet>
 <servlet-name>Trzeci</servlet-name>
 <servlet-class>tmp.WdrozenieTestTrzeci</servlet-class>
</servlet>
<servlet-mapping>
 <servlet-name>Trzeci</servlet-name>
 <url-pattern>/tmpObsluga/*</url-pattern>
</servlet-mapping>
```

Odpowiedzi na pytania zamieszczone na kolejnej stronie:

1) WdrozenieTestCzwarty 2) WdrozenieTestDrugi

**Żądania:**

- http://localhost:8080/TestOdwzorowania/niebieski.do  
**Serwlet wybrany przez kontener:** WdrozenieTestPierwszy  
(dopasowano wzorzec *rozszerzenia* \*.do)
- http://localhost:8080/TestOdwzorowania/tmpObsluga/bar  
**Serwlet wybrany przez kontener:** WdrozenieTestDrugi  
(dopasowanie *dokładne* ze wzorcem /tmpObsluga/bar)
- http://localhost:8080/TestOdwzorowania/tmpObsluga/bar/niebieski.do  
**Serwlet wybrany przez kontener:** WdrozenieTestTrzeci  
(dopasowano wzorzec  *katalogu* /tmpObsluga/\*)
- http://localhost:8080/TestOdwzorowania/tmpObsluga/niebieski.do  
**Serwlet wybrany przez kontener:** WdrozenieTestTrzeci  
(dopasowano wzorzec  *katalogu* /tmpObsluga/\*)
- http://localhost:8080/TestOdwzorowania/fred/niebieski.do  
**Serwlet wybrany przez kontener:** WdrozenieTestPierwszy  
(dopasowano wzorzec *rozszerzenia* \*.do)
- http://localhost:8080/TestOdwzorowania/tmpObsluga  
**Serwlet wybrany przez kontener:** WdrozenieTestTrzeci  
(dopasowano wzorzec  *katalogu* /tmpObsluga/\*)
- http://localhost:8080/TestOdwzorowania/tmpObsluga/bar/tmp.fo  
**Serwlet wybrany przez kontener:** WdrozenieTestTrzeci  
(dopasowano wzorzec  *katalogu* /tmpObsluga/\*)
- http://localhost:8080/TestOdwzorowania/fred/niebieski.fo  
**Serwlet wybrany przez kontener:** Błąd 404 — nie znaleziono pliku. (nie udało się dopasować wzorca)

## Drobne szczegóły...

Poniżej zamieściliśmy kolejne proste ćwiczenie, którego celem jest sprawdzenie, czy rozumiesz zasady odwzorowywania serwletów. Nie pomijaj go — przyjrzyj się dokładnie obu odwzorowaniom i żądaniom przedstawionym poniżej nich. (Odpowiedzi na te pytania możesz znaleźć u dołu poprzedniej strony, zatem nie podglądaj).



### BĄDŹ kontenerem

#### Odpowiedzi

#### Odwzorowania w deskrypcji wdrożenia:

```
<servlet>
 <servlet-name>Drugi</servlet-name>
 <servlet-class>tmp.WdrozenieTestDrugi</servlet-class>
</servlet>
<servlet-mapping>
 <servlet-name>Drugi</servlet-name>
 <url-pattern>/tmpObsluga/bar</url-pattern>
</servlet-mapping>

<servlet>
 <servlet-name>Czwarty</servlet-name>
 <servlet-class>tmp.WdrozenieTestCzwarty</servlet-class>
</servlet>
<servlet-mapping>
 <servlet-name>Czwarty</servlet-name>
 <url-pattern>/tmpObsluga/bar/*</url-pattern>
</servlet-mapping>
```

#### Żądania:

- ① <http://localhost:8080/test/tmpObsluga/bar/>  
*Serwlet wybrany przez kontener:*
- ② <http://localhost:8080/test/tmpObsluga/bar>  
*Serwlet wybrany przez kontener:*

## Konfiguracja plików powitalnych w deskrytorze wdrożenia

Wiesz już, że jeśli wpiszesz w przeglądarce sam adres witryny bez podawania nazwy konkretnego pliku, to i tak (zazwyczaj) zostanie zwrócona jakaś strona. Wpisanie w przeglądarce adresu `http://www.oreilly.com` spowoduje wyświetlenie witryny wydawnictwa O'Reilly i nawet jeśli nie wskażesz przy tym żadnego konkretnego pliku (takiego jak `home.html`), i tak zostanie zwrócona strona *domyślna*.

Można tak skonfigurować serwer, aby dla całej *witryny* istniała jedna strona domyślna, jednak nas interesują przede wszystkim strony domyślne (nazywane także stronami „powitalnymi”) dla poszczególnych *aplikacji internetowych*. Strony te konfiguruje się w deskrytorze wdrożenia i to właśnie deskryptor określa, jaką stronę kontener wyświetli w sytuacji, gdy użytkownik wpisze w przeglądarce *niepełny* adres URL — czyli adres, który, na przykład, określa katalogi, lecz nie wskazuje konkretnego pliku.

Innymi słowy, co się stanie, jeśli zostanie odebrane żądanie:

`http://www.aplikacja.com/tmp/bar` ← bar to nazwa katalogu

a *bar* to jedynie nazwa katalogu i nie ma żadnego serwletu, który byłby skojarzony z tym wzorcem adresu URL? Co zostanie wyświetlone w przeglądarce użytkownika?

### W deskrytorze wdrożenia:

```
<web-app ...>
 <welcome-file-list>
 <welcome-file>index.html</welcome-file>
 <welcome-file>default.jsp</welcome-file>
 </welcome-file-list>
</web-app>
```

Nazwy plików **NIE** mogą się zaczynać od znaku ukośnika!

Wyobraź sobie, że dysponujemy aplikacją internetową, w której kilka różnych podkatalogów dysponuje swoimi własnymi domyślnymi stronami HTML o nazwach `index.html`. Jednak w *niektórych* katalogach domyślne strony mają nazwę `default.jsp`. Określanie domyślnych stron HTML lub JSP dla każdego katalogu, który ich potrzebuje, byłoby niezwykle uciążliwe. Zamiast tego można podać listę plików, których kontener będzie, w określonej kolejności, poszukiwać we wszystkich katalogach, do których będzie się odnosiło odebrane żądanie częściowe. Innymi słowy, niezależnie od tego, którego katalogu będzie dotyczyć żądanie, kontener zawsze będzie poszukiwać plików podanych na tej samej liście — liście zdefiniowanej przy użyciu elementu `<welcome-file-list>`.

Kontener wybierze *pierwszy* plik odnaleziony na liście, zaczynając poszukiwania od pierwszego z plików powitalnych wskazanych w znaczniku `<welcome-file-list>`.



Oglądaj to!

**W jednym elemencie deskryptora wdrożenia można zdefiniować wiele różnych plików powitalnych.**

Niezależnie od tego, ile plików powitalnych chcesz zdefiniować, należy je wszystkie podać w jednym elemencie deskryptora wdrożenia — `<welcome-file-list>`. Może Cię kusić, by każdy z plików powitalnych umieścić w osobnym elemencie `<welcome-file>`, jednak takie rozwiązanie jest błędne! Każdy plik powitalny jest definiowany w osobnym elemencie `<welcome-file>`, lecz wszystkie te elementy są umieszczane w jednym elemencie `<welcome-file-list>`.



Oglądaj to!

**Nazwy plików podawane w elemencie `<welcome-file>` NIE mogą zaczynać się od znaku ukośnika!**

Nie daj się zaskoczyć. Sposób, w jaki kontener dopasowuje i wybiera pliki powitalne, jest nieco inny od sposobu dopasowywania wzorców URL. Jeśli przed nazwą pliku umieścisz ukośnik, to będzie to niezgodne z założeniami specyfikacji i może doprowadzić do problemów.

## BĄDŹ kontenerem

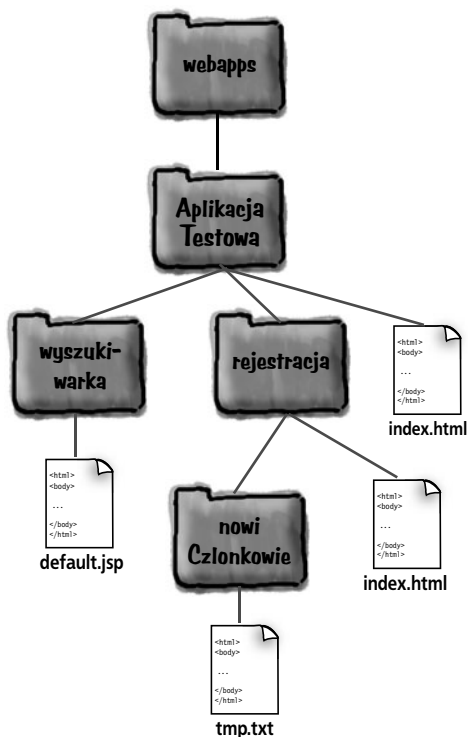


Na podstawie przedstawionego poniżej deskryptora wdrożenia i żądań spróbuj określić, które pliki powitalne zostaną wybrane przez kontener. Możesz się spodziewać, że na prawdziwym egzaminie pojawi się pytanie tego typu.

### Fragment deskryptora wdrożenia:

```
<welcome-file-list>
 <welcome-file>index.html</welcome-file>
 <welcome-file>default.jsp</welcome-file>
</welcome-file-list>
```

### Struktura katalogów aplikacji:



### Żądania:

<http://localhost:8080/AplikacjaTestowa/>

**Plik wybrany przez kontener:**

<http://localhost:8080/AplikacjaTestowa/rejestracja/>

**Plik wybrany przez kontener:**

<http://localhost:8080/AplikacjaTestowa/wyszukiwarka>

**Plik wybrany przez kontener:**

<http://localhost:8080/AplikacjaTestowa/rejestracja/nowiCzlonkowie/>

**Plik wybrany przez kontener:**



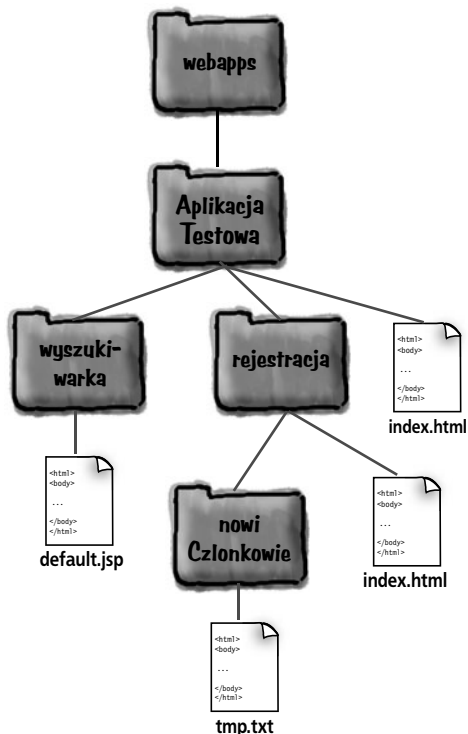
# BĄDŹ kontenerem

## Odpowiedzi

### Fragment deskryptora wdrożenia:

```
<welcome-file-list>
 <welcome-file>index.html</welcome-file>
 <welcome-file>default.jsp</welcome-file>
</welcome-file-list>
```

### Struktura katalogów aplikacji:



### Żądania:

<http://localhost:8080/AplikacjaTestowa/>

#### *Plik wybrany przez kontener:*

AplikacjaTestowa/index.html

<http://localhost:8080/AplikacjaTestowa/rejestracja/>

#### *Plik wybrany przez kontener:*

AplikacjaTestowa/rejestracja/index.html

<http://localhost:8080/AplikacjaTestowa/wyszukiwarka>

#### *Plik wybrany przez kontener:*

AplikacjaTestowa/wyszukiwarka/default.jsp

(Gdyby w katalogu wyszukiwarka oprócz pliku default.jsp znajdował się także plik index.html, to kontener wybrałby index.html, gdyż na liście w deskrypcorze wdrożenia jest on podany jako pierwszy).

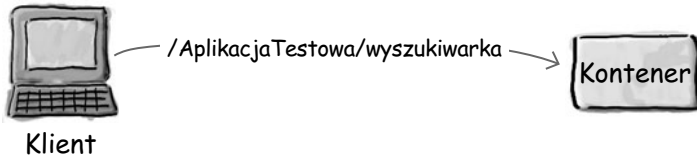
<http://localhost:8080/AplikacjaTestowa/rejestracja/nowiCzlonkowie/>

#### *Plik wybrany przez kontener:*

Jeśli dla danego katalogu nie zdefiniowano żadnego pliku powitalnego w elemencie `<welcome-file-list>` deskryptora wdrożenia, sposób postępowania kontenera zależy od jego twórców. Na przykład Tomcat w takim przypadku pokazuje listę zawartości katalogu (zawierającą jeden plik — „tmp.txt”), inne kontenery mogą natomiast zwracać błąd nr 404 — nie znaleziono pliku.

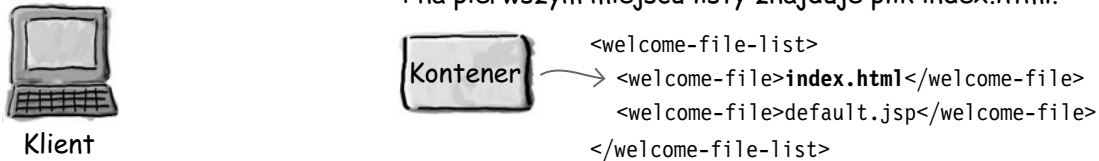
## W jaki sposób kontener wybiera plik powitalny?

- ① Klient przesyła żądanie: `http://www.aplikacja.com/AplikacjaTestowa/wyszukiwarka`



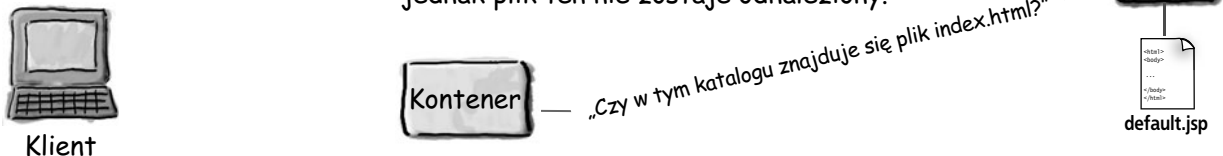
②

Kontener przegląda deskryptor wdrożenia w poszukiwaniu odwzorowania serwletu, jednak żadnego nie znajduje. Następnie analizuje zawartość elementu `<welcome-file-list>` i na pierwszym miejscu listy znajduje plik `index.html`.



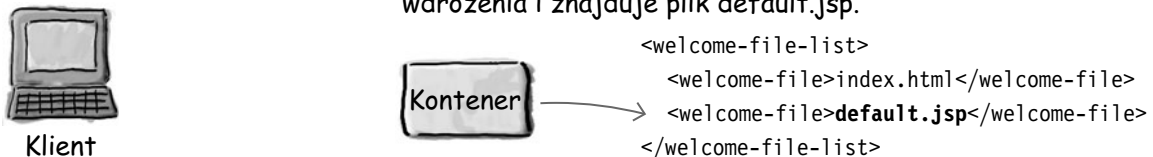
③

Kontener sprawdza, czy w katalogu `/AplikacjaTestowa/wyszukiwarka` znajduje się plik `index.html`, jednak plik ten nie zostaje odnaleziony.



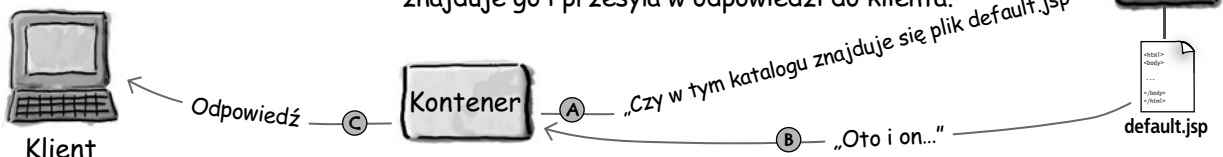
④

Kontener sprawdza kolejny element `<welcome-file>` umieszczony na liście `<welcome-file-list>` w deskryptorze wdrożenia i znajduje plik `default.jsp`.



⑤

Kontener przegląda katalog `/AplikacjaTestowa/wyszukiwarka` w poszukiwaniu pliku `default.jsp`, znajduje go i przesyła w odpowiedzi do klienta.



# Konfiguracja stron błędów w deskrypcji wdrożenia

Zapewne chcesz zapewnić możliwie przyjazną atmosferę tym użytkownikom odwiedzającym Twoją aplikację internetową, którzy nie znają dokładnego adresu interesującej ich strony, stąd decyzja o skonfigurowaniu plików powitalnych. Jednak z pewnością chcesz zapewnić komfort użytkownikom swojej aplikacji także wtedy, gdy coś *pójdzie nie tak, jak trzeba*. Zagadnieniem tym zajmowaliśmy się już wcześniej, w rozdziale poświęconym tworzeniu znaczników niestandardowych, zatem informacje zamieszczone na tej stronie są jedynie przypomnieniem.

## Deklarowanie strony wyświetlanej w przypadku wystąpienia jakichkolwiek problemów

Strona zadeklarowana w poniższy sposób będzie stosowana w przypadku wystąpienia jakichkolwiek problemów, a nie tylko problemów związanych ze stronami JSP:

```
<error-page>
 <exception-type>java.lang.Throwable</exception-type>
 <location>/stronaBledu.jsp</location>
</error-page>
```

(Dla Twojej informacji: Na stronie JSP można przesłonić tę deklarację, umieszczając na początku strony dyrektywę `page` z atrybutem `errorPage`).

## Deklarowanie strony błędu dla bardziej konkretnego wyjątku

Poniższy fragment deskryptora wdrożenia konfiguruje stronę błędu, która będzie wyświetlana tylko w przypadku zgłoszenia wyjątku `ArithmeticException`. Jeśli w deskrypcji pojawi się ta deklaracja, jak i deklaracja „ogólna” przedstawiona na poprzednim przykładzie, to za każdym razem, gdy zostanie zgłoszony wyjątek inny niż `ArithmeticException`, wciąż będzie wyświetlana strona `stronaBledu.jsp`.

```
<error-page>
 <exception-type>java.lang.ArithmeticException</exception-type>
 <location>/bladArytmetyczny.jsp</location>
</error-page>
```

## Deklarowanie strony błędu na podstawie kodu statusu HTTP

Poniższy fragment deskryptora wdrożenia konfiguruje stronę błędu, która będzie wyświetlana tylko w przypadkach, gdy kod statusu odpowiedzi przyjmie wartość 404 (nie znaleziono pliku).

```
<error-page>
 <error-code>404</error-code>
 <location>/nieZnalezionoPliku.jsp</location>
</error-page>
```



Oglądaj to!

**Nie można jednocześnie stosować elementów `<error-code>` i `<exception-type>`!**

*Można skonfigurować stronę błędu, która będzie wyświetlana bądź to na podstawie kodu statusu protokołu HTTP, BĄDŹ na podstawie typu zgłoszonego wyjątku. Jednak w tym samym elemencie `<error-page>` NIE MOŻNA jednocześnie wykorzystać obu tych możliwości.*

## Nie ma niemądrych pytań

**P. Jakie typy wyjątków można deklarować w elemencie `<exception-type>`?**

**O.** Dowolne wyjątki dziedziczące po `Throwable`. A zatem mogą to być wyjątki typu `java.lang.Error`, wyjątki czasu wykonywania programu oraz wszelkie tzw. wyjątki kontrolowane (o ile oczywiście klasa wyjątku znajduje się na ścieżce klas kontenera).

**P. Skoro już rozmawiamy na temat obsługi błędów, to czy można samodzielnie generować kod statusu?**

**O.** Tak, można. Można wywołać w tym celu metodę `sendError()` obiektu typu `HttpServletResponse`. Informuje ona kontener, iż należy wygenerować błąd, a jej efekty niczym się nie różnią od sytuacji, w której kontener wygenerowałby błąd samodzielnie. A jeśli została skonfigurowana strona błędu wyświetlana w przypadku pojawienia się tego błędu, to właśnie ona będzie wyświetlona w przeglądarce. Warto także wiedzieć, iż kody „błędów” są także nazywane kodami „statusu”, zatem oba te określenia oznaczają to samo — kody protokołu HTTP informujące o błędach.

**P. A może jakiś przykład generacji własnego kodu błędu?**

**O.** Dobrze, oto przykład:

```
response.sendError(HttpServletResponse.SC_FORBIDDEN);
```

co jest równoznaczne wywołaniu:

```
response.sendError(403);
```

Jeśli przejrzysz interfejs `HttpServletResponse`, zauważysz w nim grupę stałych definiujących kody błędów (statusu) protokołu HTTP. Pamiętaj, że podchodząc do egzaminu, nie musisz znać tych kodów na pamięć! Wystarczy, abyś wiedział, że możesz wygenerować kod błędu, że używana jest w tym celu metoda **`response.sendError()`** oraz że z punktu widzenia stron błędów definiowanych w deskrypcji wdrożenia, jak i wszelkich pozostałych operacji związanych z obsługą błędów, nie ma żadnej różnicy pomiędzy błędami HTTP wygenerowanymi programowo oraz wygenerowanymi przez kontener. Błąd 403 jest błędem 403 niezależnie od tego, KTO go wygeneruje. A tak... istnieje jeszcze przeciążona, dwuargumentowa wersja metody `sendError()` umożliwiająca podanie nie tylko kodu błędu, ale także komunikatu (w formie łańcucha znaków).



Oglądaj to!

**W elemencie `<exception-type>` musisz podawać w pełni kwalifikowane nazwy klas!**

Nie daj się zwieść takim przykładom jak:

```
<exception-type>
```

```
IOException
```

```
</exception-type>
```

W elemencie tym **MUSISZ** podawać w pełni kwalifikowane nazwy klas, przy czym mogą to być dowolne klasy dziedziczące po `Throwable`.

# Konfigurowanie inicjalizacji serwletu w deskrypcorze wdrożenia

Wiesz już, że domyślnie serwlety są inicjalizowane podczas obsługi pierwszego żądania. Oznacza to, że pierwszy klient, który odwoła się do serwletu, jest narażony na dodatkowe opóźnienia związane z ładowaniem klas, tworzeniem niezbędnych obiektów, ich inicjalizacją (określeniem kontekstu — obiektu `ServletContext`, wywołaniem procedur obsługi zdarzeń itp.). Kontener będzie musiał wykonać te wszystkie operacje, zanim zajmie się tym, co robi zazwyczaj — przydzieleniem wątku i wywołaniem metody `service()` serwletu.

Jeśli chcesz, aby serwlet nie był ładowany podczas pierwszego żądania, lecz w trakcie wdrażania aplikacji (lub w chwili jej ponownego uruchamiania), to należy umieścić w deskrypcorze wdrożenia element `<load-on-startup>`. Jakkolwiek wartość większa od zera umieszczona w tym elemencie poinformuje kontener, że dany serwlet należy załadować w trakcie wdrażania aplikacji (lub podczas każdego ponownego uruchamiania serwera).

Jeśli istnieje więcej serwletów, które chcesz zawczasu załadować, a jednocześnie zależy Ci na kontroli kolejności ich inicjalizacji, możesz ją określić za pomocą wartości definiowanych właśnie w elemencie `<load-on-startup>`. Innymi słowy, jeśli w tym elemencie zostanie podana wartość większa od zera, będzie to oznaczało, że dany serwlet należy załadować podczas wdrażania, a kolejność ładowania i inicjalizacji poszczególnych serwletów będzie zależała od wartości elementu `<load-on-startup>`.

## W deskrypcorze wdrożenia

```
<servlet>
 <servlet-name>Jeden</servlet-name>
 <servlet-class>tmp.WdrozenieTestPierwszy</servlet-class>
 <load-on-startup>1</load-on-startup>
</servlet>
```

↑ Jakkolwiek wartość większa od zera oznacza: „zainicjalizuj ten serwlet podczas wdrażania aplikacji, a nie czekaj z tym do momentu obsługi pierwszego żądania”.

**❓ Czy nie będziemy chcieli ZAWSZE ładować serwletów zawczasu? Czy nie należy po prostu domyślnie używać elementu `<load-on-startup>1</load-on-startup>`?**

**U.** Aby znaleźć odpowiedź na to pytanie, należy się zastanowić, ile serwletów wchodzi w skład naszej aplikacji i jakie jest prawdopodobieństwo, że wszystkie one będą używane. Musisz także odpowiedzieć sobie na pytanie, ile czasu wymaga załadowanie serwletu. Niektóre z serwletów są stosowane sporadycznie i w takich przypadkach lepszym rozwiązaniem może być zaoszczędzenie zasobów poprzez rezygnację z wcześniejszego ładowania serwletu; z drugiej strony, ładowanie niektórych serwletów jest tak kosztowne (dotyczy to na przykład serwletu `ActionServlet` szkieletu Struts), że nie warto narażać ani jednego klienta na podobne opóźnienia. A zatem tylko Ty możesz zdecydować, które serwlety należy ładować podczas wdrażania aplikacji, i zapewne zrobisz to, analizując poszczególne serwlety i określając czas ich ładowania i prawdopodobieństwo zastosowania.



Oglądaj to!

**Podanie wartości większej od 0 nie ma wpływu na liczbę tworzonych serwletów.**

Zastosowanie elementu `<load-on-startup>4</load-on-startup>` NIE oznacza wcale: „załaduj cztery serwlety tego typu”. Element ten sugeruje, że dany serwlet należy załadować DOPIERO po załadowaniu wszystkich serwletów, dla których w deskrypcorze wdrożenia podano elementy `<load-on-startup>` o wartości mniejszej od 4. Warto się też zastanowić, co się stanie, jeśli elementy `<load-on-startup>` dla kilku serwletów będą definiowały tę samą wartość. Otóż kontener ładuje serwlety o tej samej wartości elementu `<load-on-startup>` w kolejności, w jakiej zostały zadeklarowane w deskrypcorze wdrożenia.

# Tworzenie stron JSP zgodnych z zasadami konstrukcji dokumentów XML — dokumenty JSP

Trudno było znaleźć w tej książce miejsce właściwe dla omówienia tego zagadnienia, stąd decyzja o jego analizie tutaj, przy okazji opisywania rozmaitych dokumentów w formacie XML. Egzamin nie wymaga, abyś był ekspertem w dziedzinie stosowania języka XML, musisz jednak wykazać się znajomością dwóch zagadnień — składni kluczowych elementów deskryptora wdrożenia oraz podstawowych zasad tworzenia, tak zwanych *dokumentów JSP*. („Czyli czego? Czy w takim razie zwyczajne strony JSP nie są dokumentami?” — zapewne właśnie takie pytania przeszły Ci przez głowę. Wyobraź to sobie w następujący sposób — wszystko jest *stroną* JSP, chyba że zawiera alternatywną składnię JSP i w takim przypadku staje się *dokumentem*).

Oznacza to, że w rzeczywistości istnieją *dwa* typy składni, której możesz używać do zapisywania kodu JSP. Tekst, który w poniższej tabeli został oznaczony kolorem szarym, jest identyczny niezależnie od używanego rodzaju składni JSP.

Normalna składnia stron JSP		Składnia dokumentów JSP
Dyrektywy (oprócz taglib)	<code>&lt;%@ page import="java.util.*" %&gt;</code>	<code>&lt;jsp:directive.page import="java.uitl.*" /&gt;</code>
Deklaracje	<code>&lt;%! int y = 3; %&gt;</code>	<code>&lt;jsp:declaration&gt; int y =3; &lt;/jsp:declaration&gt;</code>
Skryptlety	<code>&lt;% lista.add("Fred"); %&gt;</code>	<code>&lt;jsp:scriptlet&gt; lista.add("Fred"); &lt;/jsp:scriptlet&gt;</code>
Tekst	Nie ma łyżeczki!	<code>&lt;jsp:text&gt; Nie ma łyżeczki &lt;/jsp:text&gt;</code>
Wyrażenia skryptowe	<code>&lt;%= it.next() %&gt;</code>	<code>&lt;jsp:expression&gt; it.next(); &lt;/jsp:expression&gt;</code>



**Relax** Oto wszystkie informacje dotyczące dokumentów JSP, jakie są wymagane na egzaminie.

Nie mamy najmniejszego zamiaru podawać jakichkolwiek dodatkowych informacji na temat pisania dokumentów JSP zgodnych z XML-em, gdyż prawdopodobnie nigdy nie będziesz korzystał z tej możliwości. Ta składnia jest przeważnie używana przez różnego typu narzędzia, a powyższą tabelę zamieściliśmy tylko po to, abyś widział, w jaki sposób używany program przekształca zwyczajną składnię JSP na dokument XML. Jeśli jednak chcesz samodzielnie tworzyć takie dokumenty, to musisz wiedzieć o kilku dodatkowych zagadnieniach. Na przykład cały dokument musi być umieszczony w elemencie `<jsp:root>` (który może zawierać pewne dodatkowe informacje), a dyrektywa taglib nie jest definiowana przy użyciu elementu `<jsp:directive>`, lecz umieszczana w znaczniku otwierającym elementu `<jsp:root>`. A zatem uspokój się i zrelaksuj.

# Zapamiętanie znaczników deskryptora wdrożenia związanych z komponentami EJB

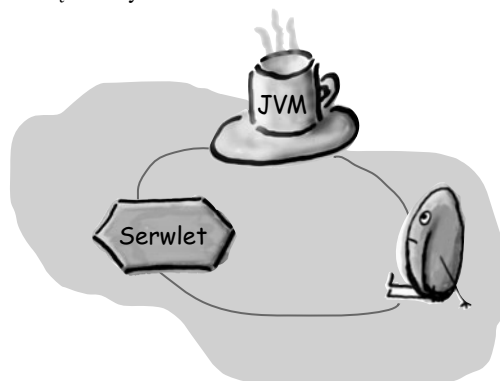
Egzamin, do którego przygotowuje Cię niniejsza książka, jest poświęcony komponentom internetowym, a nie biznesowym (choć kilka zagadnień związanych z tymi komponentami przedstawimy w rozdziale 14., poświęconym wzorcom). Jeśli jednak wdrażasz aplikację J2EE, która w warstwie biznesowej zawiera komponenty Enterprise JavaBean (EJB), to niektóre z używanych komponentów internetowych będą zapewne współpracować z komponentami EJB. Jeśli aplikacja jest wdrażana i ma działać w kontenerze J2EE (zawierającym także kontener EJB), to w deskrytorze wdrożenia możesz dodatkowo zdefiniować odwołania do komponentów EJB. Przystępując do egzaminu, jedyną informacją związaną z komponentami EJB, jaką musisz znać, jest to, iż można je deklarować w deskrytorze wdrożenia. A zatem nie będziemy tracić więcej czasu na opisywanie tych zagadnień\*.

### Odwołania do komponentu lokalnego

```
<ejb-local-ref>
 <ejb-ref-name>ejb/Klient</ejb-ref-name>
 <ejb-ref-type>Entity</ejb-ref-type>
 <local-home>com.bardzospytni.KlientHome</local-home>
 <local>com.bardzospytni.Klient</local>
</ejb-local-ref>
```

*Nazwa używana w kodzie, która będzie poszukiwana przy użyciu JNDI.*

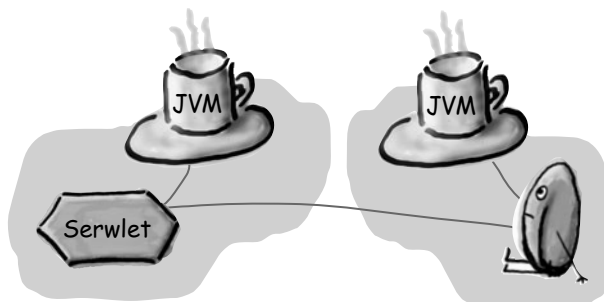
*To musi być w pełni kwalifikowana nazwa udostępnionego interfejsu komponentu.*



Komponent **LOKALNY** oznacza, że klient (w tym przypadku jest nim serwlet) oraz komponent muszą działać na tej samej wirtualnej maszynie Javy (JVM).

### Odwołania do zdalnego komponentu

```
<ejb-ref>
 <ejb-ref-name>ejb/KlientLokalny</ejb-ref-name>
 <ejb-ref-type>Entity</ejb-ref-type>
 <home>com.bardzospytni.KlientHome</home>
 <remote>com.bardzospytni.Klient</remote>
</ejb-ref>
```



Komponent **ZDALNY** oznacza, że klient (w tym przypadku jest nim serwlet) oraz komponent mogą działać na różnych wirtualnych maszynach Javy (prawdopodobnie na różnych komputerach).

\* Jeśli interesuje Cię technologia EJB, to jest pewna bardzo dobra książka...



Oglądaj to!

**Znaczniki LOCAL oraz REMOTE są niespójne!**

Znaczniki używane w deskrytorze wdrożenia do opisanie komponentów lokalnych i zdalnych mają dwa identyczne elementy:

Element `<ejb-ref-name>` określający nazwę logiczną, której będziesz używać w kodzie do wyszukiwania (za pomocą interfejsu JNDI) i pobierania interfejsu domowego komponentu. (Nie przejmuj się, jeśli jeszcze nigdy nie używałeś komponentów EJB i nie wiesz, co oznacza to ostatnie zdanie — przystępując do egzaminu, nie musisz znać technologii EJB).

Element `<ejb-ref-type>` określa, czy mamy do czynienia z komponentem encyjnym (Entity) czy też sesyjnym (Session). Te dwa elementy nie zależą od tego, czy komponent jest lokalny (czyli czy obsługuje go ta sama wirtualna maszyna Javy co komponent internetowy) czy zdalny (czyli czy jest on wykonywany przez inną wirtualną maszynę Javy).

Ale przyjrzyj się innym elementom, zaczynając od znaczników zewnętrznych — `<ejb-local-ref>` oraz `<ejb-ref>`. Być może będzie Cię kusić, aby zastosować następujące znaczniki:

`<ejb-local-ref>` ← tak

`<ejb-remote-ref>` ← Błąd!!

Tak nie można! W przypadku komponentów zdalnych należy zastosować znacznik:

`<ejb-ref>` ← Dobrze! W tym znaczniku nie ma słowa „remote”.

Innymi słowy, w odwołaniach lokalnych umieszczamy informacje o ich lokalnym charakterze, natomiast nazwa elementu dla odwołań zdalnych nie zawiera słowa „remote”. Dlaczego? Ponieważ w czasie, gdy opracowywano znacznik `<ejb-ref>`, nie istniało pojęcie „lokalnych” komponentów EJB. Wszystkie komponenty EJB były „zdalne”, a zatem rozróżnianie pomiędzy odwołaniami zdalnymi i lokalnymi nie miało sensu. Właśnie dlatego w znaczniku definiującym odwołanie zdalne nie ma słowa „remote”.

To tłumaczy także niespójności występujące w nazwach INNYCH znaczników, a konkretnie — w nazwie znacznika określającego interfejs bazowy (domowy) komponentu. W przypadku komponentu lokalnego elementem tym jest:

`<local-home>` ← tak

natomiast w przypadku komponentów zdalnych nie jest stosowany analogiczny element:

`<remote-home>` ← Błąd!!

W przypadku komponentów zdalnych używany jest element:

`<home>`

# Zapamiętanie znacznika JNDI < env-entry >

Jeśli znasz technologię EJB i (lub) JNDI, to zamieszczone tu informacje będą dla Ciebie zrozumiałe i sensowne. Jeśli jednak nie znasz tych technologii, to nie musisz się tym przejmować, gdyż z punktu widzenia przystępowania do egzaminu zapamiętanie tego elementu nie ma wielkiego znaczenia. (Szczegółowe informacje dotyczące danych środowiskowych JNDI zostały podane w książce poświęconej zagadnieniom EJB i J2EE — *Head First EJB*).

Wpis środowiskowy można postrzegać jako swoistą stałą dostępną w czasie wdrażania, której możesz używać w swojej aplikacji, podobnie jak parametrów inicjalizacyjnych serwetów i kontekstu. Innymi słowy, tak reprezentowane dane pozwalają osobie wdrażającej aplikację na przekazanie pewnych informacji do serwetu (a w tym przypadku także do komponentu EJB, gdyż jest on wdrażany jako fragment aplikacji korporacyjnej na serwerze J2EE).

W trakcie wdrażania kontener odczytuje zawartość deskryptora wdrożenia i na podstawie podanej w nim nazwy i wartości tworzy wpis JNDI (zakładając oczywiście, że aplikacja jest w pełni zgodna z technologią J2EE i nie używamy serwera dysponującego wyłącznie kontenerem *internetowym*). W czasie wykonywania aplikacji wchodzące w jej skład komponenty mogą odczytywać wartości przy użyciu JNDI, posługując się przy tym nazwą podaną w deskrytorze wdrożenia. Prawdopodobnie nie będziesz używał elementu <env-entry>, chyba że w skład Twojej aplikacji będą także wchodziły komponenty EJB. A zatem powinieneś zapamiętać ten element wyłącznie dlatego, iż pytanie na jego temat może się pojawić na egzaminie.

## Deklarowanie danych środowiskowych JNDI

```
<env-entry>
 <env-entry-name>stawki/stawki0bnizone</env-entry-name>
 <env-entry-type>java.lang.Integer</env-entry-type>
 <env-entry-value>10</env-entry-value>
</env-entry>
```

Nazwa logiczna, której będziesz używać w kodzie.

To może być dowolny typ, dla którego zdefiniowano jednoargumentowy konstruktor otrzymujący na wejściu wartość typu String (lub pojedynczy znak, jeśli tym typem jest `java.lang.Character`).

Ta wartość zostanie przekazana do konstruktora w postaci łańcucha znaków (obiektu typu `String` lub jako obiekt `Character`, jeśli w elemencie <env-entry-type> wskazano typ `java.lang.Character`).



Oglądaj to!

### Typ podany w elemencie <env-entry-type> NIE może być typem podstawowym!

Może Ci przyjść do głowy, że skoro w elemencie <env-entry-value> chcesz podać wartość całkowitą (jak w powyższym przykładzie), w elemencie <env-entry-type> powinieneś podać jakiś typ podstawowy. Jednak takie rozwiązanie będzie błędne. Możesz także myśleć, że typem wartości środowiskowej może być wyłącznie `String` lub klasa reprezentująca typy podstawowe. Jednak takie przypuszczenie także jest błędne — można użyć dowolnej klasy definiującej jednoargumentowy konstruktor pobierający wartość typu `String` (lub pojedynczy znak w przypadku zastosowania typu `Character`).

Uwaga: W elemencie <env-entry> można także umieścić opcjonalny element <description>, co jest naprawdę DOSKONAŁYM pomysłem.

## Zapamiętanie znacznika <mime-mapping>

W deskrypcji wdrożenia można także skonfigurować odwzorowania pomiędzy rozszerzeniem a typem MIME. Ten element będzie Ci zapewne najłatwiej zapamiętać, gdyż jest całkowicie zrozumiały — kojarzymy rozszerzenie z typem MIME. I nie zgadłbyś... w jakże rzadkim momencie przebłyśku prostoty i przejrzystości podelementom określającym rozszerzenie i typ MIME nadano nazwy: „extension” i „mime-type”, a to oznacza, że musisz zapamiętać tylko to, że *elementy znacznika noszą takie nazwy, jakich można by się spodziewać!*

No, chyba że zaczniesz myśleć o takich sprawach jak „typ pliku” lub „typ zawartości”. Ale nie... przecież Ty byś tego nie zrobił. Ty zapamiętasz znacznik w poprawnej postaci.

### Deklaracja elementu <mime-mapping>

```
<mime-mapping>
 <extension>mpg</extension>
 <mime-type>video/mpeg</mime-type>
</mime-mapping>
```

NIE dopisuj kropki („.”) przed rozszerzeniem



Oglądaj to!

**Przed rozszerzeniem  
nie zapisuj „.”!**

Rozszerzenie składa się wyłącznie z liter. Kropka („.”), która oddziela je od nazwy pliku, nie wchodzi w skład rozszerzenia.



Oglądaj to!

**Nie ma elementów  
<file-type> ani  
<content-type>!**

Zapamiętaj — są jedynie elementy  
<extension> i <mime-type>.  
<extension> i <mime-type>  
<extension> i <mime-type>  
<extension> i <mime-type>  
<extension> i <mime-type>



Zaostrz ołówek

Gdzie są umieszczane różne pliki?

W poniższej tabeli wpisz notatki opisujące miejsca aplikacji internetowej, w których należy umieszczać wymienione zasoby. Pierwszą odpowiedź podaliśmy za Ciebie. Odpowiedzi znajdują się na następnej karcie.

Typ zasobu	Lokalizacja
Deskryptor wdrożenia (web.xml)	Bezpośrednio w katalogu WEB-INF (który znajduje się w katalogu głównym aplikacji internetowej).
Pliki znaczników (.tag lub .tagx)	
Pliki HTML i JSP (które mają być bezpośrednio dostępne)	
Pliki HTML i JSP (które nie mają być bezpośrednio dostępne)	
Plik TLD (.tld)	
Klasy serwletów	
Klasy obsługi znaczników	
Pliki JAR	



## Zapamiętywanie znaczników deskryptora wdrożenia

Jeśli NIE zamierzasz zdawać egzaminu, to nie musisz się przejmować, gdy nie uda Ci się podać poprawnych rozwiązań do wszystkich fragmentów deskryptora (choć ostatnie dwa elementy są ważne niemal dla wszystkich).

Jeśli jednak zamierzasz przystąpić do egzaminu, to powinieneś poświęcić nieco czasu na zapamiętanie wszystkich tych elementów.

```
< _____ >
 < _____ >ejb/Klient< _____ >
 <ejb-ref-type>Entity</ejb-ref-type>
 < _____ >com.bardzospytni.Klinthome< _____ >
 <local>com.bardzospytni.Klient</local>
< _____ >
```

---

```
<ejb-ref>
 < _____ >ejb/KlientLokalny< _____ >
 <ejb-ref-type>Entity</ejb-ref-type>
 < _____ >com.bardzospytni.KlientHome< _____ >
 < _____ >com.bardzospytni.Klient< _____ >
</ejb-ref>
```

---

```
<env-entry>
 < _____ >stawki/stawkiObnizzone< _____ >
 < _____ >java.lang.Integer< _____ >
 <env-entry-value>10</env-entry-value>
</env-entry>
```

---

```
<error-page>
 < _____ >java.io.IOException< _____ >
 < _____ >/mojebledy.jsp< _____ >
</error-page>
```

---

```
< _____ >
 <welcome-file>index.html</welcome-file>
< _____ >
```



Zaostrz ołówkę      **Gdzie są umieszczane różne pliki?**

W poniższej tabeli wpisz notatki opisujące miejsca aplikacji internetowej, w których należy umieszczać wymienione zasoby. Pierwszą odpowiedź podaliśmy za Ciebie. Odpowiedzi znajdują się na następnej kartce.

Typ zasobu	Lokalizacja
Deskryptor wdrożenia (web.xml)	Bezpośrednio w katalogu WEB-INF (który znajduje się w katalogu głównym aplikacji internetowej).
Pliki znaczników (.tag lub .tagx)	Jeśli pliki znaczników NIE są umieszczone w pliku JAR, to muszą się znajdować w katalogu WEB-INF/tags lub jego podkatalogu. Jeśli jednak pliki znaczników wdrożono w pliku JAR, muszą się znajdować w katalogu META-INF/tags lub jego podkatalogu. Uwaga: dla plików znaczników umieszczonych w pliku JAR muszą istnieć odpowiednie deskryptory TLD umieszczone w tym samym pliku JAR.
Pliki HTML i JSP (które mają być bezpośrednio dostępne)	Pliki HTML i JSP dostępne dla klienta mogą się znajdować w katalogu głównym aplikacji lub jego podkatalogu, ZA WYJĄTKIEM katalogu WEB-INF (oraz jego podkatalogów). Pliki HTML i JSP wdrożone w pliku WAR nie mogą się znajdować w katalogu META-INF ani żadnym z jego podkatalogów.
Pliki HTML i JSP (które nie mają być bezpośrednio dostępne)	Strony umieszczone w katalogu WEB-INF i jego podkatalogach nie będą dostępne dla klientów (w przypadku umieszczania stron w pliku WAR należy je zapisać w katalogu META-INF).
Plik TLD (.tld)	Jeśli pliki TLD NIE są umieszczane w archiwum JAR, to muszą się znaleźć w katalogu WEB-INF lub jednym z jego podkatalogów. W przypadku umieszczania ich w archiwum JAR należy je zapisać w katalogu META-INF lub jednym z jego podkatalogów.
Klasy serwletów	Klasy serwletów należy zapisywać w hierarchii katalogów odpowiadającej nazwom pakietów i umieszczonej w katalogu WEB-INF/classes (na przykład klasę com.przyklad. Pierscien należałoby umieścić w katalogu WEB-INF/classes/com/przyklad/) bądź też w pliku JAR w katalogu WEB-INF/lib.
Klasy obsługi znaczników	Właściwie WSZYSTKIE klasy wchodzące w skład aplikacji internetowej (za wyjątkiem tych, które wchodzi w skład bibliotek dostępnych w ścieżce klas kontenera) muszą być rozmieszczane na tych samych zasadach co klasy serwletów — w strukturze katalogów odpowiadającej nazwie pakietu i zaczynającej się w katalogu WEB-INF/classes (lub, w przypadku pliku JAR, w odpowiednich katalogach umieszczonych w katalogu WEB-INF/lib).
Pliki JAR	Pliki JAR należy umieszczać w katalogu WEB-INF/lib.



## Zaostrz ołówki

Zapamiętywanie  
znaczników  
deskryptora wdrożenia

## ODPOWIEDZI

Jeśli zamierzasz zdawać egzamin, to powinieneś poświęcić nieco czasu na zapamiętanie WSZYSTKICH tych elementów (oraz wszelkich innych elementów przedstawionych w tym rozdziale, a także elementów związanych z zabezpieczaniem aplikacji opisanych w rozdziale następnym).

*Odwołanie do komponentu posiadającego interfejs „zdalny”.*

*Dane środowiskowe to sposób na przekazywanie do aplikacji internetowej informacji określanych podczas wdrażania.*

*Informuje kontener o tym, jaką stronę należy wyświetlić w przypadku wystąpienia błędu określonego typu (podanego w elemencie <exception-type>).*

*Informuje kontener, jakiej strony należy poszukiwać w przypadku odebrania żądania, które nie pasuje do żadnego konkretnego zasobu. W jednym elemencie <welcome-file-list> można umieścić więcej niż jeden element <welcome-file>.*

```
< ejb-local-ref >
 < ejb-ref-name >ejb/Klient< /ejb-ref-name >
 <ejb-ref-type>Entity</ejb-ref-type>
 < local-home >com.bardzospytni.KlintHome< /local-home >
 <local>com.bardzospytni.Klient</local>
< /ejb-local-ref >
```

---

```
<ejb-ref>
 < ejb-ref-name >ejb/KlientLokalny< /ejb-ref-name >
 <ejb-ref-type>Entity</ejb-ref-type>
 < home >com.bardzospytni.KlientHome< /home >
 < remote >com.bardzospytni.Klient< /remote >
</ejb-ref>
```

---

```
<env-entry>
 < env-entry-name >stawki/stawki0bnizone< /env-entry-name >
 < env-entry-type >java.lang.Integer< /env-entry-type >
 <env-entry-value>10</env-entry-value>
</env-entry>
```

---

```
<error-page>
 < exception-type >java.io.IOException< /exception-type >
 < location >/mojebledy.jsp< /location >
</error-page>
```

---

```
< welcome-file-list >
 <welcome-file>index.html</welcome-file>
< /welcome-file-list >
```

---



BAR  
KAWOWY

## *Egzamin próbny*

- 
- 1** Gdzie w deskrytorze wdrożenia może się pojawić element **<init-param>**?  
(Należy zaznaczyć wszystkie poprawne opcje).
- ☐ A. Bezpośrednio wewnątrz elementu **<servlet>**.
  - ☐ B. W dowolnym miejscu wewnątrz elementu **<web-application>**.
  - ☐ C. Bezpośrednio za deklaracją typu dokumentu.
  - ☐ D. Jeśli chcemy zadeklarować parametr inicjalizacyjny kontekstu, to wewnątrz elementu **<context-param>**.
- 
- 2** W jakim miejscu aplikacji należy przechowywać pliki TLD?  
(Należy zaznaczyć wszystkie poprawne opcje).
- ☐ A. Tylko w katalogu **/WEB-INF/lib**.
  - ☐ B. Tylko w katalogu **/WEB-INF/classes**.
  - ☐ C. W katalogu **/META-INF** w pliku JAR umieszczonym w katalogu **/WEB-INF/lib**.
  - ☐ D. W katalogu głównym aplikacji.
  - ☐ E. W katalogu **/WEB-INF** lub jednym z jego podkatalogów.
- 
- 3** Które z poniższych stwierdzeń dotyczących plików WAR są prawdziwe.  
(Należy zaznaczyć wszystkie poprawne opcje).
- ☐ A. Skrót WAR pochodzi od angielskich słów Web Application Resources (zasoby aplikacji internetowej).
  - ☐ B. Poprawny plik WAR musi zawierać deskryptor wdrożenia.
  - ☐ C. Na jedną aplikację internetową może się składać kilka plików WAR.
  - ☐ D. Plik WAR nie może zawierać plików JAR.

4 Deskryptor wdrożenia zawiera następującą deklarację serwletu:

```
<servlet>
 <servlet-name>MojSerwlet</servlet-name>
 <servlet-class>com.mojaorg.KlasaSerwletu</servlet-class>
</servlet>
```

W którym miejscu aplikacji można umieścić klasę tego serwletu?  
(Należy zaznaczyć wszystkie poprawne opcje).

- ☐ A. W katalogu **/META-INF** w pliku JAR.
- ☐ B. W drzewie katalogów odpowiadających nazwie pakietu i rozpoczynającym się na głównym poziomie aplikacji.
- ☐ C. W katalogu **/WEB-INF/classes** pliku JAR umieszczonego w katalogu **/WEB-INF/lib**.
- ☐ D. Jeśli plik jest umieszczany poza plikiem JAR, to w katalogu **/WEB-INF/lib**.

5 Do czego służy deskryptor wdrożenia? (Należy zaznaczyć wszystkie poprawne opcje).

- ☐ A. Do generowania kodu na podstawie zapisów XML-a przez narzędzia automatycznych generatorów.
- ☐ B. Do przekazywania informacji o konfiguracji aplikacji pomiędzy jej programistą, osobą scalającą aplikację oraz osobą odpowiedzialną za jej wdrażanie.
- ☐ C. Do konfigurowania rozmaitych aspektów aplikacji zależnych od używanego serwera.
- ☐ D. Wyłącznie do konfigurowania dostępu aplikacji do baz danych i komponentów EJB.

6 Gdzie w ramach pliku WAR należy umieścić plik **web.xml**?  
(Należy zaznaczyć wszystkie poprawne opcje).

- ☐ A. W katalogu **/WEB-INF/classes**.
- ☐ B. W katalogu **/WEB-INF/lib**.
- ☐ C. W katalogu **/WEB-INF**.
- ☐ D. W katalogu **/META-INF**.

7 Przyjrzyj się poniższym wierszom kodu:

```
10. <%@ page import="java.util.*" %>
11. <jsp:import import="java.util.*" />
12. <jsp:directive.page import="java.util.*" />
13. <jsp:page import="java.util.*" />
```

Zakładając, że prefiks „jsp” został skojarzony z przestrzenią nazw:

**http://java.sun.com/JSP/Page**

wskaż, które z poniższych stwierdzeń są prawdziwe. (Należy zaznaczyć wszystkie poprawne opcje).

- ☐ A. Wiersze 10. i 12. są równoważne w stronie JSP każdego typu.
- ☐ B. Wiersz 10. nie jest poprawny w dokumentach JSP (zgodnych ze składnią XML).
- ☐ C. Wiersz 11. spowoduje poprawne zaimportowanie pakietu **java.util**.
- ☐ D. Wiersz 12. spowoduje poprawne zaimportowanie pakietu **java.util**.
- ☐ E. Wiersz 13. spowoduje poprawne zaimportowanie pakietu **java.util**.

---

8 Które z poniższych stwierdzeń dotyczących elementu **<init-param>** deskryptora wdrożenia są prawdziwe? (Należy zaznaczyć wszystkie poprawne opcje).

- ☐ A. Elementy te są używane do deklarowania parametrów inicjalizacyjnych konkretnych serwletów.
- ☐ B. Używa się ich do deklarowania parametrów inicjalizacyjnych całej aplikacji internetowej.
- ☐ C. Metoda odczytująca wartości tych parametrów zwraca obiekt.
- ☐ D. Metoda odczytująca wartości tych parametrów wymaga podania wartości typu String.

---

9 Które z poniższych elementów są używane w deskrytorze wdrożenia i zapewniają dostęp do komponentów J2EE przy wykorzystaniu technologii JNDI? (Należy zaznaczyć wszystkie poprawne opcje).

- ☐ A. **<ejb-ref>**
- ☐ B. **<entity-ref>**
- ☐ C. **<ejb-local-ref>**
- ☐ D. **<session-ref>**
- ☐ E. **<ejb-remote-ref>**

**10** W deskrytorze wdrożenia został zadeklarowany następujący serwlet:

```
<servlet>
 <servlet-name>akcja</servlet-name>
 <servlet-class>com.mojaorg.KlasaAkcji</servlet-class>
</servlet>
```

Wskaż poprawne odwzorowanie dla tego serwletu. (Należy zaznaczyć wszystkie poprawne opcje).

- ☐ A. 

```
<servlet-mapping>
 <servlet-name>akcja</servlet-name>
 <url-pattern>*.do</url-pattern>
</servlet-mapping>
```
- ☐ B. 

```
<servlet-mapping>
 <servlet-name>com.mojaorg.KlasaAkcji</servlet-name>
 <url-pattern>*.do</url-pattern>
</servlet-mapping>
```
- ☐ C. 

```
<servlet-mapping>
 <servlet-name>akcja</servlet-name>
 <url-pattern>/kontroler</url-pattern>
</servlet-mapping>
```
- ☐ D. 

```
<servlet-mapping>
 <url-pattern>*.do</url-pattern>
</servlet-mapping>
```
- ☐ E. 

```
<servlet-mapping>
 <servlet-name>akcja</servlet-name>
</servlet-mapping>
```

**11** Dla których rodzajów komponentów aplikacji internetowej można definiować zależności? (Należy zaznaczyć wszystkie poprawne opcje).

- ☐ A. Plików JAR.
- ☐ B. Plików WAR.
- ☐ C. Klas.
- ☐ D. Bibliotek.
- ☐ E. Plików manifestu.

**12** Które z poniższych deklaracji umieszczonych w dokumencie JSP (zgodnym ze składnią XML) są poprawne. (Należy zaznaczyć wszystkie poprawne opcje).

- ☐ A. `<jsp:declaration`  
    `xmlns:jsp="http://java.sun.com/JSP/Page">`  
    `int x = 0;`  
    `</jsp:declaration>`
- ☐ B. `<jsp:declaration`  
    `xmlns:jsp="http://java.sun.com/JSP/Page">`  
    `int x;`  
    `</jsp:declaration>`
- ☐ C. `<%! int x = 0; %>`
- ☐ D. `<%! int x; %>`

**13** Które z elementów deskryptora wdrożenia mogą się pojawiać przed elementem `<web-app>`? (Należy zaznaczyć wszystkie poprawne opcje).

- ☐ A. `<listener>`
- ☐ B. `<context-param>`
- ☐ C. `<servlet>`
- ☐ D. Żaden z elementów XML nie może się pojawić przed elementem `<web-app>`.

**14** Które z poniższych stwierdzeń dotyczących mechanizmu ładowania klas kontenera są prawdziwe? (Należy zaznaczyć wszystkie poprawne opcje).

- ☐ A. Aplikacja internetowa NIE powinna podejmować prób przesłaniania klas implementowanych przez kontener.
- ☐ B. Aplikacja internetowa nie może próbować wczytywania zasobów z pliku WAR za pomocą standardowej metody `getResource()` platformy J2SE.
- ☐ C. Aplikacja internetowa może przesłaniać dowolne klasy J2EE dostępne w przestrzeni nazw *javax.\**.
- ☐ D. Programista aplikacji internetowych może przesłaniać dowolne klasy platformy J2EE, pod warunkiem że klasy, którymi są one zastępowane, znajdują się w bibliotece JAR w ramach pliku WAR.



## Egzamin próbny — odpowiedzi

**1** Gdzie w deskrytorze wdrożenia może się pojawić element **<init-param>**? (Specyfikacja serwetów, str. 107).  
(Należy zaznaczyć wszystkie poprawne opcje).

- ☒ A. Bezpośrednio wewnątrz elementu **<servlet>**.
- ☐ B. W dowolnym miejscu wewnątrz elementu **<web-application>**.  
– Odpowiedź B jest błędna, gdyż plik *web.xml* nie zawiera żadnego elementu o nazwie *<web-application>*.
- ☐ C. Bezpośrednio za deklaracją typu dokumentu.
- ☐ D. Jeśli chcemy zadeklarować parametr inicjalizacyjny kontekstu, to wewnątrz elementu **<context-param>**.  
– Odpowiedź D jest błędna, gdyż elementy *<context-param>* nie zawierają elementów *<init-param>*.

**2** W jakim miejscu aplikacji należy przechowywać pliki TLD? (Specyfikacja JSP, str. 196).  
(Należy zaznaczyć wszystkie poprawne opcje).

- ☐ A. Tylko w katalogu **/WEB-INF/lib**.
- ☐ B. Tylko w katalogu **/WEB-INF/classes**.
- ☒ C. W katalogu **/META-INF** w pliku JAR umieszczonym w katalogu **/WEB-INF/lib**.  
– Kontener nie będzie w stanie automatycznie odszukać plików TLD, jeśli zostaną one umieszczone w katalogu */WEB-INF/classes* lub */WEB-INF/lib*.
- ☐ D. W katalogu głównym aplikacji.
- ☒ E. W katalogu **/WEB-INF** lub jednym z jego podkatalogów.

**3** Które z poniższych stwierdzeń dotyczących plików WAR są prawdziwe. (Specyfikacja serwetów, 9.5 i 9.6).  
(Należy zaznaczyć wszystkie poprawne opcje).

- ☐ A. Skrót WAR pochodzi od angielskich słów Web Application Resources (zasoby aplikacji internetowej).
- ☒ B. Poprawny plik WAR musi zawierać deskryptor wdrożenia.  
– Skrót WAR pochodzi od angielskich słów *Web ARchive*, aplikacja internetowa nie może składać się z kilku plików WAR, w takim pliku można umieszczać jedynie całą aplikację.
- ☐ C. Na jedną aplikację internetową może się składać kilka plików WAR.
- ☐ D. Plik WAR nie może zawierać plików JAR.

- 4 Deskryptor wdrożenia zawiera następującą deklarację serwletu: (Specyfikacja serwletów, str. 70).

```
<servlet>
 <servlet-name>MojSerwlet</servlet-name>
 <servlet-class>com.mojaorg.KlasaSerwletu</servlet-class>
</servlet>
```

W którym miejscu aplikacji można umieścić klasę tego serwletu?  
(Należy zaznaczyć wszystkie poprawne opcje).

- ☐ A. W katalogu **/META-INF** w pliku JAR.
- ☐ B. W drzewie katalogów odpowiadających nazwie pakietu i rozpoczynającym się na głównym poziomie aplikacji.
- ☒ C. W katalogu **/WEB-INF/classes** pliku JAR umieszczonego w katalogu **/WEB-INF/lib**.  
– Odpowiedź D jest błędna, gdyż katalog **/WEB-INF/lib** jest przeznaczony do przechowywania plików JAR.
- ☐ D. Jeśli plik jest umieszczany poza plikiem JAR, to w katalogu **/WEB-INF/lib**.

- 5 Do czego służy deskryptor wdrożenia? (Należy zaznaczyć wszystkie poprawne opcje). (Specyfikacja serwletów, str. 103).

- ☐ A. Do generowania kodu na podstawie zapisów XML-a przez narzędzia automatycznych generatorów.
- ☒ B. Do przekazywania informacji o konfiguracji aplikacji pomiędzy jej programistą, osobą scalającą aplikację oraz osobą odpowiedzialną za jej wdrażanie.
- ☐ C. Do konfigurowania rozmaitych aspektów aplikacji zależnych od używanego serwera.
- ☐ D. Wyłącznie do konfigurowania dostępu aplikacji do baz danych i komponentów EJB.  
– Odpowiedź D jest niedokładna, ponieważ reprezentuje tylko niewielki podzbiór zadań stawianych deskryptorom wdrożenia.

- 6 Gdzie w ramach pliku WAR należy umieścić plik **web.xml**? (Należy zaznaczyć wszystkie poprawne opcje). (Specyfikacja serwletów, str. 70).

- ☐ A. W katalogu **/WEB-INF/classes**.
- ☐ B. W katalogu **/WEB-INF/lib**.
- ☒ C. W katalogu **/WEB-INF**.  
– Plik **web.xml** należy przechowywać w katalogu **/WEB-INF** niezależnie od tego, czy aplikacja jest wdrażana w formie pliku WAR czy też jako struktura katalogów i plików.
- ☐ D. W katalogu **/META-INF**.

7 Przyjrzyj się poniższym wierszom kodu:

(Specyfikacja JSP 2.0, str. 1 – 139).

```
10. <%@ page import="java.util.*" %>
11. <jsp:import import="java.util.*" />
12. <jsp:directive.page import="java.util.*" />
13. <jsp:page import="java.util.*" />
```

Zakładając, że prefiks „jsp” został skojarzony z przestrzenią nazw:

**http://java.sun.com/JSP/Page**

wskaż, które z poniższych stwierdzeń są prawdziwe. (Należy zaznaczyć wszystkie poprawne opcje).

- ☐ A. Wiersze 10. i 12. są równoważne w stronie JSP każdego typu.
- ☒ B. Wiersz 10. nie jest poprawny w dokumentach JSP (zgodnych ze składnią XML).
- ☐ C. Wiersz 11. spowoduje poprawne zaimportowanie pakietu **java.util**.
- ☒ D. Wiersz 12. spowoduje poprawne zaimportowanie pakietu **java.util**.
- ☐ E. Wiersz 13. spowoduje poprawne zaimportowanie pakietu **java.util**.

– Odpowiedź A jest błędna, gdyż w przypadku tworzenia dokumentu JSP wiersz 10. byłby błędny.

– Odpowiedzi C i E są błędne, gdyż nie są to poprawne elementy należące do przestrzeni nazw **http://java.sun.com/JSP/Page**.

8 Które z poniższych stwierdzeń dotyczących elementu **<init-param>** deskryptora wdrożenia są prawdziwe? (Należy zaznaczyć wszystkie poprawne opcje).

(Specyfikacja serwletów SRV.B oraz API).

- ☒ A. Elementy te są używane do deklarowania parametrów inicjalizacyjnych konkretnych serwletów.
- ☐ B. Używa się ich do deklarowania parametrów inicjalizacyjnych całej aplikacji internetowej.
- ☐ C. Metoda odczytująca wartości tych parametrów zwraca obiekt.
- ☒ D. Metoda odczytująca wartości tych parametrów wymaga podania wartości typu **String**.

– Parametry inicjalizacyjne mogą obejmować zasięgiem aplikację lub wybrany serwlet. Parametry zasięgu serwletu są w deskrytorze wdrożenia deklarowane przy użyciu elementu **<init-param>**, a metoda, która służy do odczytywania ich wartości, otrzymuje argument i zwraca wartość typu **String**. Z kolei parametry zasięgu aplikacji są w deskrytorze wdrożenia deklarowane przy użyciu elementu **<context-param>**, a odpowiednia metoda otrzymuje i zwraca wartość typu **String** (podobnie jak w przypadku parametrów zasięgu serwletu).

9 Które z poniższych elementów są używane w deskrytorze wdrożenia i zapewniają dostęp do komponentów J2EE przy wykorzystaniu technologii JNDI? (Należy zaznaczyć wszystkie poprawne opcje).

(Specyfikacja serwletów, 9.11).

- ☒ A. **<ejb-ref>**
- ☐ B. **<entity-ref>**
- ☒ C. **<ejb-local-ref>**
- ☐ D. **<session-ref>**
- ☐ E. **<ejb-remote-ref>**

– Co więcej, element **<ejb-local-ref>** udostępnia twórcom aplikacji internetowych odwołanie JNDI do komponentów J2EE.

10 W deskrypcorze wdrożenia został zadeklarowany następujący serwlet:

(Specyfikacja serwletów, str. 86).

```
<servlet>
 <servlet-name>akcja</servlet-name>
 <servlet-class>com.mojaorg.KlasaAkcji</servlet-class>
</servlet>
```

Wskaż poprawne odwzorowanie dla tego serwletu. (Należy zaznaczyć wszystkie poprawne opcje).



A. `<servlet-mapping>`  
     `<servlet-name>akcja</servlet-name>`  
     `<url-pattern>*.do</url-pattern>`  
`</servlet-mapping>`



B. `<servlet-mapping>`  
     `<servlet-name>com.mojaorg.KlasaAkcji</servlet-name>`  
     `<url-pattern>*.do</url-pattern>`

– Odpowiedź B jest błędna, gdyż nazwa serwletu została w niej pomyłona z nazwą klasy.



C. `<servlet-mapping>`  
     `<servlet-name>akcja</servlet-name>`  
     `<url-pattern>/kontroler</url-pattern>`  
`</servlet-mapping>`



D. `<servlet-mapping>`  
     `<url-pattern>*.do</url-pattern>`  
`</servlet-mapping>`

– Odpowiedź D jest błędna, gdyż został w niej pominięty element `<servlet-name>`, który musi być umieszczany w elemencie `<servlet-mapping>`.



E. `<servlet-mapping>`  
     `<servlet-name>akcja</servlet-name>`  
`</servlet-mapping>`

11 Dla których rodzajów komponentów aplikacji internetowej można definiować zależności? (Należy zaznaczyć wszystkie poprawne opcje).

(Specyfikacja serwletów, 9.7.1).



A. Plików JAR.



B. Plików WAR.

– Zależności bibliotek można definiować w pliku `/META-INF/MANIFEST.MF`.



C. Klas.



D. Bibliotek.



E. Plików manifestu.

12 Które z poniższych deklaracji umieszczonych w dokumencie JSP (zgodnym ze składnią XML) są poprawne. (Należy zaznaczyć wszystkie poprawne opcje).

(Specyfikacja JSP wersja 2.0, str. 1 – 139).



A. `<jsp:declaration`

`xmlns:jsp="http://java.sun.com/JSP/Page">`

`int x = 0;`

`</jsp:declaration>`



B. `<jsp:declaration`

`xmlns:jsp="http://java.sun.com/JSP/Page">`

`int x;`

`</jsp:declaration>`



C. `<%! int x = 0; %>`



D. `<%! int x; %>`

– Odpowiedź C i D są błędne, gdyż poprawną formą zapisu, jaką należy stosować w dokumentach JSP, jest `<jsp:declaration>`.

13 Które z elementów deskryptora wdrożenia mogą się pojawiać przed elementem `<web-app>`? (Należy zaznaczyć wszystkie poprawne opcje).

(Specyfikacja serwletów, str. 107).



A. `<listener>`



B. `<context-param>`



C. `<servlet>`



D. Żaden z elementów XML nie może się pojawić przed elementem `<web-app>`.

– Element `<web-app>` jest elementem głównym deskryptora wdrożenia aplikacji internetowej.

14 Które z poniższych stwierdzeń dotyczących mechanizmu ładowania klas kontenera są prawdziwe? (Należy zaznaczyć wszystkie poprawne opcje).

(Specyfikacja serwletów, 9.7.2).



A. Aplikacja internetowa NIE powinna podejmować prób przesłaniania klas implementowanych przez kontener.



B. Aplikacja internetowa nie może próbować wczytywania zasobów z pliku WAR za pomocą standardowej metody `getResource()` platformy J2SE.



C. Aplikacja internetowa może przesłaniać dowolne klasy J2EE dostępne w przestrzeni nazw `javax.*`.



D. Programista aplikacji internetowych może przesłaniać dowolne klasy platformy J2EE, pod warunkiem że klasy, którymi są one zastępowane, znajdują się w bibliotece JAR w ramach pliku WAR.

– Odpowiedź B jest błędna, gdyż aplikacja internetowa może korzystać z metody `getResource()` mechanizmu ładowania klas w celu uzyskania dostępu do zawartości dowolnego pliku WAR.

– Odpowiedź C i D są błędne, gdyż aplikacja internetowa NIE może przesłaniać żadnych klas z przestrzeni nazw `java.*` ani `javax.*`.



## 12. Bezpieczeństwo aplikacji internetowych

# **Zachowaj je w tajemnicy, ukryj je w bezpiecznym miejscu**

Oni tam są... źli faceci...  
oni są wszędzie! Muszę się nauczyć  
zasad uwierzytelniania  
i autoryzacji... Muszę się nauczyć sposobów  
bezpiecznego przesyłania danych.  
Ja... Ja... SŁYSZAŁEŚ TO?



**Twoja aplikacja internetowa jest w *niebezpieczeństwie*.** Problemy czyhają w każdym zakamarku sieci. Zagroženiem są „crackerzy”, „scummerzy” oraz wszelkiego typu kryminaliści próbujący włamać się do Twojego systemu w poszukiwaniu wymiernych korzyści lub po prostu dla zabawy. Nie chcesz, aby ci źli ludzie podsłuchiwali transakcje realizowane w Twoim sklepie internetowym albo przechwytywali numery kart kredytowych. Nie chcesz, by byli w stanie przekonać Twój serwer, iż tak naprawdę są Bardzo Ważnymi Klientami Posiadającymi Bardzo Duże Upusty. I w końcu nie chcesz, by *ktokolwiek* (niezależnie od tego, czy dobry czy zły) miał dostęp do ważnych informacji na temat pracowników. Czy Janek z działu marketingu naprawdę musi wiedzieć, że Liza z działu technicznego zarabia trzy razy więcej niż on? I czy naprawdę chcesz, by Janek wziął sprawy w swoje ręce i (w nieupoważniony sposób) zalogował się do serwetu ZminaWysokosciWynagrodzenia?



### **Bezpieczeństwo aplikacji internetowej**

- 5.1.** Bazując na specyfikacji serwletów, porównaj i wskaż różnice pomiędzy następującymi pojęciami: a) uwierzytelnianie, b) autoryzacja, c) integralność danych oraz d) poufność.
- 5.2.** W deskrytorze wdrożenia zadeklaruj: ograniczenia bezpieczeństwa, zasób internetowy, gwarancję bezpiecznego transportu, konfigurację logowania oraz rolę bezpieczeństwa.
- 5.3.** Opisz mechanizm działania różnych typów uwierzytelniania (BASIC, DIGEST, FORM oraz CLIENT-CERT).

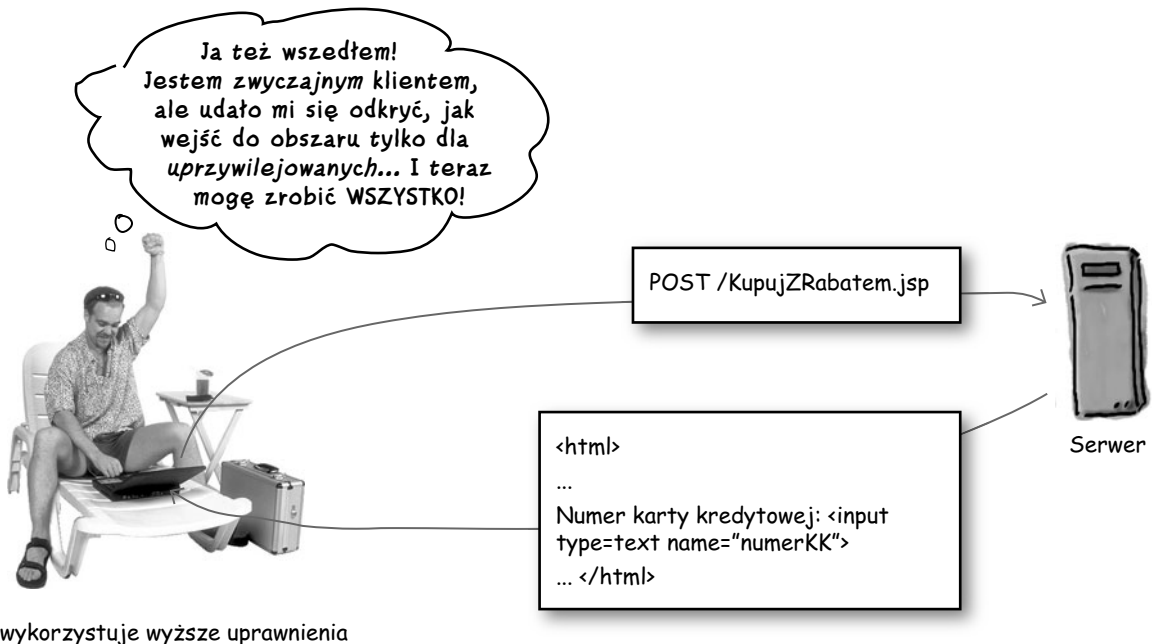
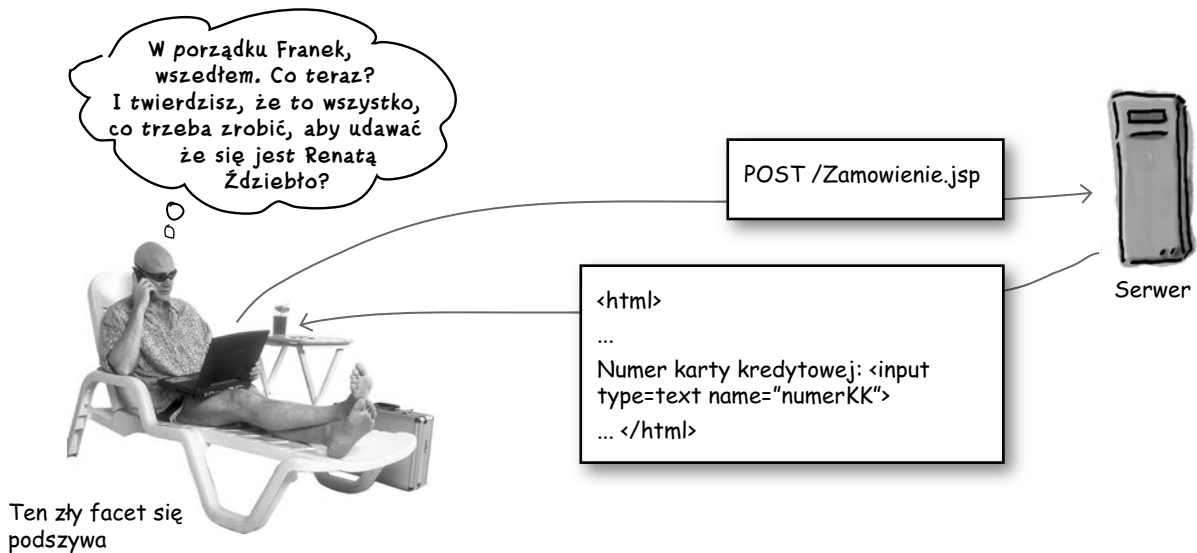
### **Uwagi wyjaśniające:**

*Wszystkie cele podane w lewej kolumnie zostały wyczerpująco opisane w niniejszym rozdziale, włącznie z elementami deskryptora wdrożenia związanymi z zabezpieczaniem aplikacji internetowych, które nie zostały zaprezentowane w rozdziale poprzednim, poświęconym zagadnieniom wdrażania aplikacji.*

*Nie możemy zapewnić Ci całkowitego bezpieczeństwa, jednak informacje zamieszczone tutaj pozwolą Ci zacząć studiowanie tych zagadnień i w zupełności wystarczą, by przystąpić do egzaminu.*

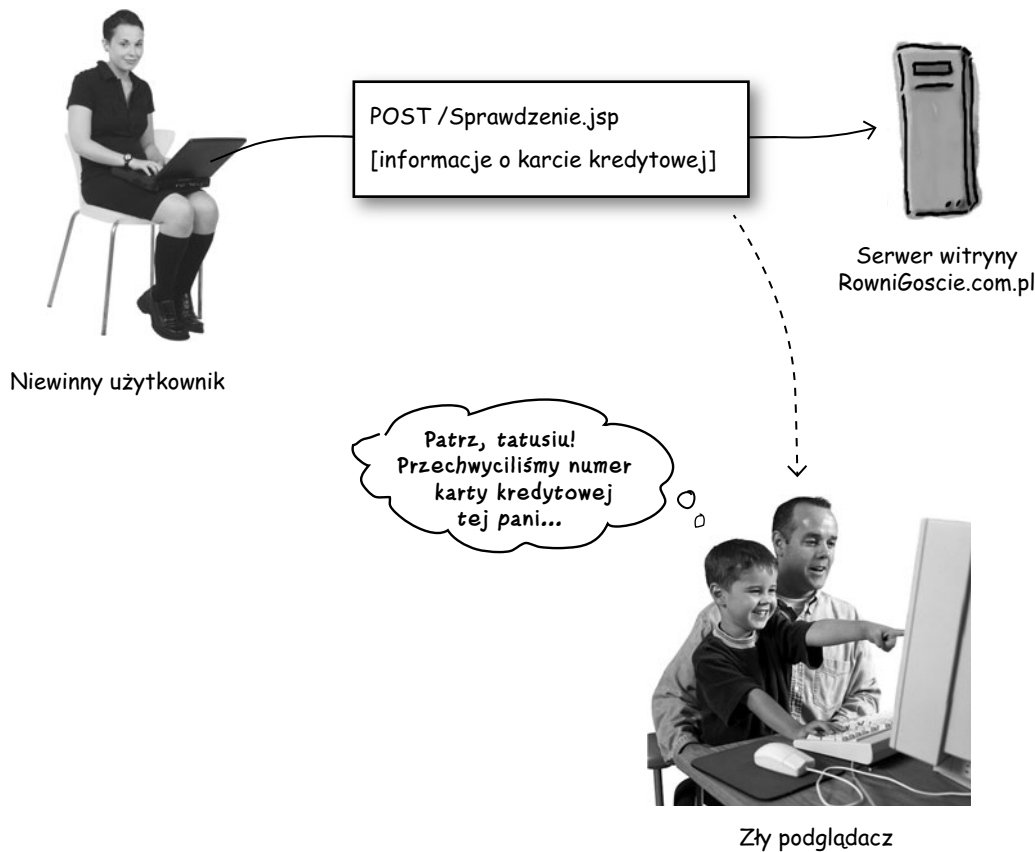
## Źli faceci są wszędzie

Jako twórca aplikacji internetowej musisz ją ochraniać. Istnieją trzy podstawowe sposoby działalności *złych facetów*, na które musisz zwracać uwagę: **podsywanie się**, **przyznawanie sobie wyższych uprawnień** i **podglądanie**.



# Problem zabezpieczeń nie dotyczy wyłącznie SERWERA...

Najgorsze może być **podśluchiwanie**. Osoby stosujące tę technikę nie tylko próbują oszukać *aplikację internetową*, lecz także mogą przysporzyć problemów Twoim dobrym *klientom*. Straty mogą więc być podwójne. Jeśli próba podsłuchania transmisji powiedzie się, podsłuchujący może przechwycić numer karty kredytowej klienta i błyskawicznie go wykorzystać.



# Wielka czwórka bezpieczeństwa serwletów

Mechanizmy zabezpieczeń serwletów pozwalają nam — programistom aplikacji internetowych — skutecznie walczyć z podszywaniem się, nieuprawnionym korzystaniem z wyższych uprawnień i podglądaniem. Jeśli chodzi o specyfikację serwletów (a zatem także o egzamin), to kwestie bezpieczeństwa serwletów sprowadzają się do czterech zagadnień: **uwierzytelniania**, **autoryzacji**, **poufności** oraz **integralności danych**.



## Krótką opowieści o bezpieczeństwie

Pewnego dnia szef wezwał Boba do swojego gabinetu. „Znalazłem dla Ciebie fascynujący nowy projekt!” — powiedział szef. Bob aż jęknął. „Wiem, że zadania, które dawałem Ci w przeszłości, nie były najlepsze, ale to powinno być bardzo interesujące... Chciałbym, abyś opracował zabezpieczenia naszej nowej witryny do handlu internetowego”. „Zagadnienia bezpieczeństwa są trudne i nudne...” — odrzekł Bob. „Mylisz się” — powiedział na to szef. „Nie w J2EE 1.4. Tam zabezpieczenia serwetów są podobno odlotowe”.

„Zlecam Ci na razie projekt pilotażowy, żebyś wiedział, od czego zacząć, a następnie, kiedy już wszystko sobie przemyślisz, pogadamy o szczegółach” — zdecydował szef. „No dobra” — mruknął Bob. „Zwal to na mnie”.

„Jak zapewne wiesz, ta witryna o piwie stała się ostatnio bardzo popularna. Dodaliśmy do niej kilka nowych możliwości i zainteresowanie użytkowników jest ogromne. Niektórym wystarczają *bezpłatne* przepisy umieszczone na witrynie, jednak znacznie więcej osób niż początkowo przypuszczaliśmy, jest skłonna *zapłacić* za nasze doskonałe składniki. A prawdziwym hitem okazał się też nasz program **Piwowar Praktykujący**. Jeśli użytkownik zdecyduje, że ponownie zakupi składniki do produkcji piwa, to wnosząc jednokrotną dodatkową opłatę, może podnieść swój status do **Mistrza Piwowskiego**. Tacy użytkownicy uzyskują specjalne zniżki i zdobywają *Punkty Piwne*, które następnie mogą zamieniać na odlotowe nagrody z pianką...”

Bob słuchał, wyobrażając już sobie kod, który będzie musiał napisać, aby to wszystko zaimplementować, i żegnając się już z wymarzonymi wakacjami na Karaibach. A szef cały czas kontynuował przedstawianie swoich wizji...

„Jednak w takiej sytuacji musimy mieć pewność, że nikt nie przechwyci ani nie ukradnie numerów kart kredytowych naszych klientów. A może inaczej... może lepiej byłoby upewnić się, że kiedy *użytkownik* się loguje, to jest to on sam, a nie jeden z jego *przyjaciół*? Uważam, że od teraz nasi użytkownicy będą musieli posługiwać się *hasłami*”.

„To wszystko, jak na razie, ma sens” — powiedział Bob. — „Czy użytkownicy składający zamówienia w naszym sklepie internetowym powinni otrzymywać od nas coś na kształt kodów potwierdzenia?”. „Doskonały pomysł” — powiedział szef. — „I jeszcze jedno, żebyś nie zapomniał — lepiej upewnij się, że tylko nasi **Piwowarzy Praktykujący** będą dostawać specjalne zniżki”.

„Myślę, że na początek to wystarczy” — rzekł szef. „Ale wiesz, jak to jest... pewnie już niedługo będziemy chcieli zaoferować jakiś nowy, wyższy stopień członkostwa...”.



### WYTEŻ UMYŚŁ

**Jakie pojęcia z zakresu zabezpieczania aplikacji internetowych można wyróżnić w powyższej opowieści?**

Przeczytaj opowieść jeszcze raz i zaznacz te miejsca, w których wymagania podawane przez szefa odnoszą się do:

- uwierzytelniania;
- autoryzacji;
- poufności;
- integralności danych.

(Tak, tak... wiemy, że to jest oczywiste, ale chcemy rozbudzić Twoją ciekawość, zanim przejdziemy do szczegółowego przedstawiania tych zagadnień).

# Krótka opowieści o bezpieczeństwie

Pewnego dnia szef wezwał Boba do swojego gabinetu. „Znalazłem dla Ciebie fascynujący nowy projekt” — powiedział szef. Bob aż jęknął. „Wiem, że zadania, które dawałem Ci w przeszłości, nie były najlepsze, ale to powinno być bardzo interesujące... Chciałbym, abyś opracował zabezpieczenia naszej nowej witryny do handlu internetowego”. „Zagadnienia bezpieczeństwa są trudne i nudne...” — odrzekł Bob. „Myślisz się” — powiedział na to szef. „Nie w J2EE 1.4. Tam zabezpieczenia serwerów są podobno odłotowe”.

„Złecam Ci na razie projekt pilotażowy, żebyś wiedział, od czego zacząć, a następnie, kiedy już wszystko sobie przemyślisz, pogadamy o szczegółach” — zdecydował szef. „No dobra” — mrknął Bob. „Zwał to na mnie”.

„Jak zapewne wiesz, ta witryna o piwie stała się ostatnio bardzo popularna. Dodałiśmy do niej kilka nowych możliwości i zainteresowanie użytkowników jest ogromne. Niektórym wystarczą *bezpłatne* przepisy umieszczane na witrynie, jednak znacznie więcej osób niż początkowo przypuszczaliśmy, jest skłonna *zapłacić* za nasze doskonałe składniki. A... prawdziwym hitem okazał się też nasz program **Piwowar Praktykujący**. Jeśli użytkownik zdecyduje, że ponownie zakupi składniki do produkcji piwa, to wnosząc jednokrotną dodatkową opłatę może podnieść swój status do **Mistrza Piwowarskiego**. Tacy użytkownicy uzyskują specjalne zniżki i zdobywają *Punkty Piwne*, które następnie mogą zamieniać na odłotowe nagrody z pianką...”

Bob słuchał, wyobrażając już sobie kod, który będzie musiał napisać, aby to wszystko zaimplementować, i żegnając się już z wymarzonymi wakacjami na Karaibach. A szef cały czas kontynuował przedstawianie swoich wizji... „Jednak w takiej sytuacji musimy mieć pewność, że nikt nie przechwyci ani nie ukradnie numerów kart kredytowych naszych klientów. A może inaczej... może lepiej byłoby upewnić się, że kiedy *użytkownik* się loguje, to jest to on sam, a nie jeden z jego *przyjaciół*? Uważam, że od teraz nasi użytkownicy będą musieli posługiwać się *hasłami*”.

## Bezpieczeństwo aplikacji internetowych

„To wszystko, jak na razie, ma sens” — powiedział Bob. — „Czy użytkownicy składający zamówienia w naszym sklepie internetowym powinni otrzymywać od nas coś na kształt kodów potwierdzenia?”. „Doskonale pomysł” — powiedział szef — „! jeszcze jedno, żebyś nie zapominał — lepiej upewnij się, że tylko nasi **Piwowarzy Praktykujący** będą dostawać specjalne zniżki”.

„Myśle, że na początek to wystarczy” — powiedział szef. „Ale wiesz, jak to jest... pewnie już niedługo będziemy chcieli zaferować jakiś nowy, wyższy stopień członkostwa...”.

**POUFNOŚĆ ! INTEGRALNOŚĆ DANYCH** — w tym momencie serwer zwraca uwagę do prywatnej informacji! przez osoby nieupoważnione mogłaby mieć fatalne skutki.

**AUTORYZACJA** — kiedy już określiliśmy, z kim rozmawiamy, to musimy się upewnić, że osoby te mają prawo do wykonania tych czynności, które chcą wykonać.

### Mistrza Piwowarskiego

**POUFNOŚĆ** — byłoby fatalnym niedopatrzeniem zasad bezpieczeństwa, gdyby numery kart kredytowych klientów wpadły w niepowołane ręce.

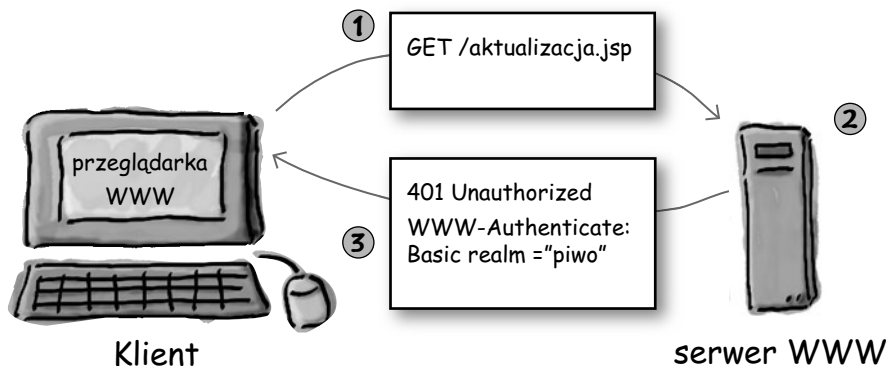
**UWIERZYTELNIANIE** — kiedy któśkolwiek wspomina o *hasłach*, to zapewne mówi o *uwierzytelnianiu*... czy on naprawdę jest tą osobą, za którą się podaje? Jeśli tak, to powinien znać *hasło*!

# Jak realizować uwierzytelnianie w świecie protokołu HTTP?

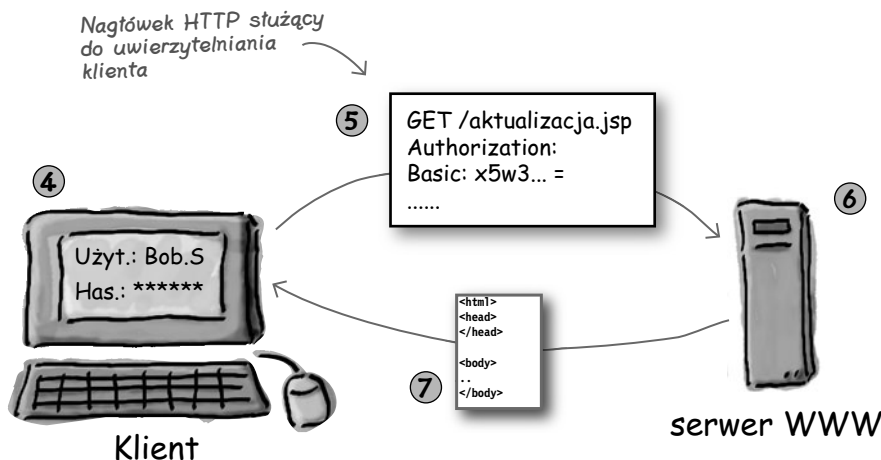
## Początek bezpiecznej transakcji

Zacznijmy od przeanalizowania wymiany informacji pomiędzy przeglądarką a kontenerem internetowym w chwili, gdy klient prosi o przesłanie bezpiecznego zasobu przechowywanego na witrynie. To naprawdę jest BARDZO proste.

### Z perspektywy HTTP...



1. Przeglądarka przesyła żądanie zasobu o nazwie „aktualizacja.jsp”.
2. Serwer ustala, że „aktualizacja.jsp” jest zasobem chronionym, do którego dostęp jest ograniczony.
3. Kontener przesyła odpowiedź HTTP o kodzie 401 („brak upoważnienia”) wraz z nagłówkiem www-authenticate oraz informacjami o domenie bezpieczeństwa.

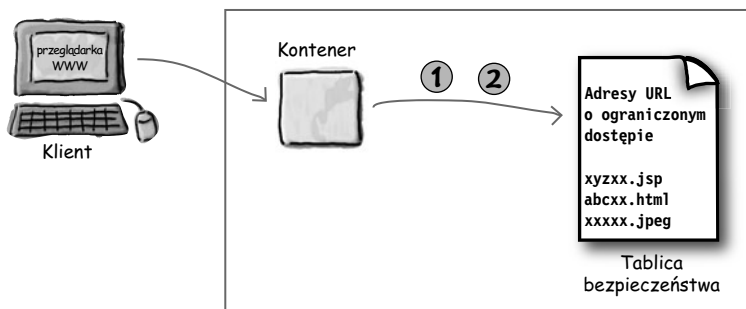


4. Przeglądarka odbiera odpowiedź z kodem 401 i (na podstawie otrzymanych informacji o domenie bezpieczeństwa) prosi użytkownika o podanie nazwy i hasła.
5. Przeglądarka ponownie prosi o dostęp do zasobu „aktualizacja.jsp” (pamiętaj, że protokół HTTP jest bezstanowy), jednak tym razem żądanie zawiera odpowiedni nagłówek HTTP pozwalający na uwierzytelnienie żądania oraz nazwę i hasło użytkownika.
6. Kontener sprawdza, czy podano poprawną nazwę użytkownika i hasło, a jeśli wszystko się zgadza, to przeprowadza uwierzytelnienie użytkownika.
7. Jeśli wszystkie sprawy związane z bezpieczeństwem są w porządku, to kontener zwraca kod HTML; w przeciwnym razie zwracana jest kolejna odpowiedź 401.

## Nieco dokładniejsza analiza sposobu, w jaki kontener przeprowadza uwierzytelnianie i autoryzację

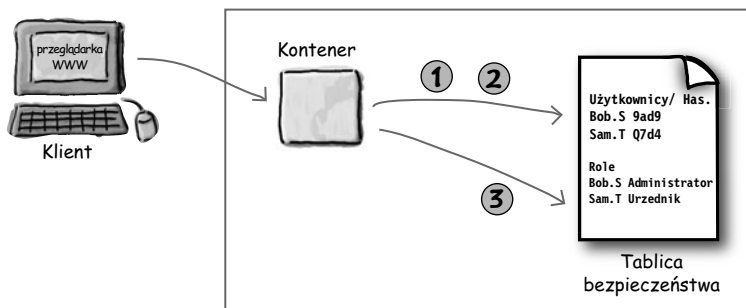
Na poprzedniej stronie praktycznie tylko wspomnieliśmy o operacjach wykonywanych przez kontener. W niniejszym rozdziale będziemy przedstawiać informacje na różnym poziomie szczegółowości... Tym razem nieco dokładniej przyjrzymy się szczegółom.

### Z perspektywy HTTP...



Żądanie początkowe, BRAK hasła

1. Po otrzymaniu żądania kontener odnajduje adres URL w „tablicy bezpieczeństwa” (przechowywanej w miejscu, w którym kontener gromadzi informacje dotyczące bezpieczeństwa i zabezpieczeń).
2. Jeśli kontener znajdzie adres URL w tablicy bezpieczeństwa, to sprawdza, czy dostęp do żadanego zasobu jest ograniczony. Jeśli jest, generuje kod 401.



Drugie żądanie, zawiera podane hasło

1. Kiedy kontener otrzymuje żądanie zawierające nazwę użytkownika oraz hasło, sprawdza adres URL zapisany w tablicy bezpieczeństwa.
2. Jeśli adres zostanie odnaleziony w tablicy (i okaże się, że zasób jest chroniony), kontener sprawdzi nazwę i hasło użytkownika, aby przekonać się, czy są poprawne.
3. Jeśli nazwa użytkownika oraz hasło są poprawne, to kontener sprawdza, czy danemu użytkownikowi przypisano odpowiednią „rolę” niezbędną do uzyskania dostępu do zasobu (czyli przeprowadza autoryzację). Jeśli tak, żądany zasób jest przekazywany do klienta.

### Jak robi to kontener?

Właśnie zdobyłeś ogólne informacje na temat sposobu, w jaki kontener przeprowadza uwierzytelnianie i autoryzację. Jednak jakie konkretne czynności kontener wykonuje w tym celu? Zabawmy się i spróbujmy odgadnąć, co tak naprawdę się dzieje we wnętrzu kontenera...

#### Operacje wykonane przez kontener:

##### ❶ *Odszukanie* żadanego zasobu

Wiemy już, że kontener naprawdę doskonale radzi sobie ze znajdowaniem zasobów. Jednak teraz, kiedy zasób został już znaleziony, kontener musi określić, czy dany zasób jest dostępny dla *każdego*, czy raczej dostęp do tego zasobu podlega jakimś *ograniczeniom*. Czy może sam serwet ma jakiś rodzaj flagi bezpieczeństwa? A może jest gdzieś jakaś tablica?

##### ❷ Przeprowadzenie *uwierzytelnienia*

Kiedy kontener określi, że ma do czynienia z zasobem chronionym, musi *uwierzytelić* klienta. Innymi słowy, kontener musi sprawdzić, czy „Bob” naprawdę *jest* Bobem. (Najczęściej spotykanym rozwiązaniem jest sprawdzenie, czy Bob zna swoje hasło).

##### ❸ Przeprowadzenie *autoryzacji*

Kiedy kontener upewni się, że to naprawdę Bob prosi o wyświetlenie chronionego zasobu, to kolejną czynnością, jaką musi wykonać, jest sprawdzenie, czy Bob ma do tego *prawo*. Zastanówmy się. Jeśli mamy 2 miliony użytkowników, a naszą aplikację tworzy 100 serwetów, to moglibyśmy utworzyć miłą tablicę składającą się z 200 milionów komórek...

O rany! To może przysporzyć poważnych problemów, jeśli w pośpiechu staniemy się nieuwinni i popełnimy błąd.

Poświęcam bardzo DUŻO cykli na sprawy związane z bezpieczeństwem! Każda rzecz, która może sprawić, że operacje związane z zapewnianiem bezpieczeństwa będą efektywniejsze, może się przyczynić do poprawy wydajności działania.



Serwer



## WYTEŻ UMYSŁ

**Jakie operacje związane z logiką zabezpieczania oraz związane z nimi informacje powinny być podane na stałe w serwlecie?**

Nazwy użytkowników i hasła?

Role poszczególnych użytkowników?

Reguły dostępu do poszczególnych serwetów?

## Nie łącz kodu i informacji związanych z bezpieczeństwem!

W zdecydowanej większości aplikacji internetowych i w przeważającej liczbie przypadków ograniczenia związane z bezpieczeństwem aplikacji powinny być obsługiwane w sposób *deklaratywny* — w deskrypcji wdrożenia. Dlaczego?

### Dziesięć najważniejszych powodów przemawiających za deklaratywnym określeniem bezpieczeństwa

- 10 *Któż z nas nie potrzebuje więcej praktyki w pracy z dokumentami w języku XML?*
- 9 Często można w naturalny sposób odwzorować stanowiska zajmowane przez poszczególne osoby w dziale informatycznym firmy.
- 8 To świetnie wygląda w życiorysie zawodowym.
- 7 Pozwala na bardziej elastyczne wykorzystywanie już napisanych serwetów.
- 6 Te zagadnienia pojawiają się na egzaminie.
- 5 Dzięki temu programiści zajmujący się tworzeniem aplikacji mogą wielokrotnie używać serwetów bez konieczności dostępu do ich kodu źródłowego.
- 4 Bo tak jest fajnie!
- 3 Takie rozwiązanie pozwala ograniczyć koszty utrzymania rozwijanej aplikacji.
- 2 Jest to czynnik, którym można wytłumaczyć i usprawiedliwić wysoką cenę kontenera.
- 1 Takie rozwiązanie jest zgodne z ideą programowania z wykorzystaniem komponentów.

## Kto zajmuje się zabezpieczeniem aplikacji internetowej?

Moje zadanie jest bardzo łatwe. W większości przypadków nawet nie muszę myśleć o sprawach związanych z bezpieczeństwem. I dobrze! To mi odpowiada, gdyż wyznaję zasadę: „Zabezpieczanie aplikacji jest trudne... i dlatego należy go unikać”.



Kim — twórca serwletów

Moje zadanie jest bardziej wymagające. To ja decyduję, jakie role mają sens w danej aplikacji. W przypadku aplikacji „piwnej” Kima kluczowymi rolami są: Gość, Członek oraz Administrator. Ja dodaję te role do informacji o użytkownikach zapisanych w specjalnym pliku na naszym kontenerze. Ponieważ używamy Tomcata, nasz plik użytkowników nosi nazwę tomcat-users.xml.



Annie  
— administratorka aplikacji

Moje zadanie jest niezwykle ważne i odpowiedzialne! Kiedy dostaję sporządzoną przez Annie listę ról oraz opis czynności wykonywanych przez serwlety napisane przez Kima, decyduję, które role mają mieć dostęp do poszczególnych serwletów. Deskryptor wdrożenia zapewnia mi prosty, choć nieco rozwlekły sposób informowania kontenera, kto ma mieć dostęp do poszczególnych serwletów. A, i jeszcze jedno — zaoferowana przez Ciebie suma nie była wystarczająca...



Dick — wdrożeniowiec

## Nie ma niemądrych pytań

**P.** Nie łapię — czy jeśli tworzę serwlet, to w ogóle nie muszę zwracać uwagi na zagrożenia związane z bezpieczeństwem?

**U.** Ależ skąd — powinieneś. Kim, twórca serwletów, wyraził się nieco sarkastycznie. W przypadku tworzenia serwletów kluczowym zagadnieniem jest ich modularność. Na przykład ze wszech miar sensownym rozwiązaniem jest rozdzielenie możliwości przeglądania jakichś informacji od możliwości ich aktualizowania. Jeśli te dwa przypadki użycia zostaną zaimplementowane jako niezależne serwlety, to wdrożeniowiec będzie miał łatwiejsze zadanie podczas określania różnych ograniczeń dostępu związanych z korzystaniem z tych możliwości.

**P.** Nie wiem, gdzie pracujecie, ale w mojej sytuacji muszę się zajmować wszystkimi trzema zagadnieniami — tworzeniem kodu, zarządzaniem aplikacją i wdrażaniem jej.

**U.** W rzeczywistości to jest bardzo często spotykana sytuacja. Jednak pomimo to zalecamy, aby w przypadku implementacji mechanizmów bezpieczeństwa, robić to *etapami* i „wyobrazać sobie”, że za każdym razem mamy zupełnie inne obowiązki.

**P.** A co z zabezpieczeniami programowymi? Czy rzeczywiście pasują do tego schematu zabezpieczania aplikacji internetowych?

**U.** Tymi zagadnieniami zajmiemy się w dalszej części rozdziału. Na razie wystarczy, byś wiedział, że blisko 95 procent wszystkich czynności związanych z zabezpieczaniem serwletów można realizować w sposób *deklaracyjny*. Mechanizmy programowe nie są stosowane zbyt często (patrz „Dziesięć najważniejszych powodów...”).

**P.** Jak na razie wszystkie podane informacje są związane z uwierzytelnianiem i autoryzacją, a co z pozostałymi dwoma zagadnieniami wchodzącymi w skład „wielkiej czwórki”?

**U.** Sprawami związanymi z *poufnością* i *integralnością* danych zajmiemy się w dalszej części rozdziału. Specyfikacja serwletów sprawia, iż zagrożenia te są bardzo łatwe, dlatego też koncentrujemy się tutaj na uwierzytelnianiu i autoryzacji, gdyż są one znacznie bardziej złożone i trudne do implementacji, jak również dlatego, że znacznie częściej pojawiają się na egzaminie (uwaga — to jest podpowiedź!).

**P.** Wydaje mi się, że kiedy mowa o bezpieczeństwie serwletów, to określenie „rola” jest, rzekłbym, „przeciążone”...

**U.** Słuszna uwaga! Specyfikacje różnych technologii wchodzących w skład platformy J2EE (EJB, serwlety, JSP) często uwzględniają „rodzaje” osób, które mogą się zajmować *tworzeniem* i *zarządzaniem* komponentami. Rodzaje te odpowiadają **rolom** różnych pracowników działu informatycznego. Z kolei programiści zajmujący się zagadnieniami bezpieczeństwa aplikacji internetowych wyróżniają raczej istniejące **typy użytkowników**. Na przykład „gość” może mieć bardzo ograniczone uprawnienia do korzystania z aplikacji. Te „role użytkowników” można definiować i odzworowywać w deskryptorze wdrożenia.

**P.** Słyszałem o czymś określanym terminem „cross-site hacking”, co to takiego?

**U.** To technika, którą można zastosować w przypadku, gdy witryna wyświetla teksty wpisywane w formularzu przez użytkowników (na przykład wpisy w księdze gości). Jeśli wrogo nastawiony użytkownik wpisze w polu kod HTML połączony, na przykład, z kodem JavaScript, a serwer tego nie wychwyci, to niczego nieświadoma przeglądarka wykona ten potencjalnie *niebezpieczny* kod wraz z normalnym i *poprawnym* kodem strony. Innymi słowy, serwer przesyła do przeglądarki użytkownika kod napisany przez *innego* użytkownika bez sprawdzenia, czy nie zawiera on niebezpiecznego kodu skryptowego.

**P.** A zatem musimy posługiwać się mechanizmami zabezpieczeń wchodzącymi w skład „wielkiej czwórki”. Jak trudno jest skonfigurować i utrzymywać te cuda? Chodzi mi o to, czy to jest... bolesne?

**U.** Owszem... obawiamy się, że to może boleć... *trochę*. W rzeczywistości *niektóre* aspekty zabezpieczania aplikacji są bardzo proste, natomiast inne faktycznie mogą być pracochłonne. Na szczęście korzystanie z tych mechanizmów zabezpieczeń nigdy nie jest szczególnie skomplikowane, może być jedynie nieco nużące.

# Najważniejsze zadania związane z zabezpieczaniem serwletów

Poniższa tabela pozwoli Ci zrozumieć podstawowe aspekty zabezpieczania serwletów. Najbardziej czasochłonną czynnością jest implementacja **autoryzacji**, a następnie **uwierzytelniania**. Z punktu widzenia serwletów konfiguracja poufności oraz integralności danych jest bardzo prosta\*.

Pojęcie	Kto jest odpowiedzialny?	Poziom złożoności	Niezbędny wkład pracy	Znaczenie na egzaminie
Uwierzytelnianie	Administrator	Średni	Wysoki	Średnie
Autoryzacja	Wdrożeniowiec (przeważnie)	Wysoki	Wysoki	Wysokie
Poufność	Wdrożeniowiec	Niski	Niski	Niskie
Integralność danych	Wdrożeniowiec	Niski	Niski	Niskie

W niniejszym rozdziale najwięcej uwagi poświęcimy zagadnieniom autoryzacji, gdyż są one najważniejsze i najbardziej złożone spośród wszystkich zagadnień bezpieczeństwa niezależnych od producenta używanego serwera.

\* Prawdę mówiąc, zdobycie certyfikatu SSL wcale nie jest zadaniem trywialnym, a zatem w tym przypadku „proste” oznacza — „nie trzeba niczego dodawać w kodzie serwletu”.

## Co trzeba wiedzieć o uwierzelnianiu, aby móc zajmować się autoryzacją...

W dalszej części rozdziału bardziej szczegółowo zajmiemy się zagadnieniami uwierzelniania, jednak na razie skoncentrujemy się na ocenie informacji na temat *uwierzelniania* niezbędnych do prawidłowego funkcjonowania mechanizmów *autoryzacji*. Nie można bowiem przeprowadzić *autoryzacji* użytkownika, póki nie zostanie on *uwierzelniony*.

Specyfikacja serwletów nie podaje, *jak* kontener powinien implementować obsługę danych służących do uwierzelniania, w tym nazw użytkowników i haseł. Ogólna koncepcja sprowadza się do udostępniania przez kontener jakiejś (zależnej od producenta) tablicy reprezentującej nazwy użytkowników, ich hasła oraz role skojarzone z poszczególnymi użytkownikami. Niemniej jednak niemal wszyscy dostawcy kontenerów idą znacznie dalej i dają możliwość korzystania z korporacyjnych informacji uwierzelniających, które bardzo często są przechowywane w relacyjnych bazach danych lub systemach LDAP (których prezentacja wykracza poza ramy tematyczne niniejszej książki). Zazwyczaj informacje używane podczas uwierzelniania użytkowników są podawane i utrzymywane przez administratora.

### „Dziedziny” bezpieczeństwa

Niestety, **dziedzina** to kolejny termin związany z zagadnieniami bezpieczeństwa, którego znaczenie zostało „przeciążone”. Z punktu widzenia specyfikacji serwletów, **dziedzina** to miejsce, w którym są przechowywane informacje **uwierzelniające**. W przypadku kontenera Tomcat tę funkcję pełni plik nazwany *tomcat-users.xml* (przechowywany w katalogu *conf*, NIE w katalogu *webapps*). Ten jeden plik zawiera dane użytkowników stosowane we WSZYSTKICH aplikacjach umieszczonych w katalogu *webapps*. Jest on powszechnie określany jako **dziedzina pamięci**, gdyż podczas uruchamiania Tomcat wczytuje zawartość tego pliku do pamięci. Choć takie rozwiązanie doskonale nadaje się do testowania, to jednak nie zaleca się stosowania go w serwerach produkcyjnych. Przede wszystkim chodzi o to, że w celu aktualizacji informacji o użytkownikach konieczne jest ponowne uruchamianie Tomcata.

#### Plik *tomcat-users.xml*

```
<tomcat-users>
 <role rolename="Gosc" />
 <role rolename="Czlonek" />
 <user username="Janek" password="j23nadaje" roles="Gosc, Czlonek" />
</tomcat-users>
```

Używany przez Ciebie serwer aplikacji będzie zapewne potrzebował innych informacji podanych w innej postaci, jednak będzie on pozwalał na skojarzenie nazw użytkowników, ich haseł oraz ról.

Za przeprowadzanie uwierzelniania odpowiada jakaś struktura danych podobna do tej. W przypadku Tomcata można używać pliku o nazwie „*tomcat-users.xml*” zawierającego nazwy, hasła i role użytkowników, których kontener będzie używać podczas przeprowadzania uwierzelniania.

Pamiętaj! Te informacje NIE są częścią deskryptora wdrożenia, ich postać zależy wyłącznie od używanego serwera!

### Włączanie uwierzelniania

Aby rozpocząć korzystanie z uwierzelniania (innymi słowy, aby kontener zaczął pytać użytkowników o ich nazwy i hasła), trzeba umieścić odpowiednią informację w deskrytorze wdrożenia. Nie przejmuj się na razie wszelkimi szczegółami i znaczeniem kodu włączającego uwierzelnianie, a jeśli chcesz wypróbować te mechanizmy, to po prostu umieść w deskrytorze wdrożenia poniższy fragment kodu:

```
<login-config>
 <auth-method>BASIC</auth-method>
</login-config>
```

Znaczenie tego kodu opiszemy szczegółowo w dalszej części rozdziału — na razie wystarczy, że umieścisz przedstawiony element w swoim deskrytorze wdrożenia, aby włączyć mechanizm uwierzelniania użytkowników.

### Pierwszy etap autoryzacji — definiowanie ról

Najczęściej stosowaną metodą autoryzacji dostępu do serwletów jest sprawdzanie, czy „role” bezpieczeństwa przypisane konkretnemu użytkownikowi pozwalają mu na odwoływanie się do danego serwletu (przy użyciu zastosowanej metody żądania HTTP). Oczywiście za wykonanie tego sprawdzenia odpowiada kontener, a zatem pierwszym krokiem jest skojarzenie ról podanych w *pliku* „użytkowników” używanym przez dany serwer z rolami zdefiniowanymi w *deskrytorze wdrożenia*.

Annie to "Administrator",  
"Członek" oraz "Gosc".



### ZAŁĘŻNE OD SERWERA:

#### Element <role> używany w pliku tomcat-users.xml

Struktura danych zawierająca nazwy użytkowników i ich hasła, zależna od używanego serwera.

```
<tomcat-users>
 <role rolename="Administrator" />
 <role rolename="Członek" />
 <role rolename="Gosc" />
 <user username="Annie" password="admin" roles="Administrator, Członek, Gosc" />
 <user username="Diane" password="koder" roles="Członek, Gosc" />
 <user username="Ted" password="wedrowniczek" roles="Gosc" />
</tomcat-users>
```

Diane to "Członek"  
oraz "Gosc"

Ted to "Gosc"

W przypadku Tomcata używany plik tomcat-users.xml powinien wyglądać podobnie jak przedstawiony w tym przykładzie. Zwróć uwagę na możliwość kojarzenia wielu ról z jednym użytkownikiem

### SPECYFIKACJA SERWLETÓW:

#### Element <security-role> w deskrytorze wdrożenia (web.xml):

```
<security-role><role-name>Administrator</role-name></security-role>
<security-role><role-name>Członek</role-name></security-role>
<security-role><role-name>Gosc</role-name></security-role>
```

```
<login-config>
 <auth-method>BASIC</auth-method>
</login-config>
```

Pamiętaj, że aby korzystać z uwierzytelniania, musisz umieścić w deskrytorze wdrożenia element <login-config>.

Kiedy kontener stanie przed koniecznością przeprowadzenia autoryzacji, skojarzy posiadane informacje o użytkownikach i przypisanych im rolach z elementami <role-name> umieszczonymi w deskrytorze wdrożenia wewnątrz elementów <security-role>.

Elementy <role-name> w deskrytorze wdrożenia podaje wdrożeniowiec; dzięki nim kontener może skojarzyć role z poszczególnymi użytkownikami.

## Drugi etap autoryzacji — definiowanie ograniczeń w dostępie do zasobów i metod

W końcu coś ciekawego. Wreszcie doszliśmy do miejsca, w którym podaje się (w sposób *deklaratywny*), że dana kombinacja zasobu i metody żądania jest dostępna wyłącznie dla użytkowników o odpowiednich *rolach*. Definiując zabezpieczenia aplikacji, zazwyczaj będziemy tworzyli właśnie elementy `<security-constraint>` umieszczane w deskrypcorze wdrożenia. (Więcej szczegółów podamy w dalszej części rozdziału).

### Element `<security-constraint>` w deskrypcorze wdrożenia:

```
<web-app>
...
<security-constraint>
 <web-resource-collection>
 <web-resource-name>AktualizacjaPrzepisow</web-resource-name>

 <url-pattern>/Piwo/DodajPrzepis/*</url-pattern>
 <url-pattern>/Piwo/WyswietlPrzepis/*</url-pattern>

 <http-method>GET</http-method>
 <http-method>POST</http-method>
 </web-resource-collection>
 <auth-constraint>
 <role-name>Administrator</role-name>
 <role-name>Czlonек</role-name>
 </auth-constraint>
</security-constraint>
</web-app>
```

To obowiązkowa nazwa wykorzystywana przez narzędzia. Nie zobaczysz jej nigdzie indziej...

Element `<url-pattern>` definiuje zasoby, do których dostęp chcemy OGRANICZYĆ.

Element(y) `<http-method>` definiuje metody HTTP, których wolno używać podczas odwotywnia się do zasobów zdefiniowanych przy użyciu podanych wcześniej wzorców adresów URL.

Opcjonalny element `<auth-constraint>` określa, jakie role mogą używać wymienionych wcześniej metod HTTP. Innymi słowy, określa on, KTO może używać żądań GET oraz POST i odwotywniać się do zasobów o określonych wzorcach adresów URL.

Każda z nas może używać metod GET i POST do odwotywnia się do zasobów umieszczonych w katalogach `/Piwo/DodajPrzepis` oraz `/Piwo/WyswietlPrzepis`.



Administrator



A to pech! Moja rola (Gosc) nie została podana w elemencie `<auth-constraint>`, a zatem nie mogę używać ani metody GET, ani POST w swoich odwotywniach do zasobów umieszczonych w tych katalogach. Jednak mogę używać metod TRACE, HEAD oraz PUT...



Gosc

Ponieważ Annie i Diane mają role "Czlonек", mogą odwotywniać się do zasobów, których adresy odpowiadają wzorcom podanym w elementach `<url-pattern>` przy użyciu metod GET i POST. Z drugiej strony, Ted, który jest tylko „gościem”, nie może używać tych metod.

## Reguły ograniczeń <security-constraint> dla elementów <web-resource-collection>

Pamiętaj — przeznaczeniem podelementu <web-resource-collection> jest wskazanie kontenerowi kombinacji zasobów i metod protokołu HTTP, do których dostęp należy *ograniczać* według ról zdefiniowanych w odpowiednim znaczniku <auth-constraint>. Chcielibyśmy móc powiedzieć Ci w tym momencie, że możesz się zrelaksować, jednak naprawdę musisz znać szczegółowe informacje związane ze stosowaniem tego elementu. Jeśli popełnisz choć jeden drobny błąd w części deskryptora wdrożenia odpowiedzialnej za zabezpieczenie aplikacji, może się okazać, że jej najbardziej wrażliwe i ważne składowe będą dostępne... *dla każdego*.

### Podelement <web-resource-collection> elementu <security-constraint>

```
<web-app>
...
<security-constraint>
 <web-resource-collection>

 <web-resource-name>
 AktualizacjaPrzepisow
 </web-resource-name>
 <url-pattern>/Piwo/DodajPrzepis/*</url-pattern>
 <url-pattern>/Piwo/WyswietlPrzepis/*</url-pattern>

 <http-method>GET</http-method>

 </web-resource-collection>

 <auth-constraint>
 ...
 </auth-constraint>
</security-constraint>
</web-app>
```

To są katalogi z ograniczonym dostępem.

W ten sposób określamy, że z metody GET mogą korzystać TYLKO role zdefiniowane w elemencie <auth-constraint>. Ponieważ jednak dla POZOSTAŁYCH metod nie zdefiniowano żadnych ograniczeń, mogą z nich korzystać wszyscy użytkownicy.

### Kluczowe zagadnienia dotyczące elementu <web-resource-collection>

- Element <web-resource-collection> zawiera dwa główne podelementy: <url-pattern> (jeden lub kilka) oraz <http-method> (opcjonalny; zero, jeden lub wiele).
- Wzorce URL oraz metody HTTP łącznie określają, które żądania podlegają **ograniczeniom** i jako takie mogą być skutecznie realizowane przez role wskazane w elemencie <auth-constraint>.
- Element <web-resource-name> jest WYMAGANY (choć prawdopodobnie nigdy nie będziesz go używać). (Przyjmij, że stworzono go z myślą o środowiskach IDE lub przyszłych zastosowaniach).
- Element <description> jest OPCJONALNY.
- W elementach <url-pattern> stosuje się standardową konwencję nazewnictwa i reguły odwzorowywania serwetów (więcej informacji na temat wzorców adresów URL można znaleźć w rozdziale poświęconym wdrażaniu aplikacji internetowych).
- **Musisz określić przynajmniej jeden element <url-pattern>**, ale *możesz* też podać ich więcej.
- Poprawne metody, które można podawać w elemencie <http-method>, to: GET, POST, PUT, TRACE, DELETE, HEAD oraz OPTIONS.
- Jeśli nie wskażesz *żadnej* metody protokołu HTTP, danemu ograniczeniu będą podlegały **WSZYSTKIE** (zatem odpowiedni zasób będzie dostępny wyłącznie dla ról opisanych w elemencie <auth-constraint>).
- Jeśli wskażesz jakieś metody przy użyciu elementów <http-method>, to tylko one będą podlegać ograniczeniom. Innymi słowy, podanie choćby jednej metody automatycznie sprawia, że *żadne* z pozostałych metod nie będą podlegać ograniczeniom.
- W jednym elemencie <security-constraint> można umieścić wiele elementów <web-resource-collection>.
- Element <auth-constraint> odnosi się do WSZYSTKICH elementów <web-resource-collection> umieszczonych wewnątrz elementu <security-constraint>.



Oglądaj to!

**Ograniczenia nie obowiązują na poziomie ZASOBU.  
Ograniczenia obowiązują na poziomie ŻĄDANIA HTTP.**

Można by pomyśleć, że ograniczany jest dostęp do samych zasobów. Jednak w rzeczywistości ograniczeniom podlega kombinacja zasobu oraz użytej metody HTTP. Stwierdzając: „dostęp do tego zasobu jest ograniczany”, tak naprawdę stwierdzasz: „dostęp do tego zasobu jest ograniczany w przypadku użycia metody GET protokołu HTTP”. Dostęp do zasobów jest zawsze ograniczany na podstawie stosowanej metody HTTP, choć MOŻNA skonfigurować element `<web-resource-collection>` w taki sposób, aby ograniczenia dotyczyły WSZYSTKICH metod; wystarczy w tym celu nie podawać w elemencie `<web-resource-collection>` żadnego elementu `<http-method>`.

Element `<auth-constraint>` NIE definiuje, jakie role mają prawo dostępu do zasobów wskazanych w elemencie `<web-resource-collection>`. Definiuje on role, które mogą wykonać **żądanie podlegające ograniczeniom**. Nie należy wyobrażać sobie tego procesu jako: „Bob jest Członkiem, zatem może się odwołać do serwletu DodajPrzepis”. Pomyśl: „Bob jest Członkiem, zatem może odwoływać się do serwletu DodajPrzepis przy użyciu metod GET i POST”.



Oglądaj to!

**Jeśli definiujesz element `<http-method>`, żadna z NIEwymienionych metod protokołu HTTP nie będzie podlegała ograniczeniom.**

Zadaniem serwera WWW jest UDOSTĘPNIANIE, a zatem domyślnie zakładamy, że chcemy, by żadna metoda HTTP nie podlegała ograniczeniom, chyba że wprost (za pomocą elementu `<http-method>`) zażądamy ograniczenia możliwości stosowania określonej metody w dostępie do konkretnego zasobu (wskazanego przy użyciu elementu `<url-pattern>`). Jeśli, definiując ograniczenia, podamy jedynie element `<http-method>GET</http-method>`, to żądania wykorzystujące wszelkie pozostałe metody protokołu HTTP — POST, HEAD, PUT itd. — nie będą w żaden sposób ograniczane! A to oznacza, że każdy, niezależnie od przypisanej mu roli (a nawet niezależnie od tego, czy przeprowadzono uwierzytelnianie) będzie mógł korzystać z tych metod HTTP.

To wszystko prawda, ALE TYLKO pod warunkiem, że został podany przynajmniej jeden element `<http-method>`. Jeśli żaden taki element NIE został podany, to ograniczenia będą dotyczyć WSZYSTKICH metod HTTP. (Jednak prawdopodobnie nigdy nie będziesz tak postępował, gdyż cała idea zabezpieczania polega na ograniczaniu konkretnych żądań HTTP dotyczących konkretnych zasobów).

Oczywiście żadne metody HTTP nie będą działać, jeśli w serwletach nie przesłonisz odpowiednich metod `doXXX()`. A zatem, jeśli w serwlecie zaimplementowałeś jedynie metodę `doGet()`, a w deskrytorze podałeś wyłącznie element `<http-method>` dla metody GET, to i tak nikt nie będzie mógł wykonać żądania wykorzystującego metodę POST, gdyż serwer nie będzie wiedzieć, jak należy ją obsłużyć.

A zatem możemy nieco zmodyfikować wcześniejszą zasadę i stwierdzić, iż każda metoda HTTP obsługiwana przez serwlet (dzięki temu, że została przesłonięta odpowiednia metoda serwletu) będzie mogła być używana, CHYBA ŻE:

1. W elemencie `<security-constraint>` nie podasz żadnego elementu `<http-method>`, co będzie oznaczać, że ograniczenia dotyczą wszystkich metod protokołu HTTP.
2. Wprost określisz metodę za pomocą elementu `<http-method>`. Pamiętaj, że użycie choćby jednego elementu `<http-method>` w definicji ograniczenia automatycznie wyklucza stosowanie tego ograniczenia dla pozostałych metod protokołu HTTP.

# Zaskakujące reguły ograniczeń <security-constraint> dla podelementu <auth-constraint>

Choć fragment — constraint nazwy elementu sugeruje, że ma on coś wspólnego z ograniczaniem, to jednak element ten wskazuje role, które MAJĄ PRAWO dostępu do zasobów aplikacji wskazanych w elemencie <web-resource-collection>.

## Podelement <auth-constraint> elementu <security-constraint>

```
<web-app>
```

```
...
```

```
<security-constraint>
```

```
<web-resource-collection>
```

```
...
```

```
</web-resource-collection>
```

```
<auth-constraint>
```

```
<role-name>Administrator</role-name>
```

```
<role-name>Członek</role-name>
```

```
</auth-constraint>
```

```
</security-constraint>
```

```
</web-app>
```

W ten sposób określamy, że Administrator i Członek mają prawo do korzystania z kombinacji zasobów i metod HTTP wskazanych w powyższym elemencie <web-resource-collection>. Jednak „Gosc” nie został wymieniony w elemencie <auth-constraint>, zatem osoby należące do tej roli nie mają prawa wykonywania żądań podlegających ograniczeniom.

### Reguły stosowania elementu <role-name>

- Umieszczanie elementu <role-name> wewnątrz elementu <auth-constraint> jest OPCJONALNE.
- Jeśli zdefiniowano jakieś elementy <role-name>, każdy z nich określa rolę, która MA PRAWO przesyłania żądań.
- Jeśli wewnątrz elementu <auth-constraint> NIE zdefiniowano ŻADNYCH elementów <role-name>, ŻADEN UŻYTKOWNIK NIE MA PRAWA DOSTĘPU.
- Jeśli użyto elementu <role-name>\*</role-name>, oznacza to, że WSZYSCY użytkownicy MAJĄ PRAWO dostępu.
- Wielkość liter w nazwach ról **ma znaczenie**.

### Reguły stosowania elementu <auth-constraint>

- Umieszczanie elementu <auth-constraint> wewnątrz elementu <security-constraint> jest OPCJONALNE.
- Jeśli zdefiniowano jakiś element <auth-constraint>, kontener MUSI przeprowadzać uwierzytelnianie dla wskazanych adresów URL.
- Jeśli element <auth-constraint> NIE istnieje, to kontener MUSI zezwolić na nieuwierzytelniany dostęp do wskazanych zasobów.
- Dla przejrzystości warto w elemencie <auth-constraint> umieścić podelement <description>.

# Sposób działania elementu < auth-constraint >



Zawartość elementu <auth-constraint>	Role, które mają dostęp do zasobów	
<pre>&lt;security-constraint&gt;   &lt;auth-constraint&gt;     &lt;role-name&gt;Administrator&lt;/role-name&gt;     &lt;role-name&gt;Członek&lt;/role-name&gt;   &lt;/auth-constraint&gt; &lt;/security-constraint&gt;</pre>	Administrator Członek	
<pre>&lt;security-constraint&gt;   &lt;auth-constraint&gt;     &lt;role-name&gt;Gosc&lt;/role-name&gt;   &lt;/auth-constraint&gt; &lt;/security-constraint&gt;</pre>	Gosc	
<pre>&lt;security-constraint&gt;   &lt;auth-constraint&gt;     &lt;role-name&gt;*&lt;/role-name&gt;   &lt;/auth-constraint&gt; &lt;/security-constraint&gt;</pre>	Wszyscy	
Brak elementu <auth-constraint>	Wszyscy	
<pre>&lt;security-constraint&gt;   &lt;auth-constraint /&gt; &lt;/security-constraint&gt;</pre>	Nikt	

Te dwa elementy mają  
takie SAMO znaczenie.



Oglądaj to!

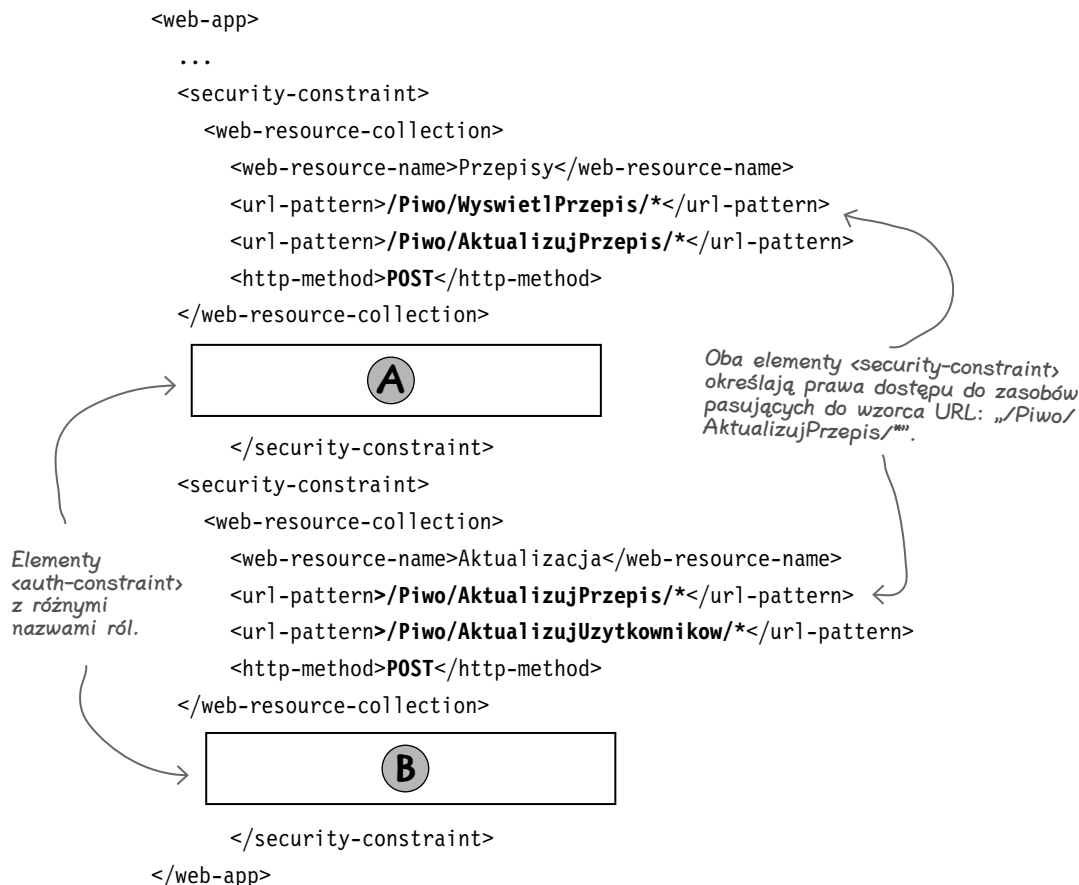
**BRAK elementu <auth-constraint> jest przeciwieństwem pustego elementu <auth-constraint/>!**

Zapamiętaj to dobrze: Jeśli nie podasz ról, które podlegają ograniczeniom, to **ŻADNE** z nich nie będą ograniczane. Jeśli jednak UŻYJESZ elementu <auth-constraint>, to JEDYNIE te role, które zostały w nim podane jawnie, będą miały prawo dostępu (no, chyba że w elemencie <auth-name> umieścisz gwiazdkę „\*”). Jeśli nie chcesz, by jakakolwiek rola miała dostęp do danego zasobu, MUSISZ użyć pustego elementu: <auth-constraint/>. Przekaze on kontenerowi wiadomość: „Mówię wprost, które role mają prawo dostępu, a w tym przypadku nie ma takich ról!”.

## W jaki sposób różne elementy `<security-constraint>` mogą na siebie oddziaływać?

Właśnie w chwili, gdy pomyślałeś, że już zrozumiałeś działanie elementu `<security-constraint>`, zdałeś sobie sprawę, iż jeśli w deskrytorze umieścisz *kilka* takich elementów, to mogą pomiędzy nimi występować konflikty. Przyjrzyj się deskrytorowi podanemu poniżej i wyobraź sobie różne konfiguracje zabezpieczeń, które można by w nich zdefiniować. Co się stanie, jeśli na przykład jeden element `<security-constraint>` *zabrania* dostępu, a inny element *zezwała* tej samej roli na dostęp do... tego samego zasobu? Który element `<security-constraint>` zwycięży w tej rywalizacji? Odpowiedź znajdziesz w tabeli zamieszczonej na kolejnej stronie.





**Kilka elementów `<security-constraint>` zawierających te same (lub częściowo pokrywające się) wzorce URL i elementy `<http-method>`:**



**W jaki sposób kontener ma przeprowadzić uwierzytelnianie, skoro prawa dostępu do tego samego zasobu zostały określone w więcej niż jednym elemencie `<security-constraint>`?**

## Konkurujące ze sobą elementy <auth-constraint>

Poniżej opisano sposób, w jaki kontener określa wynikowe prawa dostępu do zasobów, dla których zdefiniowano dwa lub więcej elementów <security-constraint> z częściowo lub całkowicie pokrywającymi się elementami <web-resource-collection>. Oznaczenia A oraz B odnoszą się do brakujących fragmentów deskryptora wdrożenia przedstawionego na poprzedniej stronie.

Zawartość fragmentu <span>A</span>	Zawartość fragmentu <span>B</span>	Kto ma dostęp do katalogu „AktualizujPrzepis”
1 <code>&lt;auth-constraint&gt;   &lt;role-name&gt;Gosc&lt;/role-name&gt; &lt;/auth-constraint&gt;</code>	<code>&lt;auth-constraint&gt;   &lt;role-name&gt;Administrator&lt;/role-name&gt; &lt;/auth-constraint&gt;</code>	Role Gosc oraz Administrator 
2 <code>&lt;auth-constraint&gt;   &lt;role-name&gt;Gosc&lt;/role-name&gt; &lt;/auth-constraint&gt;</code>	<code>&lt;auth-constraint&gt;   &lt;role-name&gt;*&lt;/role-name&gt; &lt;/auth-constraint&gt;</code>	Wszyscy 
3 <code>&lt;auth-constraint /&gt;</code>	<code>&lt;auth-constraint&gt;   &lt;role-name&gt;Administrator&lt;/role-name&gt; &lt;/auth-constraint&gt;</code>	Nikt 
4 <b>Brak elementu &lt;auth-constraint&gt;</b>	<code>&lt;auth-constraint&gt;   &lt;role-name&gt;Administrator&lt;/role-name&gt; &lt;/auth-constraint&gt;</code>	Wszyscy 

### Reguły interpretacji powyższej tabeli:

- 1 W wyniku połączenia poszczególnych nazw ról dostęp zostanie przyznany wszystkim wskazanym rolom.
- 2 Rola o nazwie „\*” łączy się z wszelkimi innymi rolami i gwarantuje, że *wszyscy* będą mieć prawo dostępu.
- 3 Pusty znacznik <auth-constraint> łączy się ze wszystkimi innymi i sprawia, że *nikt* nie będzie dysponować prawem dostępu! Innymi słowy, ostatnie słowo zawsze należy właśnie do pustego znacznika <auth-constraint>.
- 4 Jeśli jeden z elementów <security-constraint> nie zawiera *żadnego* elementu <auth-constraint>, to łączy się on z dowolnymi innymi znacznikami i zapewnia, że *wszyscy* mają prawo dostępu.

Jeśli reguły zdefiniowane w dwóch różnych niepustych elementach <auth-constraint> odnoszą się do tego samego zasobu, to prawo dostępu do niego zostanie przydzielone sumie ról zdefiniowanych w obu elementach <auth-constraint>.

### Nie ma niemądrych pytań

**P.** Rozumiem, że umieszczenie w deskryptorze wdrożenia pustego elementu `<auth-constraint/>` informuje kontener, że **NIKT**, niezależnie od przypisanych ról, nie może uzyskać dostępu do chronionego zasobu. Jednak nie rozumiem, dlaczego **ktokolwiek** chciałby definiować takie prawo dostępu. Jaki jest pożytek z zasobu, z którego nikt nie może skorzystać?

**U.** W tym przypadku stwierdzenie „nikt” oznacza „nikt spoza aplikacji internetowej”. Innymi słowy, *klient* nie może uzyskać dostępu do zasobu, lecz inna część aplikacji *może* z niego korzystać bez żadnych ograniczeń. Być może będziesz chciał przekierować obsługę żądania do innej, chronionej części aplikacji, jednak nigdy nie będziesz chciał, by klient mógł się do niej odwoływać w sposób bezpośredni. Takie zasoby, do których nikt „z zewnątrz” nie ma prawa dostępu, można by porównać do metod prywatnych w Javie — są one przeznaczone wyłącznie do użytku wewnętrznego.

**P.** Dlaczego element `<auth-constraint>` jest umieszczany wewnątrz elementu `<security-constraint>`, a nie wewnątrz elementu `<web-resource-collection>`?

**U.** Takie rozwiązanie umożliwia nam definiowanie pojedynczego elementu `<auth-constraint>` (być może obejmującego wiele ról), by następnie wskazać wiele kolekcji zasobów, dla których należy stosować listę ról elementu `<auth-constraint>`. Można na przykład zdefiniować element `<auth-constraint>` dla roli *CzestyNabywca*, po czym umieścić w jego wnętrzu po jednym elemencie `<web-resource-collection>` dla każdego składnika aplikacji, do którego *CzestyNabywca* ma szczególne prawa dostępu.

**P.** Czy naprawdę muszę siedzieć i wpisywać każdego użytkownika mojej aplikacji, podając przy tym jego hasło i przypisane mu role?

**U.** Jeśli używasz sposobu podawania informacji o użytkownikach stosowanego w serwerze Tomcat, to owszem — musisz. Niemniej jednak istnieje szansa, że w praktyce będziesz używać jakiegoś serwera umożliwiającego połączenie LDAP lub obsługującego bazę danych, w której są przechowywane faktyczne informacje o użytkownikach.

## Serwlet przepisów Alicji — historia o programowym zapewnianiu bezpieczeństwa...

Alicja wie, że w większości przypadków deklaratywne metody zabezpieczeń są całkowicie wystarczające. Są elastyczne, skuteczne, przenośne i niezawodne. Jednak wraz z rozwojem aplikacji internetowych poszczególne serwlety stawały się coraz bardziej wyspecjalizowane. O ile kiedyś *pojedynczy* serwlet implementowałby całą logikę biznesową związaną z obsługą pracowników i kadry zarządzającej, o tyle obecnie odpowiednie funkcje dużo częściej zaimplementuje się w co najmniej dwóch odrębnych serwletach.

Na swoje szczęście Alicja odziedziczyła serwlet `SerwletPrzepisow` po kimś innym. Jednocześnie doszły ją słuchy, że serwlet ten wykorzystuje programowe mechanizmy zabezpieczeń, więc zaczęła przeglądać jego kod źródłowy w poszukiwaniu fragmentów odpowiedzialnych za bezpieczeństwo...

```
if (request.isUserInRole("Menedzer")) {
 // wykonujemy stronę AktualizujPrzepis
 ...
} else {
 // wykonujemy stronę PokazPrzepis
 ...
}
```

← Kto wymyślił słowo „Menedzer” jako nazwę roli? Może autor tego serwletu nie wiedział, jakie są role w Twojej firmie?



### Zaostrz ołówek

#### A jakie są implikacje?

Przeanalizuj to, czego się już dowiedziałeś w tym rozdziale, przyjrzyj się przedstawionemu powyżej fragmentowi kodu i spróbuj odpowiedzieć na poniższe pytania:

Jakie czynności związane z zapewnieniem bezpieczeństwa musiały mieć miejsce przed wykonaniem tego fragmentu kodu?

Jaką czynność związaną z zabezpieczaniem aplikacji reprezentuje przedstawiony fragment kodu?

Jaką rolę w realizacji powyższego fragmentu kodu odgrywa deskryptor wdrożenia?

Jak myślisz, w jaki sposób działa przedstawiony fragment kodu?

Co by się stało, gdyby rola „Menedzer” nie była dostępna?

# Dostosowywanie metod — metoda `isUserRole()`

W klasie `HttpServletRequest` dostępne są trzy metody związane z programowym zapewnianiem bezpieczeństwa. Są to:

**`getUserPrincipal()`** — metoda ta jest używana przede wszystkim w przypadku stosowania komponentów EJB. Tych zagadnień nie poruszamy w niniejszej książce.\*

**`getRemoteUser()`** — metoda, której można używać w celu sprawdzenia statusu uwierzytelniania. Jest ona używana bardzo rzadko, zatem nie będziemy jej opisywać w niniejszej książce (również na egzaminie nie pojawią się żadne pytania na jej temat).

**`isUserRole()`** — właśnie tą metodą *teraz* się zajmujemy. Zamiast autoryzowania użytkowników na poziomie żądania HTTP (GET, POST itd.) można przeprowadzać autoryzację dostępu do *fragmentów* metody. W ten sposób uzyskujemy możliwość *dostosowywania* sposobu działania metody na podstawie przynależności użytkownika do jednej z ról. Jeśli jest realizowana któraś z metod obsługi żądania (`doGet()`, `doPost()` itd.), oznacza to, że użytkownik przeszedł przez deklaratywne mechanizmy zabezpieczeń. Teraz jednak chcemy warunkowo wykonać jakieś dodatkowe operacje zabezpieczające w ramach aktualnie wykonywanej metody na podstawie informacji o tym, czy użytkownik należy do konkretnej roli.

## Jak to działa?

- ❶ Przed wywołaniem metody `isUserRole()` należy przeprowadzić **uwierzytelnianie** użytkownika. Jeśli metoda zostanie wywołana dla użytkownika, który NIE został uwierzytelniony, to zawsze zwróci ona wartość `false`.
- ❷ Kontener otrzymuje argument wywołania metody `isUserRole()`, w tym przypadku jest nim "Menedzer", i porównuje go z rolami zdefiniowanymi w aplikacji.
- ❸ Jeśli istnieje odwzorowanie kojarzące danego użytkownika z odpowiednią rolą, metoda zwróci wartość `true`.

\* Znamy jednak pewną bardzo fajną książkę poświęconą zagadnieniom technologii EJB...



***W jaki sposób dopasować role używane w deskryptorze wdrożenia z rolami używanymi w serwlecie?***

# Deklaratywna strona bezpieczeństwa programowego

Istnieje duże prawdopodobieństwo, że jeśli programista podał w kodzie serwletu nazwę roli (w formie argumentu wywołania metody `isUserInRole()`), to była to *nazwa wymyślona*. Programista albo nie *znał* prawdziwych nazw, albo pisał właśnie komponent, który będzie wielokrotnie używany przez wiele firm, które jednak najprawdopodobniej nie będą używać takich samych nazw ról jak te wykorzystane w jego kodzie. (Oczywiście jeśli programista naprawdę chce napisać komponent nadający się do *wielokrotnego wykorzystania*, to podawanie nazw ról na stałe w kodzie jest Koszmarnym Pomysłem; jak na razie jednak powstrzymamy się od komentarzy).

Okazuje się, że deskryptor wdrożenia dysponuje mechanizmem pozwalającym na odwzorowywanie zakodowanych (co oznacza *wymyślonych*) nazw

ról stosowanych w *serwlecie* na „oficjalne” nazwy zadeklarowane w *kontenerze* przy użyciu elementów `<security-role>`. Wyobraźmy sobie, na przykład, że programista używał wartości "Menedzer" w wywołaniach metody `isUserInRole()`, lecz w Twojej firmie w elemencie `<security-role>` stosowana jest wartość "Administrator", a co więcej — rola "Menedzer" w ogóle nie istnieje. A zatem, nawet jeśli nie możesz powstrzymać programisty przed umieszczaniem nazw w kodzie, to masz narzędzie, które możesz zastosować w momencie, gdy nazwy używane w kodzie nie odpowiadają *prawdziwym* nazwom ról używanym w aplikacji. Poza tym, nawet gdybyś dysponował kodem źródłowym serwletu, to czy chciałoby Ci się modyfikować go, rekompilować i ponownie uruchamiać aplikację wyłącznie po to, by zamienić każde wystąpienie łańcucha "Menedzer" na "Administrator"?

## W kodzie serwletu

```
if (request.isUserInRole("Menedzer")) {
 // wykonujemy stronę AktualizujPrzepis
 ...
} else {
 // wykonujemy stronę PokazPrzepis
 ...
}
```

W tym przypadku, gdyby element `<security-role-ref>` nie istniał, to powyższy kod nie działałby poprawnie, gdyż nie istniałaby rola o nazwie „Menedzer”.

## W deskrytorze wdrożenia

```
<web-app ...>
 <servlet>
 <security-role-ref>
 <role-name>Menedzer</role-name>
 <role-link>Administrator</role-link>
 </security-role-ref>
 ...
 </servlet>
 ...
 <security-role>
 <role-name>Administrator</role-name>
 </security-role>
 ...
</web-app>
```

Element `<security-role-ref>` odwzorowuje programowe (trwale zakodowane w serwlecie) nazwy ról na odpowiednie elementy deklaratywne `<security-role>`.



Oglądaj to!

**Kontener wykorzysta skojarzenie zdefiniowane przy użyciu elementu `<security-role-ref>`, nawet jeśli nazwy programowe odpowiadają „prawdziwym” nazwom podanym w elemencie `<security-role>`.**

Kiedy kontener pobiera argument metody `isUserInRole()`, to w **PIERWSZEJ** kolejności sprawdza, czy jego wartość odpowiada któremuś z podanych elementów `<security-role-ref>`. Jeśli taki element zostanie odnaleziony, to kontener wykorzysta go nawet w sytuacji, gdy nazwa roli podana w kodzie odpowiada jednej z nazw zdefiniowanych w elemencie `<security-role>`. Zastanów się nad tych przez chwilę — mogłoby się zdarzyć, że faktycznie będziesz używać roli o nazwie "Menedzer", lecz mogłaby ona mieć całkowicie odmienne znaczenie niż to, o którym myślał programista. A zatem mógłbyś odwzorować tę podaną w kodzie nazwę "Menedzer" na "Administrator", a następnie odwzorować nazwę "Dyrektor" na "Menedzer". Więc, jeśli ta sama nazwa roli zostanie podana w elemencie `<security-role-ref>` oraz `<security-role>`, to zawsze zostanie zastosowana nazwa z pierwszego z tych elementów.

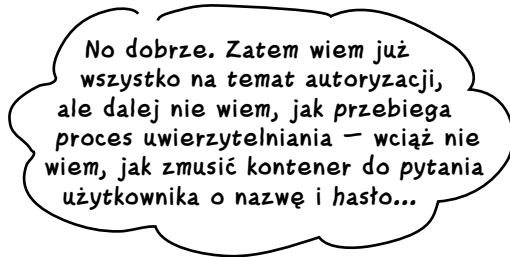
Ćwiczenie z zakresu bezpieczeństwa



Przyjmij, że wszystkie ograniczenia przedstawione poniżej wykorzystują te same elementy <url-pattern> oraz <http-method>. Opierając się na przedstawionych poniżej kombinacjach elementów, podaj, które role dysponują prawem do bezpośredniego dostępu do wskazanych zasobów.

	Nikt	Gosc	Członek	Administrator	Każdy
1 <security-constraint> ... <auth-constraint> <role-name>Gosc</role-name> </auth-constraint> </security-constraint>					
2 <security-constraint> ... <auth-constraint /> </security-constraint>					
3 <security-constraint> ... <auth-constraint> <role-name>Administrator</role-name> </auth-constraint> </security-constraint> <security-constraint> ... <auth-constraint> <role-name>Gosc</role-name> </auth-constraint> </security-constraint>					
4 <security-constraint> ... <auth-constraint> <role-name>Gosc</role-name> </auth-constraint> </security-constraint> <security-constraint> ... <auth-constraint> <role-name>*</role-name> </auth-constraint> </security-constraint>					
5 <security-constraint> ... <auth-constraint> <role-name>Członek</role-name> </auth-constraint> </security-constraint>  <security-constraint> ... </security-constraint>					
6 <security-constraint> ... <auth-constraint> <role-name>Członek</role-name> </auth-constraint> </security-constraint> <security-constraint> ... <auth-constraint/> </security-constraint>					

Przyjmij, że nie zdefiniowano  
ŻADNEGO elementu  
<auth-constraint>.



No dobrze. Zatem wiem już wszystko na temat autoryzacji, ale dalej nie wiem, jak przebiega proces uwierzytelniania – wciąż nie wiem, jak zmusić kontener do pytania użytkownika o nazwę i hasło...

## Uwierzytelnianie raz jeszcze

Z punktu widzenia kontenera J2EE uwierzytelnianie sprowadza się do konieczności wykonania następujących operacji: pobrania *nazwy* użytkownika i *hasła* oraz sprawdzenia ich *poprawności*.

Kontener automatycznie rozpocznie proces uwierzytelniania za pierwszym razem, gdy nieuwierzytelniony użytkownik zażąda dostępu do chronionego zasobu. Istnieją cztery rodzaje uwierzytelniania, które udostępnia kontener, a *podstawowa* różnica pomiędzy nimi polega na „poziomie bezpieczeństwa przesyłanej nazwy użytkownika i jego hasła”.

### CZTERY typy uwierzytelniania

**BASIC** — uwierzytelnianie proste; w tym przypadku informacje o użytkowniku są przesyłane w postaci zakodowanej (*ale nie zaszyfrowanej*). Choć rozwiązanie to może stwarzać pozory bezpieczeństwa, trzeba pamiętać, iż w rzeczywistości jego poziom jest bardzo niski, gdyż używany sposób kodowania (**base64**) jest powszechnie znany.

**DIGEST** — uwierzytelnianie przy użyciu skrótu; w tym przypadku informacje podane podczas logowania są przesyłane w bardziej bezpieczny sposób, jednak ze względu na to, że ten mechanizm szyfrowania nie jest powszechnie stosowany, implementacja tej metody uwierzytelniania w kontenerach J2EE jest opcjonalna. Więcej informacji na jej temat można znaleźć w dokumencie IETF *RFC 2617* ([www.ietf.org/rfc/rfc2617.txt](http://www.ietf.org/rfc/rfc2617.txt)).

**CLIENT-CERT** — uwierzytelnianie przy użyciu certyfikatu klienta; ta metoda uwierzytelniania pozwala na przekazywanie informacji podanych podczas logowania w bardzo bezpieczny sposób, przy użyciu certyfikatu klucza publicznego (*PKC*, ang. *Public Key Certificate*). Wadą tego mechanizmu jest wymóg posiadania odpowiedniego certyfikatu przed podjęciem próby zalogowania się do systemu. Ponieważ klienci stosunkowo rzadko posiadają niezbędne certyfikaty, zatem ta metoda uwierzytelniania przeważnie jest używana w rozwiązaniach typu „biznes dla biznesu”.

Przedstawione powyżej trzy typy uwierzytelniania — BASIC, DIGEST oraz CLIENT-CERT — używają standardowego okna dialogowego przeglądarki do podawania nazwy i hasła użytkownika. Zasada działania czwartego typu uwierzytelniania — FORM — jest inna.

**FORM** — ten typ uwierzytelniania pozwala na stworzenie własnego formularza, którego postać określana jest przy użyciu standardowych elementów HTML. Jednak spośród wszystkich czterech typów uwierzytelniania ten wykorzystuje najmniej bezpieczny sposób przesyłania informacji podanych przez użytkownika. W tym przypadku informacje te są przesyłane *bez* jakiegokolwiek szyfrowania.

## Implementacja uwierzytelniania

To zadanie jest bardzo proste — sprowadza się ono do zadeklarowania schematu uwierzytelniania w deskrypcji wdrożenia. Podstawowym elementem używanym w tym celu jest `<login-config>`.

### Cztery przykłady postaci elementu `<login-config>`

```
<web-app...>
```

```
...
```

```
<login-config>
```

```
<auth-method>BASIC</auth-method> ←
```

```
</login-config>
```

```
</web-app>
```

Uwierzytelnianie proste jest proste. Po zadeklarowaniu tego elementu w deskrypcji wdrożenia kontener zajmie się całą resztą i będzie automatycznie prosił o podanie nazwy użytkownika i hasła w momencie przestania żądania dotyczącego zasobu, do którego dostęp jest ograniczony.

— lub —

```
<web-app...>
```

```
...
```

```
<login-config>
```

```
<auth-method>DIGEST</auth-method> ←
```

```
</login-config>
```

```
</web-app>
```

Jeśli używany kontener obsługuje ten typ uwierzytelniania, to będzie w stanie zająć się WSZYSTKIMI szczegółami.

— lub —

```
<web-app...>
```

```
...
```

```
<login-config>
```

```
<auth-method>CLIENT-CERT</auth-method> ←
```

```
</login-config>
```

```
</web-app>
```

Uwierzytelnianie typu CLIENT-CERT jest łatwe w konfiguracji, lecz klient musi posiadać stosowne certyfikaty. Ten typ uwierzytelniania zapewnia WYJĄTKOWĄ ochronę i bezpieczeństwo.

— lub —

```
<web-app...>
```

```
...
```

```
<login-config>
```

```
<auth-method>FORM</auth-method> ←
```

```
<form-login-config>
```

```
<form-login-page>/stronaLogowania.html</form-login-page>
```

```
<form-error-page>/stronaBledu.html</form-error-page>
```

```
</form-login-config>
```

```
</login-config>
```

```
</web-app>
```

Uwierzytelnianie typu FORM jest najtrudniejsze do skonfigurowania; przyjrzymy się mu szczegółowo na następnej stronie.

Za wyjątkiem uwierzytelniania typu FORM zadeklarowanie elementu `<login-config>` w deskrypcji wdrożenia jest równoznaczne z zakończeniem implementacji uwierzytelniania! (Oczywiście zakładając, że już wcześniej skonfigurowałeś na serwerze wszystkie informacje dotyczące użytkowników, ich haseł oraz skojarzonych z nimi ról).

# Uwierzalnianie typu FORM

Choć w porównaniu z *innymi* typami uwierzalniania implementacja uwierzalniania typu FORM jest znacznie bardziej skomplikowana, niemniej jednak nie jest ona aż tak straszna. W pierwszej kolejności należy stworzyć własny formularz HTML służący do podawania nazwy użytkownika i hasła (choć równie dobrze może on być generowany przez stronę JSP). Następnie trzeba stworzyć stronę z informacją o błędzie, którą kontener będzie wyświetlać w momencie, gdy informacje podane przez użytkownika w formularzu logowania nie będą poprawne. Ostatnim etapem jest podanie obu tych stron w deskrytorze wdrożenia, w elemencie `<login-config>`. Uwaga: W przypadku stosowania uwierzalniania typu FORM pamiętaj o wykorzystaniu protokołu SSL lub włączeniu opcji śledzenia sesji, gdyż w przeciwnym razie kontener może nie rozpoznać informacji ponownie przesłanych z formularza logowania!

## Czynności wykonywane przez CIEBIE:

- ① Zadeklarowanie elementu `<login-config>` w deskrytorze wdrożenia.
- ② Stworzenie formularza HTML służącego do logowania.
- ③ Stworzenie strony HTML z informacją o błędzie.

### 1 W deskrytorze wdrożenia...

```
<login-conf>
 <auth-method>FORM</auth-method>
 <form-login-config>
 <form-login-page>/stronaLogowania.html</form-login-page>
 <form-error-page>/stronaBledu.html</form-error-page>
 </form-login-config>
</login-conf>
```

**Kluczem do komunikacji z kontenerem są trzy nazwy umieszczone w formularzu HTML służącym do logowania. Oto one:**

- `j_security_check`;
- `j_username`;
- `j_password`.

### 2 W pliku stronaLogowania.html...

```
Zaloguj się bratku:

<form method="POST" action="j_security_check">
 <input type="text" name="j_username">
 <input type="password" name="j_password">
 <input type="submit" value="Zaloguj">
</form>
```

Aby kontener mógł poprawnie przeprowadzić uwierzalnianie, atrybut `action` formularza logowania MUSI mieć wartość `j_security_check`.

Kontener wymaga, by nazwa użytkownika w żądaniu była zapisana w parametrze nazwanym `j_username`.

Kontener wymaga, by hasło użytkownika w żądaniu było zapisane w parametrze nazwanym `j_password`.

### 3 W pliku stronaBledu.html...

```
<html><body>
 Przykro mi stary... to nie to hasło!
</body></html>
```

**Nie dekoncentruj się!**

Jeśli zamierzasz przystąpić do egzaminu, musisz zapamiętać wszystkie informacje podane na tej stronie!

## Podsumowanie informacji o typach uwierzytelniania

Poniższa tabela przedstawia podstawowe cechy wszystkich czterech typów uwierzytelniania. Kolumna „Specyfikacja” określa, czy dany typ mechanizmu uwierzytelniania został zdefiniowanych w specyfikacji protokołu HTTP czy też platformy J2EE. (Podpowiedź: Przystępując do egzaminu, musisz pamiętać informacje zamieszczone w tej tabeli).

Typ uwierzytelniania	Specyfikacja	Integralność danych	Uwagi
BASIC	HTTP	Base64 — niska	Standardowa metoda protokołu HTTP obsługiwana przez wszystkie przeglądarki.
DIGEST	HTTP	Wysoka, lecz niższa niż w przypadku stosowania protokołu SSL	Opcjonalna zarówno dla protokołu HTTP, jak i kontenerów J2EE.
FORM	J2EE	Bardzo niska — brak jakiegokolwiek szyfrowania	Pozwala na stosowanie własnych formularzy do logowania.
CLIENT-CERT	J2EE	Wysoka — klucz publiczny (PKC)	Wysoka, lecz użytkownicy muszą posiadać certyfikaty.

### Nie ma niemądrych pytań

**❓ A co wspólnego z uwierzytelnianiem ma integralność danych?**

**U.** Podczas uwierzytelniania użytkownik przesyła do serwera swoją nazwę i hasło. **Integralność danych** oraz **poufność** określają stopień zabezpieczenia tych informacji przed możliwością ich podejrzenia lub modyfikacji. Interesują nas właśnie zagadnienia związane z zapewnianiem integralności i poufności danych podczas logowania.

*Integralność* danych oznacza, że dane, które docierają do serwera to te same dane, które zostały podane przez użytkownika. Innymi słowy, oznacza ona, że nikt nie zmodyfikował tych danych podczas ich przesyłania. Z kolei *poufność* danych oznacza, że nikt nie mógł nawet podejrzeć przesyłanych informacji. W większości przypadków traktujemy poufność oraz integralność danych jako jeden wspólny cel — metodę *zabezpieczania informacji podczas ich przesyłania*.



### Zaostrz ołówek

Uzupełnij brakujące fragmenty przedstawionej niżej aplikacji wykorzystującej uwierzytelnianie bazujące na formularzu stworzonym przez programistę. Ćwiczenie to ma pomóc Ci w zapamiętaniu informacji związanych z uwierzytelnianiem podawanych w deskryptorze wdrożenia oraz formularzu HTML. (Rozwiązanie można znaleźć na poprzedniej stronie).

#### Deskryptor wdrożenia

```
<login-conf>
 <auth-method> </auth-method>
 <form-login-config>
 < >/stronaLogowania.html</ >
 <form-error-page>/stronaBledu.html</form-error-page>
 </form-login-config>
</login-conf>
```

#### HTML

Zaloguj się bratku: <br>
<form method="POST" action="  ">
 <input type="text" name="  ">
 <input type="password" name="j\_password">
 <input type="submit" value="Zaloguj">
</form>

Uwierzytelnianie wykorzystujące niestandardowy formularz nie zapewnia żadnego szyfrowania danych. Ale ja nie chcę używać tych ohydnych okienek dialogowych wyświetlanych przez przeglądarki, które są używane we wszystkich trzech pozostałych typach uwierzytelniania. Och... gdyby tylko był jakiś sposób pozwalający na zastosowanie własnego formularza i ochranianie przesyłanej nazwy użytkownika i jego hasła...



## Ona nie wie o dostępnych w J2EE „chronionych połączeniach warstwy transportowej”

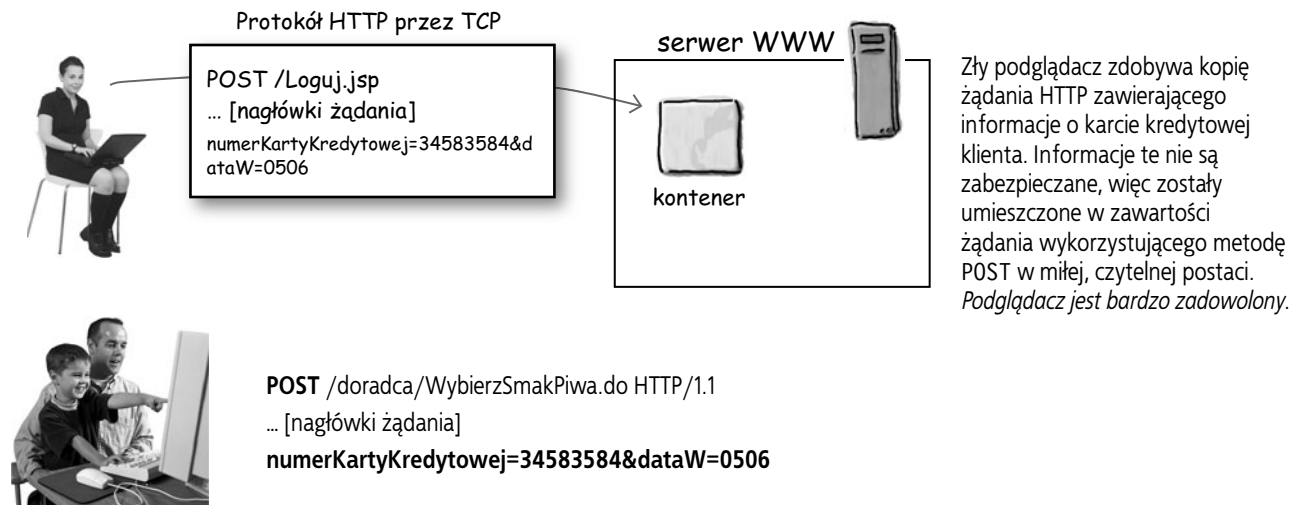
**Nie wpadaj w panikę!** Możesz używać własnego formularza i jednocześnie przysyłać dane w bezpieczny sposób. Informacje podawane podczas logowania są przecież zwyczajnymi **danymi**, zatem możesz je zabezpieczać w taki sam sposób, w jaki zwykle zabezpieczasz informacje o numerach kart kredytowych wpisywane przez użytkowników w Twoim sklepie internetowym — a więc za pomocą oferowanych przez kontener mechanizmów zapewniających integralność i poufność danych.

# Zabezpieczanie przesyłanych informacji

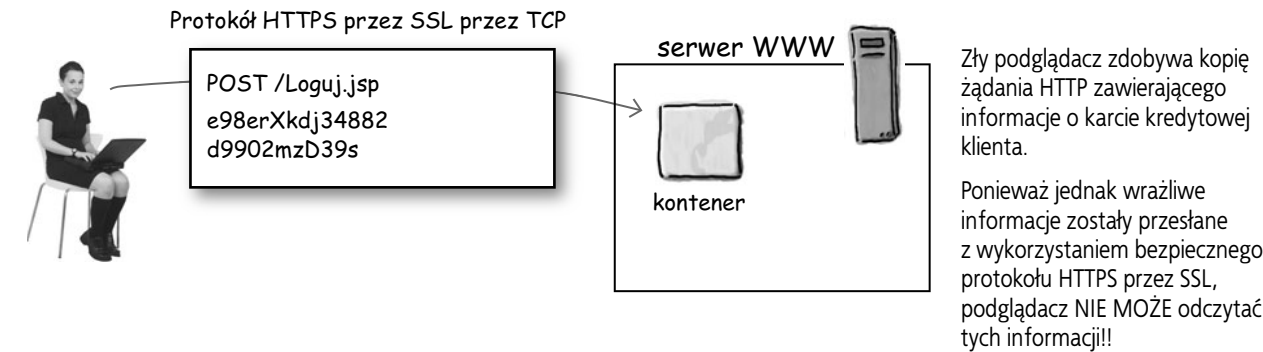
## — rozwiązaniem jest protokół HTTPS

Jeśli poinformujesz kontener J2EE, że chcesz, by dane były przesyłane z zachowaniem ich integralności i (lub) poufności, to specyfikacja J2EE gwarantuje, że będą one transmitowane przy wykorzystaniu „**zabezpieczonego połączenia warstwy transportowej**”. Innymi słowy, kontenery nie są *zobowiązane* do korzystania z żadnego konkretnego protokołu zapewniającego bezpieczne połączenia, lecz w praktyce niemal zawsze korzystają z bezpiecznego protokołu HTTPS przez SSL.

### Żądanie HTTP — niezabezpieczone w żaden sposób



### Zabezpieczone żądanie przesyłane przy użyciu protokołu HTTPS przez SSL





NIE mów mi, że jeśli  
zdecyduję się na stosowanie  
bezpiecznego przesyłania danych,  
to **WSZYSTKIE** żądania i odpowiedzi  
w mojej aplikacji będą  
szyfrowane...

### Zaostrz ołówek



Przeanalizuj informacje zamieszczone w tym rozdziale. Jeśli Twoja aplikacja internetowa ma być szybka, wydajna i bezpieczna, to musisz odpowiedzieć sobie na kilka pytań... (na te pytania nie znajdziesz odpowiedzi w treści książki, więc nie trać czasu na ich szukanie...).

Czy każde żądanie i odpowiedź muszą być bezpieczne? Jeśli nie, to które fragmenty aplikacji wymagają stosowania bezpiecznej transmisji danych?

Co, według Ciebie, oznacza poufność danych?

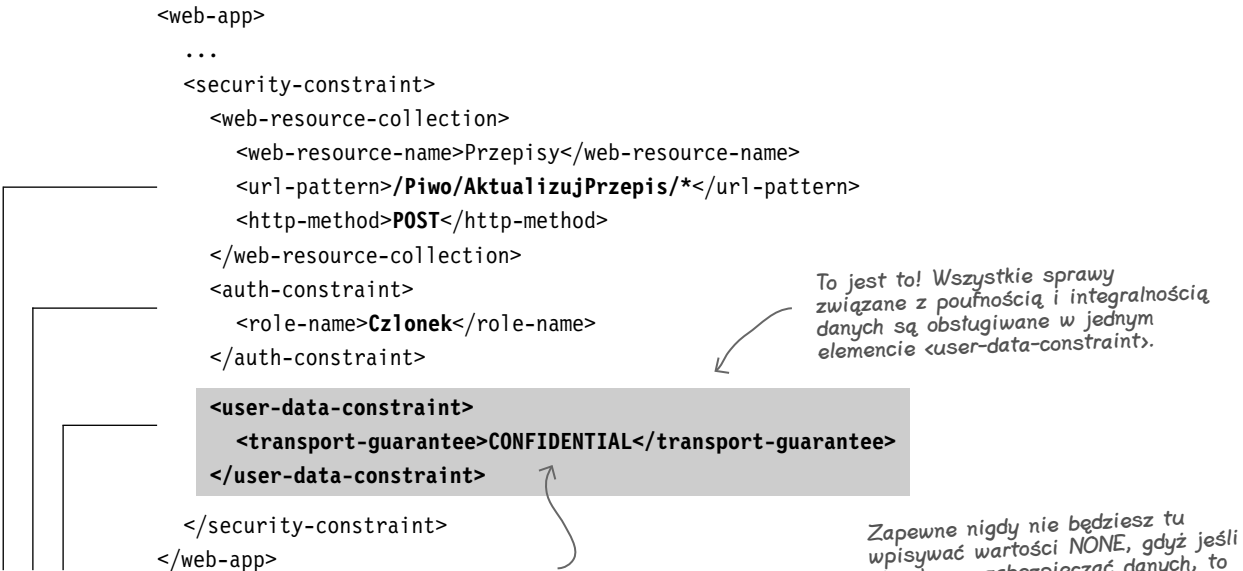
Co, według Ciebie, oznacza integralność danych?

Gdybyś mógł stosować bezpieczną transmisję danych wyłącznie w przypadku niektórych żądań i odpowiedzi, to w jaki sposób chciałbyś poinformować kontener, o *które* żądania i odpowiedzi chodzi?

Czy potrafisz wskazać inne elementy deskryptora wdrożenia operujące na równie wysokim poziomie szczegółowości co elementy, których należałoby użyć do deklarowania bezpiecznej transmisji danych?

# Jak efektywnie i deklaratywnie zaimplementować poufność i integralność danych?

A zatem znowu powracamy do deskryptora wdrożenia. Okazuje się, że i tym razem w celu implementacji zarówno poufności, jak i integralności danych będziemy stosowali nasz stary, sprawdzony element `<security-constraint>`, uzupełniając go o nowy podelement — `<user-data-constraint>`. Jeśli się nad tym zastanowisz, to okaże się, że rozwiązanie to jest sensowne — jeśli bowiem planujesz uwierzytelnianie dostępu do zasobu, to prawdopodobnie będziesz także chciał, by wykorzystywane przy tym informacje były transmitowane w bezpieczny sposób.



Połącz te trzy elementy w jedną logiczną całość mającą następujące znaczenie:

Tylko użytkownicy należący do roli **Członek** mogą przesyłać żądania używające metody POST i odnoszące się do zasobów umieszczonych w katalogu **AktualizujPrzepis**, przy czym informacje mają być przesyłane w bezpieczny sposób.

## Dopuszczalne wartości elementu <transport-guarantee>

### NONE

To jest wartość domyślna. Oznacza ona, że żadne zabezpieczanie danych nie jest stosowane.

### INTEGRAL

Dane nie mogą być zmodyfikowane podczas przekazywania.

### CONFIDENTIAL

Nikt nie może podejrzeć danych podczas ich przesyłania.

Uwaga: Chociaż specyfikacja tego nie określa, to jednak niemal wszystkie kontenery J2EE wykorzystują protokół SSL do zapewnienia bezpiecznej transmisji danych; to z kolei oznacza, że użycie obu wartości — INTEGRAL oraz CONFIDENTIAL — przynosi takie same efekty — zapewnia zarówno poufność, jak i integralność danych. Ponieważ w elemencie `<security-constraint>` można umieścić tylko jeden element `<user-data-constraint>`, niektórzy zalecają stosowanie wartości CONFIDENTIAL. W praktyce jednak nie ma to i nie będzie miało znaczenia, chyba że zaczniemy stosować nowy (i nadzwyczajny) kontener, który nie używa protokołu SSL.

Chwileczkę... jak można zagwarantować poufność danych przesyłanych w żądaniu? Przecież kontener nawet nie wie, że ma zabezpieczyć transmisję do momentu, gdy klient prześle żądanie...



## Ochrona danych żądania

Pamiętaj, że informacje umieszczone w elemencie `<security-constraint>` w deskrytorze wdrożenia odnoszą się do tego, co się stanie *po* przesłaniu żądania. Innymi słowy, kontener zaczyna przeglądać zawartość tego elementu w celu określenia sposobu odpowiedzi w chwili, kiedy klient już przesłał żądanie. *Dane żądania zostały już zatem przesłane*. W jaki sposób można przekazać przeglądarce wiadomość: „A tak swoją drogą, jeśli użytkownik zażąda dostępu do *tego* zasobu, to *przed* wysłaniem żądania nawiąż połączenie przy użyciu SSL”.

Co zatem możemy zrobić?

Już wiesz, w jaki sposób można zmusić klienta do wyświetlenia okna dialogowego lub strony z formularzem do logowania — poprzez zdefiniowanie w deskrytorze wdrożenia zasobu, do którego dostęp jest ograniczony. W takim przypadku, kiedy użytkownik o niepotwierdzonej tożsamości zgłosi żądanie, kontener automatycznie zapoczątkuje proces uwierzytelniania.

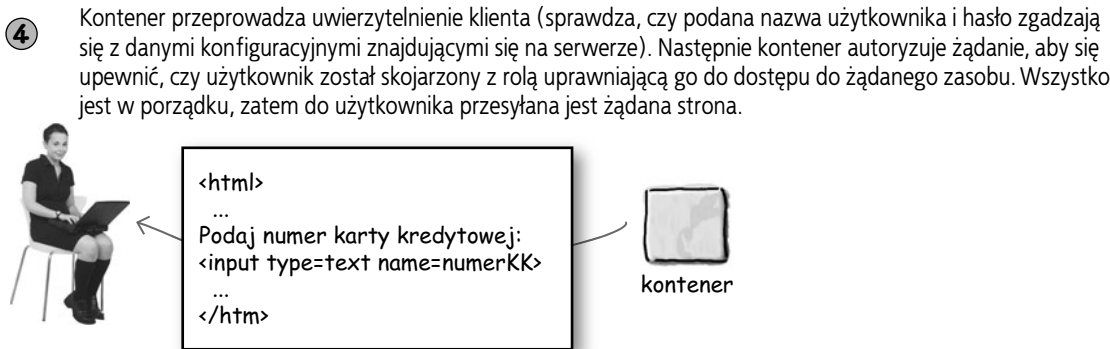
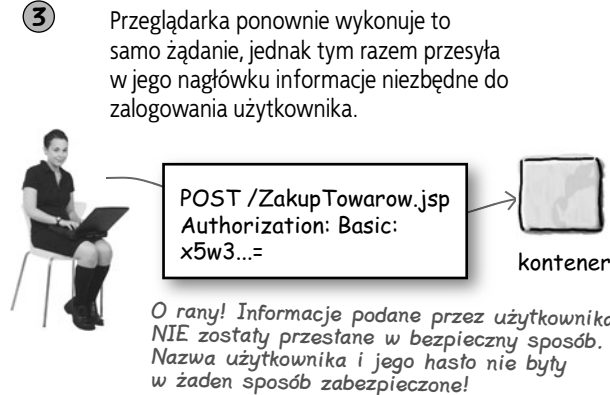
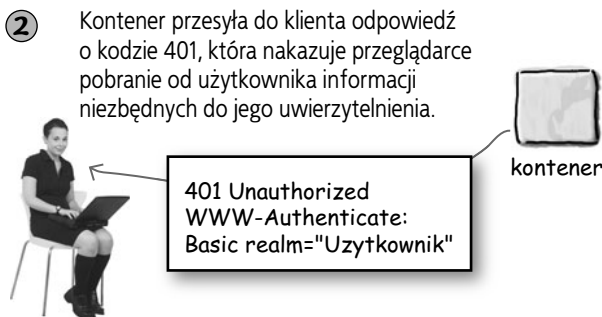
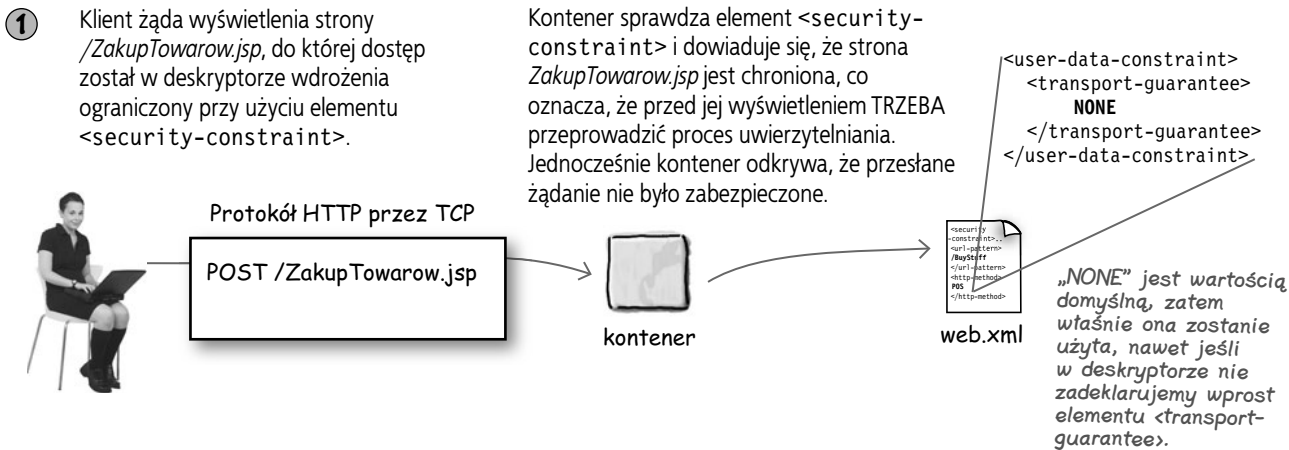
A zatem teraz musimy określić, w jaki sposób zabezpieczyć dane przesyłane w żądaniu nawet (a czasami *zwłaszcza*) w sytuacji, gdy użytkownik nie został wcześniej zalogowany.

Przecież może nam chodzić o zabezpieczenie informacji podawanych podczas logowania!

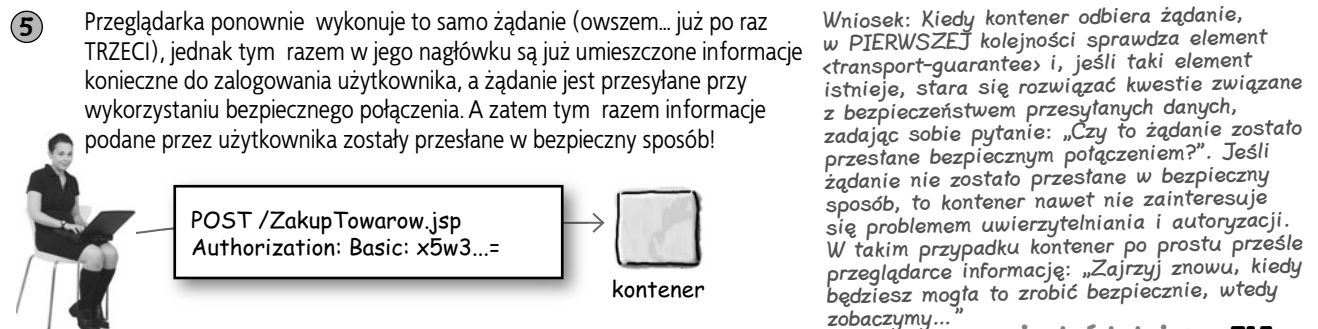
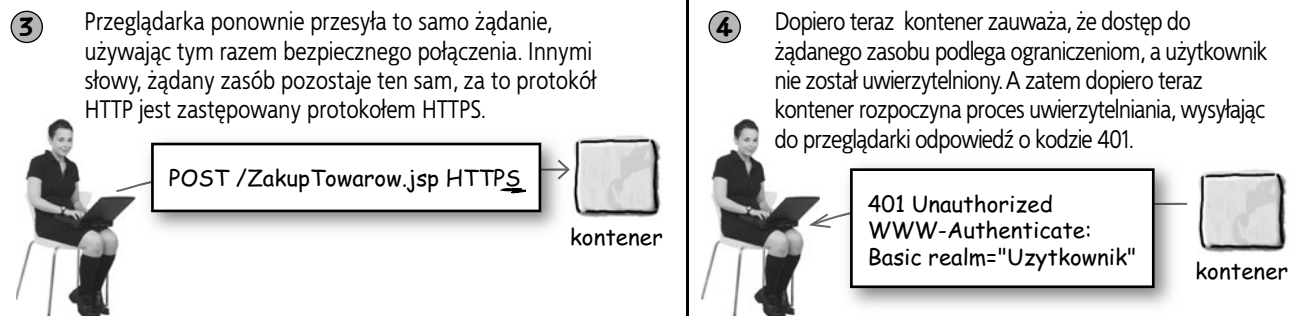
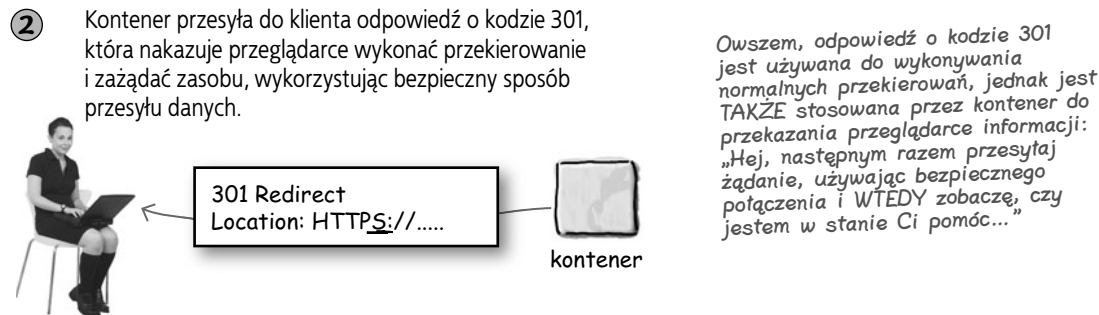
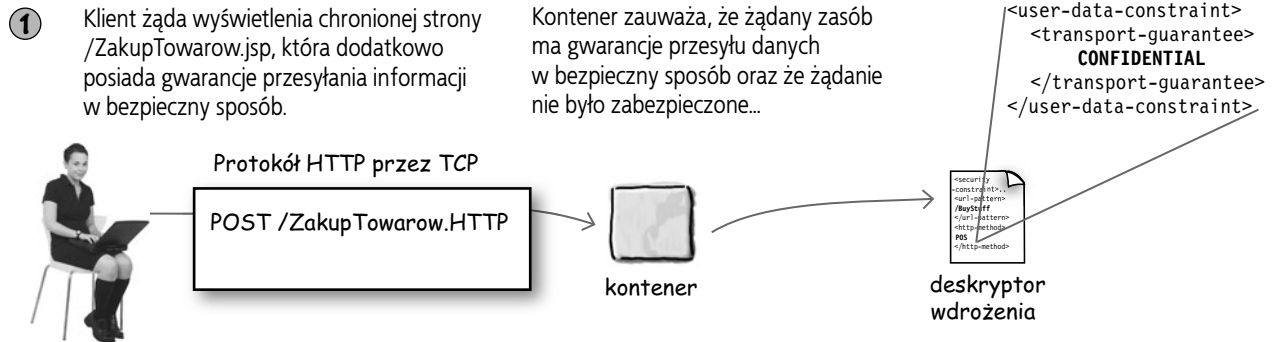
Przewróć kartkę, aby zobaczyć rozwiązanie tego problemu...

Bez zastosowania elementu <transport-guarantee>

Nieuwierzytelniony klient żąda dostępu do zasobu o ograniczonym dostępie, NIE dysponując przy tym gwarancją bezpiecznej transmisji danych



## Nieuwierzytelniony klient żąda dostępu do zasobu o ograniczonym dostępie, dysponując przy tym gwarancją BEZPIECZNEJ transmisji danych





Oglądaj to!

**Aby się upewnić, że informacje niezbędne do zalogowania użytkownika zostaną przesłane w sposób bezpieczny, należy zdefiniować gwarancje bezpiecznej transmisji dla WSZYSTKICH ochronianych zasobów, które mogą spowodować rozpoczęcie procesu uwierzytelniania!**

*Pamiętaj, że jeśli deklaratywnie korzystasz z mechanizmów uwierzytelniania, to klient nigdy nie wykonuje jawnego żądania skierowanego do strony logowania. Klient zapoczątkowuje proces uwierzytelniania i autoryzacji, żądając dostępu do jednego z chronionych zasobów. A zatem, aby się upewnić, że informacje niezbędne do zalogowania użytkownika zostaną przesłane przy wykorzystaniu bezpiecznego połączenia, należy dodać element <transport-guarantee> do definicji KAŻDEGO chronionego zasobu, do którego odwołanie może zapoczątkować proces uwierzytelniania!*

*W ten sposób kontener odbierze żądanie dotyczące chronionego zasobu, jednak ZANIM każe przeglądarce pobrać informacje niezbędne do zalogowania użytkownika, powie jej: „W ogóle nie WOLNO ci przysłać tego żądania, dopóki na nawiążesz bezpiecznego połączenia”. Później, kiedy przeglądarka ponownie prześle żądanie, usłyszysz w odpowiedzi: „A... świetnie... tym razem używasz bezpiecznego połączenia, jednak wciąż potrzebuję informacji niezbędnych do uwierzytelnienia użytkownika”. A zatem przeglądarka wyświetli stosowny formularz lub okienko dialogowe, pobierze od użytkownika niezbędne informacje i prześle je w TRZECIM żądaniu.*

### Nie ma niemądrych pytań

**P. Nie rozumiem, dlaczego kontener przesyła klientowi odpowiedź 301 (przekierowanie) w przypadku, gdy okaże się, że żądanie nie zostało przesłane przy wykorzystaniu bezpiecznego połączenia. Czy takie przekierowanie nie spowoduje po prostu wysłania ponownego żądania do tego samego zasobu?**

**U.** Zazwyczaj można sobie wyobrazić przekierowanie jako stwierdzenie typu: „Hej, przeglądarko, należy wyświetlić stronę o *innym* adresie”. Pamiętaj jednak, że przekierowanie jest niewidoczne dla użytkownika, gdyż jego przeglądarka automatycznie zażąda nowego zasobu o adresie przekazanym w nagłówku odpowiedzi 301.

Jednak w przypadku bezpiecznego przesyłania danych sprawy wyglądają nieco inaczej. Zamiast polecenia „Przejdź do *innego* zasobu” kontener przekazuje przeglądarce informację: „Zażądaj jeszcze raz tego samego zasobu, lecz tym razem użyj innego protokołu — HTTPS, a nie HTTP”.

**P. A zatem, czy obsługa protokołu HTTPS przez SSL jest w jakiś sposób wbudowana w kontener?**

**U.** Specyfikacja tego nie gwarantuje, jednak jest bardzo prawdopodobne, że kontener będzie używać właśnie tego protokołu (tak zwanych bezpiecznych gniazd). Jednak wcale nie jest powiedziane, że **możliwość jego wykorzystania będzie dostępna automatycznie!** Prawdopodobnie będziesz musiał samodzielnie skonfigurować SSL w swoim kontenerze, a co ważniejsze — będziesz także musiał uzyskać odpowiedni certyfikat!

Chociaż sam musisz to sprawdzić w dokumentacji kontenera, z dużym prawdopodobieństwem można przyjąć, że kontener będzie w stanie samodzielnie wygenerować certyfikat, który będziesz mógł wykorzystać do testów. Jednak do celów produkcyjnych będziesz musiał uzyskać certyfikat klucza publicznego z jakiegoś „oficjalnego” źródła, takiego jak firma VeriSign.

(Powinieneś wiedzieć, że certyfikaty oraz protokoły takie jak HTTPS i SSL znacznie wykraczają poza zakres egzaminu. Podchodząc do egzaminu, musisz jedynie wiedzieć, jakie informacje należy umieszczać w deskryptorze wdrożenia i dlaczego. Nie musisz być od razu administratorem ani specjalistą od zabezpieczeń sieciowych).



## Zaostrz ołówek

Skonfiguruj zabezpieczenia aplikacji internetowej, uzupełniając trzy bloki deskryptora wdrożenia. Aplikacja ma działać w następujący sposób:

Chcemy, aby wszyscy mogli wykonywać zapytania używające metody GET i odwołujące się do zasobów umieszczonych w katalogu *Piwo/AktualizujPrzepis* (w tym także we wszystkich jego podkatalogach), jednak zapytania używające metody POST odwołujące się do tych zasobów mogą wykonywać jedynie osoby skojarzone z rolą "Administrator". Co więcej, chcemy, by dane były zabezpieczone tak, aby nikt nie był w stanie ich podejrzeć.

```
<web-app ...>
```

```
<security-constraint>
```

```
</security-constraint>
```

```
...
```

```
</web-app>
```



Wypełnij poniższą tabelę, zapisując w niej odpowiednie elementy deskryptora wdrożenia. Odpowiedzi możesz znaleźć na następnej kartce. (Nawet nie próbuj tam ZERKAĆ!).

Zamierzony cel zabezpieczenia	Kod umieszczany w deskrypcji wdrożenia
Chcesz, by kontener automatycznie przeprowadzał uwierzytelnianie typu BASIC.	
Chcesz używać swojej własnej strony z formularzem logowania nazwanej <i>stronaLogowania.html</i> (znajdującej się bezpośrednio w katalogu głównym aplikacji). Jeśli użytkownik nie może zostać uwierzytelniony, to ma być wyświetlana strona <i>bladLogowania.html</i> .	
Chcesz ograniczyć dostęp do wszystkich zasobów z rozszerzeniem <i>.do</i> , tak aby wszyscy mogli odwoływać się do nich przy wykorzystaniu metody GET, lecz tylko użytkownicy skojarzeni z rolą "Członek" mogli używać metody POST.  (NIE musisz podawać elementów deskryptora niezbędnych do skonfigurowania sposobu logowania).	
Chcesz ograniczyć dostęp do wszystkich zasobów przechowywanych w katalogu <i>foo/bar</i> , tak aby niezależnie od używanej metody HTTP mogli się do nich odwoływać jedynie użytkownicy skojarzeni z rolą "Administrator".  (NIE musisz podawać elementów deskryptora niezbędnych do skonfigurowania sposobu logowania).	



## ODPOWIEDZI

Chcemy, aby wszyscy mogli wykonywać żądania używające metody GET i odwołujące się do zasobów umieszczonych w katalogu Piwo/AktualizujPrzepis (w tym także we wszystkich jego podkatalogach), jednak żądania używające metody POST odwołujące się do tych zasobów mogą wykonywać jedynie osoby skojarzone z rolą "Administrator". Co więcej, chcemy, by dane były zabezpieczone tak, aby nikt nie był w stanie ich podejrzeć.

```
<web-app ...>
```

```
<security-constraint>
```

```
<web-resource-collection>
```

```
<web-resource-name>Przepisy</web-resource-name>
```

```
<url-pattern>/Piwo/AktualizujPrzepis/*</url-pattern>
```

```
<http-method>POST</http-method>
```

```
</web-resource-collection>
```

Pamiętaj, że wzorzec URL reprezentujący zabezpieczane katalogi musi się kończyć ciągiem znaków `"/"`.

```
<auth-constraint>
```

```
<role-name>Administrator</role-name>
```

```
</auth-constraint>
```

Gdybyś nie umieścił w deskrypcorze **ŻADNEGO** elementu `<auth-constraint>`, to **KĄDZY** mógłby wykonywać żądania wykorzystujące metodę **POST**. Dopiero umieszczenie tego znacznika powoduje, że tylko użytkownicy skojarzeni z rolą Administrator mogą używać kombinacji obejmującej podany wzorzec URL i wskazaną metodę HTTP.

```
<user-data-constraint>
```

```
<transport-guarantee>CONFIDENTIAL</transport-guarantee>
```

```
</user-data-constraint>
```

Równie dobrze mógłbyś w tym miejscu użyć wartości **INTEGRAL** i niemal wszystkie kontenery także zapewniłyby poufność przesyłanych informacji, gdyż znakomita większość kontenerów realizuje bezpieczne połączenia przy wykorzystaniu protokołu SSL (choć specyfikacja tego nie wymaga).

```
</security-constraint>
```

```
...
```

```
</web-app>
```



# Zaostrz ołówek

## ODPOWIEDZI

Zamierzony cel zabezpieczenia	Kod umieszczany w deskrytorze wdrożenia
Chcesz, by kontener automatycznie przeprowadzał uwierzytelnianie typu BASIC.	<pre>&lt;web-app ...&gt; ... &lt;login-config&gt;   &lt;auth-method&gt;BASIC&lt;/auth-method&gt; &lt;/login-config&gt; &lt;/web-app&gt;</pre>
Chcesz używać swojej własnej strony z formularzem logowania nazwanej <i>stronaLogowania.html</i> (znajdującej się bezpośrednio w katalogu głównym aplikacji). Jeśli użytkownik nie może zostać uwierzytelniony, to ma być wyświetlana strona <i>bladLogowania.html</i> .	<pre>&lt;web-app ...&gt; ... &lt;login-conf&gt;   &lt;auth-method&gt;FORM&lt;/auth-method&gt;   &lt;form-login-config&gt;     &lt;form-login-page&gt;stronaLogowania.html&lt;/form-login-page&gt;     &lt;form-error-page&gt;bladLogowania.html&lt;/form-error-page&gt;   &lt;/form-login-config&gt; &lt;/login-conf&gt; &lt;/web-app&gt;</pre>
Chcesz ograniczyć dostęp do wszystkich zasobów z rozszerzeniem „.do”, tak aby wszyscy mogli odwoływać się do nich przy wykorzystaniu metody GET, lecz tylko użytkownicy skojarzeni z rolą "Członek" mogli używać metody POST.  Konfigurujemy dwie rzeczy: zasoby o ograniczonym dostępie (na przykład wzorzec URL oraz metodę HTTP) oraz elementy <auth-constraint> definiujące role, które mają prawo dostępu do odwoływania się przy użyciu określonej metody HTTP (<http-method>) do określonych zasobów (<url-pattern>)	<pre>&lt;web-app ...&gt; ... &lt;security-constraint&gt;   &lt;web-resource-collection&gt;     &lt;web-resource-name&gt;SuperSprawa&lt;/web-resource-name&gt;     &lt;url-pattern&gt;*.do&lt;/url-pattern&gt;     &lt;http-method&gt;POST&lt;/http-method&gt;   &lt;/web-resource-collection&gt;   &lt;auth-constraint&gt;     &lt;role-name&gt;Członek&lt;/role-name&gt;   &lt;/auth-constraint&gt; &lt;/security-constraint&gt; &lt;/web-app&gt;</pre> <p>Zastosowaliśmy tutaj wzorzec adresu URL, który zawsze zaczyna się od znaku gwiazdki (*).</p>
Chcesz ograniczyć dostęp do wszystkich zasobów przechowywanych w katalogu foo/bar, tak aby niezależnie od używanej metody HTTP mogli się do nich odwoływać jedynie użytkownicy skojarzeni z rolą "Administrator".	<pre>&lt;web-app ...&gt; ... &lt;security-constraint&gt;   &lt;web-resource-collection&gt;     &lt;web-resource-name&gt;Obsługa&lt;/web-resource-name&gt;     &lt;url-pattern&gt;/foo/bar/*&lt;/url-pattern&gt;   &lt;/web-resource-collection&gt;   &lt;auth-constraint&gt;     &lt;role-name&gt;Administrator&lt;/role-name&gt;   &lt;/auth-constraint&gt; &lt;/security-constraint&gt; &lt;/web-app&gt;</pre> <p>Pominęliśmy element &lt;http-method&gt;, aby dostęp do WSZYSTKICH metod protokołu HTTP mieli tylko użytkownicy skojarzeni z rolą Administrator.</p>



Przyjmij, że wszystkie ograniczenia przedstawione poniżej wykorzystują te same elementy `<url-pattern>` oraz `<http-method>`. Opierając się na przedstawionych poniżej kombinacjach elementów, podaj, które role dysponują prawem do bezpośredniego dostępu do wskazanych zasobów.

	Nikt	Gosc	Czlonek	Administrator	Każdy
1 <code>&lt;security-constraint&gt;</code> ... <code>&lt;auth-constraint&gt;</code> <code>&lt;role-name&gt;Gosc&lt;/role-name&gt;</code> <code>&lt;/auth-constraint&gt;</code> <code>&lt;/security-constraint&gt;</code>		X			
2 <code>&lt;security-constraint&gt;</code> ... <code>&lt;auth-constraint /&gt;</code> <code>&lt;/security-constraint&gt;</code>	X				
3 <code>&lt;security-constraint&gt;</code> ... <code>&lt;auth-constraint&gt;</code> <code>&lt;role-name&gt;Administrator&lt;/role-name&gt;</code> <code>&lt;/auth-constraint&gt;</code> <code>&lt;/security-constraint&gt;</code> <code>&lt;security-constraint&gt;</code> ... <code>&lt;auth-constraint&gt;</code> <code>&lt;role-name&gt;Gosc&lt;/role-name&gt;</code> <code>&lt;/auth-constraint&gt;</code> <code>&lt;/security-constraint&gt;</code>		X		X	
4 <code>&lt;security-constraint&gt;</code> ... <code>&lt;auth-constraint&gt;</code> <code>&lt;role-name&gt;Gosc&lt;/role-name&gt;</code> <code>&lt;/auth-constraint&gt;</code> <code>&lt;/security-constraint&gt;</code> <code>&lt;security-constraint&gt;</code> ... <code>&lt;auth-constraint&gt;</code> <code>&lt;role-name&gt;*&lt;/role-name&gt;</code> <code>&lt;/auth-constraint&gt;</code> <code>&lt;/security-constraint&gt;</code>					X
5 <code>&lt;security-constraint&gt;</code> ... <code>&lt;auth-constraint&gt;</code> <code>&lt;role-name&gt;Czlonek&lt;/role-name&gt;</code> <code>&lt;/auth-constraint&gt;</code> <code>&lt;/security-constraint&gt;</code>  <code>&lt;security-constraint&gt;</code> ... <code>&lt;/security-constraint&gt;</code>  Przyjmij, że nie zdefiniowano ŻADNEGO elementu <code>&lt;auth-constraint&gt;</code> .					X
6 <code>&lt;security-constraint&gt;</code> ... <code>&lt;auth-constraint&gt;</code> <code>&lt;role-name&gt;Czlonek&lt;/role-name&gt;</code> <code>&lt;/auth-constraint&gt;</code> <code>&lt;/security-constraint&gt;</code> <code>&lt;security-constraint&gt;</code> ... <code>&lt;auth-constraint/&gt;</code> <code>&lt;/security-constraint&gt;</code>	X				



## *Egzamin próbny*

---

1 Który z mechanizmów zabezpieczeń zawsze działa niezależnie od warstwy transportowej. (Należy zaznaczyć wszystkie poprawne opcje).

- ☐ A. autoryzacja
- ☐ B. integralność danych
- ☐ C. uwierzytelnianie
- ☐ D. poufność

---

2 Dysponujemy deskryptorem wdrożenia zawierającym trzy poprawne elementy **<security-constraint>**, z których wszystkie definiują ograniczenie dostępu do zasobu A, wykorzystując przy tym następujące elementy **<auth-constraint>**:

```
<auth-constraint>
 <role-name>Janek</role-name>
</auth-constraint>
<auth-constraint />
<auth-constraint>
 <role-name>Alicja</role-name>
</auth-constraint>
```

Podaj, kto dysponuje dostępem do zasobu A:

- ☐ A. nikt
- ☐ B. każdy
- ☐ C. tylko Janek
- ☐ D. tylko Alicja
- ☐ E. tylko Janek i Alicja
- ☐ F. każdy z wyjątkiem Janka i Alicji

- 3** W której z poniższych operacji mogłyby pomóc mechanizmy zapewniania integralności danych, w jakie jest wyposażony kontener J2EE? (Należy zaznaczyć wszystkie poprawne opcje).
- ☐ A. Sprawdzenie, czy konkretny użytkownik ma prawa dostępu do konkretnej strony HTML.
  - ☐ B. Zapewnienie, że nikt nie może odczytać żądania HTTP przesyłanego z klienta na serwer.
  - ☐ C. Sprawdzenie, czy klient wysyłający żądanie kierowane do chronionej strony JSP jest skojarzony z rolą, która dysponuje prawami niezbędnymi do uzyskania dostępu do tej strony.
  - ☐ D. Zapewnienie, że nikt nie będzie w stanie zmodyfikować zawartości żądania HTTP podczas jego przesyłania z klienta na serwer.

- 4** Które z pól są wymagane w formularzu służącym do logowania, w przypadku wykorzystania uwierzytelniania bazującego na niestandardowym formularzu. (Należy zaznaczyć wszystkie poprawne opcje).
- ☐ A. **pw**
  - ☐ B. **id**
  - ☐ C. **j\_pw**
  - ☐ D. **j\_id**
  - ☐ E. **password**
  - ☐ F. **j\_password**

- 5** Które typy uwierzytelniania wymagają specjalnej wartości atrybutu `action` formularza HTML? (Należy zaznaczyć wszystkie poprawne opcje).
- ☐ A. Uwierzytelnianie proste.
  - ☐ B. Uwierzytelnianie bazujące na formularzu niestandardowym.
  - ☐ C. Uwierzytelnianie wykorzystujące skrót (DIGEST).
  - ☐ D. Uwierzytelnianie bazujące na protokole HTTPS.

- 
- 6 Które mechanizmy zabezpieczania można zaimplementować przy wykorzystaniu metod dostępnych w interfejsie **HttpServletRequest**? (Należy zaznaczyć wszystkie poprawne opcje).
- ☐ A. autoryzację
  - ☐ B. integralność danych
  - ☐ C. uwierzytelnianie
  - ☐ D. poufność
- 
- 7 Która z metod interfejsu **HttpServletRequest** jest najbliższej związana z elementem **<security-role-ref>**?
- ☐ A. **getHeader()**
  - ☐ B. **getCookies()**
  - ☐ C. **isUserInRole()**
  - ☐ D. **getUserPrincipal()**
  - ☐ E. **isRequestedSessionIDValid()**
- 
- 8 Które elementy deskryptora wdrożenia mogą zawierać podelementy **<transport-guarantee>**? (Należy zaznaczyć wszystkie poprawne opcje).
- ☐ A. **<auth-constraint>**
  - ☐ B. **<security-role-ref>**
  - ☐ C. **<form-login-config>**
  - ☐ D. **<user-data-constraint>**
- 
- 9 Stosowanie którego z mechanizmów uwierzytelniania zaleca się wyłącznie w przypadku jednoczesnego stosowania cookies lub śledzenia sesji SSL?
- ☐ A. Uwierzytelnianie proste.
  - ☐ B. Uwierzytelnianie bazujące na formularzu niestandardowym.
  - ☐ C. Uwierzytelnianie wykorzystujące skrót (DIGEST).
  - ☐ D. Uwierzytelnianie bazujące na protokole HTTPS.



BAR  
KAWOWY

## Egzamin próbny — odpowiedzi

- 1 Który z mechanizmów zabezpieczeń zawsze działa niezależnie od warstwy transportowej. (Należy zaznaczyć wszystkie poprawne opcje). *(Specyfikacja serwletów, rozdział 12.).*

- ☒ A. autoryzacja
- ☐ B. integralność danych
- ☐ C. uwierzytelnianie
- ☐ D. poufność

– Odpowiedź A jest poprawna. Po uwierzytelnieniu klienta proces autoryzacji jest w całości realizowany w ramach kontenera. Z kolei uwierzytelnianie, w zależności od sposobu skonfigurowania elementu `<auth-method>`, może mieć wpływ na warstwę transportową.

- 2 Dysponujemy deskryptorem wdrożenia zawierającym trzy poprawne elementy `<security-constraint>`, z których wszystkie definiują ograniczenie dostępu do zasobu A, wykorzystując przy tym następujące elementy `<auth-constraint>`: *(Specyfikacja serwletów, 12.8.1).*

```
<auth-constraint>
 <role-name>Janek</role-name>
</auth-constraint>
<auth-constraint />
<auth-constraint>
 <role-name>Alicja</role-name>
</auth-constraint>
```

Podaj, kto dysponuje dostępem do zasobu A:

- ☒ A. nikt
- ☐ B. każdy
- ☐ C. tylko Janek
- ☐ D. tylko Alicja
- ☐ E. tylko Janek i Alicja
- ☐ F. każdy z wyjątkiem Janka i Alicji

– Odpowiedź A jest poprawna. Zastosowanie pustego elementu `<auth-constraint>` powoduje, że żadne inne ustawienia odnoszące się do tego samego zasobu nie będą stosowane, przez co dostęp do niego nie będzie możliwy.

- 3** W której z poniższych operacji mogłyby pomóc mechanizmy zapewniania integralności danych, w jakie jest wyposażony kontener J2EE? (Należy zaznaczyć wszystkie poprawne opcje). (Specyfikacja serwletów, 12.1).
- ☐ A. Sprawdzenie, czy konkretny użytkownik ma prawa dostępu do konkretnej strony HTML.
  - ☐ B. Zapewnienie, że nikt nie może odczytać żądania HTTP przesyłanego z klienta na serwer. – Odpowiedź B opisuje przeznaczenie poufności.
  - ☐ C. Sprawdzenie, czy klient wysyłający żądanie kierowane do chronionej strony JSP jest skojarzony z rolą, która dysponuje prawami niezbędnymi do uzyskania dostępu do tej strony.
  - ☒ D. Zapewnienie, że nikt nie będzie w stanie zmodyfikować zawartości żądania HTTP podczas jego przesyłania z klienta na serwer. – Odpowiedź D jest poprawna. Ten cel zazwyczaj osiąga się poprzez zastosowanie protokołu HTTPS.

- 4** Które z pól są wymagane w formularzu służącym do logowania, w przypadku wykorzystania uwierzytelniania bazującego na niestandardowym formularzu. (Specyfikacja serwletów, 12.5.3).
- (Należy zaznaczyć wszystkie poprawne opcje).
- ☐ A. `pw`
  - ☐ B. `id`
  - ☐ C. `j_pw`
  - ☐ D. `j_id`
  - ☐ E. `password`
  - ☒ F. `j_password` – Odpowiedź F jest poprawna. Hasło użytkownika musi być podane w polu o nazwie `j_password`; dodatkowo w polu `j_username` należy podać nazwę użytkownika.

- 5** Które typy uwierzytelniania wymagają specjalnej wartości atrybutu `action` formularza HTML? (Specyfikacja serwletów, 12.5.3.1).
- (Należy zaznaczyć wszystkie poprawne opcje).
- ☐ A. Uwierzytelnianie proste.
  - ☒ B. Uwierzytelnianie bazujące na formularzu niestandardowym. – Odpowiedź B jest poprawna. By można było stosować uwierzytelnianie bazujące na własnym formularzu, atrybut `action` znacznika definiującego formularz musi mieć wartość `j_security_check`.
  - ☐ C. Uwierzytelnianie wykorzystujące skrót (DIGEST).
  - ☐ D. Uwierzytelnianie bazujące na protokole HTTPS.

- 6 Które mechanizmy zabezpieczania można zaimplementować przy wykorzystaniu metod dostępnych w interfejsie **HttpServletRequest**? (Należy zaznaczyć wszystkie poprawne opcje). (Specyfikacja serwletów, 12.3).
- ☒ A. autoryzację – Odpowiedź A jest poprawna. Metoda `isUserInRole()` może być stosowana programowo i pomagać w sprawdzeniu, czy rola użytkownika daje mu prawo dostępu do danego zasobu.
  - ☐ B. integralność danych
  - ☒ C. uwierzytelnianie – Odpowiedź C jest poprawna. Metoda `getRemoteUser()` może być stosowana programowo i pomagać w określeniu, czy klient został uwierzytelniony.
  - ☐ D. poufność
- 
- 7 Która z metod interfejsu **HttpServletRequest** jest najbliższej związana z elementem **<security-role-ref>**? (Specyfikacja serwletów, 12.3).
- ☐ A. `getHeader()`
  - ☐ B. `getCookies()`
  - ☒ C. `isUserInRole()` – Odpowiedź C jest poprawna. Element `<security-role-ref>` jest stosowany do odwzorowywania ról podanych na stałe w kodzie serwletu na role zadeklarowane w deskrytorze wdrożenia. Metoda `isUserInRole()` jest stosowana w serwletach do sprawdzenia zawartości elementów `<security-role-ref>`.
  - ☐ D. `getUserPrincipal()`
  - ☐ E. `isRequestedSessionIDValid()`
- 
- 8 Które elementy deskryptora wdrożenia mogą zawierać podelementy **<transport-guarantee>**? (Należy zaznaczyć wszystkie poprawne opcje). (Specyfikacja serwletów, 13.4).
- ☐ A. `<auth-constraint>`
  - ☐ B. `<security-role-ref>`
  - ☐ C. `<form-login-config>`
  - ☒ D. `<user-data-constraint>` – Odpowiedź D jest poprawna. Element `<transport-guarantee>` jest umieszczany wewnątrz elementu `<user-data-constraint>` i określa, czy daną kolekcję zasobów należy przesyłać przy użyciu bezpiecznego połączenia, na przykład takiego, jakie zapewnia protokół SSL.
- 
- 9 Stosowanie którego z mechanizmów uwierzytelniania zaleca się wyłącznie w przypadku jednoczesnego stosowania cookies lub śledzenia sesji SSL? (Specyfikacja serwletów, 12.5.3.1).
- ☐ A. Uwierzytelnianie proste.
  - ☒ B. Uwierzytelnianie bazujące na formularzu niestandardowym. – Odpowiedź B jest poprawna. Implementacja śledzenia sesji w przypadku stosowania logowania wykorzystującego niestandardowy formularz może być kłopotliwa, dlatego zalecane jest stosowanie odrębnego mechanizmu śledzenia sesji.
  - ☐ C. Uwierzytelnianie wykorzystujące skrót (DIGEST).
  - ☐ D. Uwierzytelnianie bazujące na protokole HTTPS.



## 13. Filtry i opakowania

### Potęga filtrów

Nawet nie MYŚL, że uda ci się skontaktować z mistrzem bez wcześniejszej rozmowy ze mną. To ja kontroluję, co i kto trafia do mistrza; kontroluję także to, co mistrz przekazuje innym...

Mówią, że zainspirował go wzorzec Intercepting Filter.



**Filtry pozwalają na przechwytywanie żądań.** A jeśli można przechwycić *żądanie*, to można także kontrolować *odpowiedź*. Ale najlepsze w tym wszystkim jest to, że serwlet nie ma o tym **najmniejszego pojęcia**. Serwlet nigdy nie wie, czy coś się zdarzyło w czasie pomiędzy odebraniem żądania przez kontener a wywołaniem metody `serv i ce()` serwletu. A co to oznacza dla Ciebie? Dłuższe wakacje. Ponieważ czas, który musiałbyś poświęcić na modyfikowanie tylko *jednego* z istniejących serwletów, możesz poświęcić na napisanie i skonfigurowanie filtra, który będzie miał wpływ na *wszystkie* Twoje serwlety. Może chciałbyś dodać opcję śledzenia żądań we *wszystkich* serwletach tworzących aplikację? Nie ma żadnego problemu. A może chciałbyś w określony sposób modyfikować wyniki generowane przez wszystkie *serwlety* wchodzące w skład aplikacji? Nie ma żadnego problemu. A co najlepsze — nie musisz przy tym w *żaden sposób* modyfikować kodu serwletów. Może się okazać, że filtry są najpotężniejszym narzędziem do tworzenia aplikacji internetowych, jakim dysponujesz.



## Filtry

- 3.3.** Opisz model przetwarzania żądań stosowany w kontenerze; napisz i skonfiguruj filtr; stwórz opakowanie żądania i odpowiedzi; na podstawie przedstawionego problemu opisz, w jaki sposób można w nim zastosować filtr lub opakowanie.
- 11.1.** Na podstawie opisu scenariusza składającego się z listy problemów wskaż wzorzec, który można zastosować w jego rozwiązaniu. Oto lista wzorców, które musisz znać: **Intercepting Filter** (filtr przechwytyjący), **Model-View-Controller** (Model-Widok-Kontroler), **Front Controller** (kontroler frontowy), **Service Locator** (lokalizator usług), **Business Delegate** (delegat biznesowy) oraz **Transfer Object** (obiekt transferu).
- 11.2.** Każdy z następujących wzorców: **Intercepting Filter** (filtr przechwytyjący), **Model-View-Controller** (Model-Widok-Kontroler), **Service Locator** (lokalizator usług), **Business Delegate** (delegat biznesowy) oraz **Transfer Object** (obiekt transferu) dopasuj do stwierdzeń opisujących potencjalne korzyści, które można dzięki nim osiągnąć.


## Uwagi wyjaśniające:

*Ten cel egzaminacyjny został w całości opisany w niniejszym rozdziale.*


*Filtry, które zostały opisane w niniejszym rozdziale, są przykładami wzorca Intercepting Filter (na pewno się tego nie spodziewałeś, prawda?). Zagadnieniami związanymi z wzorcami zajmiemy się dopiero w następnym rozdziale, lecz to właśnie w TYM rozdziale opiszemy projekt ilustrujący wspomniany wzorzec projektowy.*

## Rozbudowa możliwości całej aplikacji internetowej

Czasami niezbędne jest rozbudowanie systemu w sposób, który obejmuje wiele przypadków użycia lub żądań. Na przykład możesz chcieć śledzić czasy odpowiedzi systemu dla wszystkich występujących w nim interakcji z użytkownikiem.



Dzisiaj dostałam dobrą i złą wiadomość. Dobra jest taka, że nowa opcja „Dodaj nowy przepis” w naszej aplikacji piwnej jest bardzo popularna. Złą wiadomością jest to, że szefowie chcą, abyśmy śledzili wszystkich użytkowników używających tych serwletów...

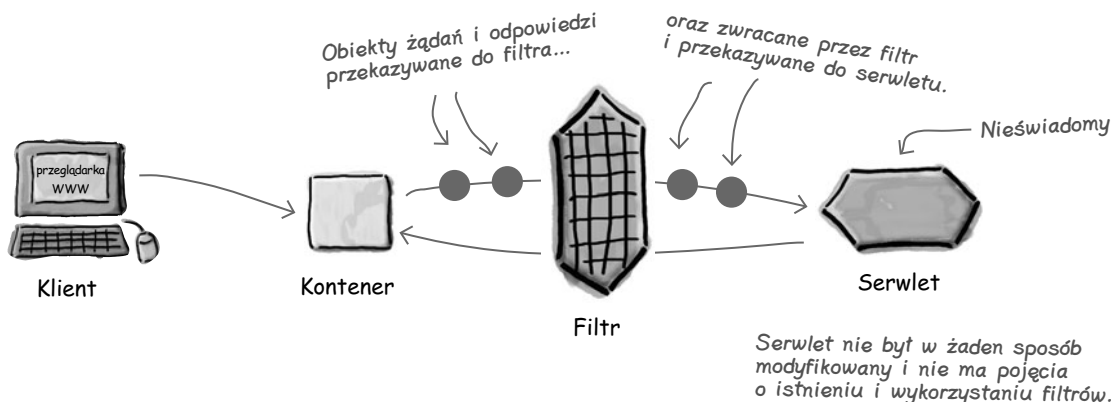


Nie mam zamiaru modyfikować tych wszystkich działających serwletów, zwłaszcza że jestem absolutnie pewna, że jak tylko dodam do nich opcję śledzenia użytkowników, to szefowie każą mi ją wyłączyć...

# A może by zastosować coś w rodzaju „filtra”?

Filtry są komponentami Javy — bardzo podobnymi do serwletów — których można używać do przechwytywania i przetwarzania żądań (*zanim* zostaną przekazane do serwletu) oraz przetwarzania odpowiedzi (już *po* wygenerowaniu odpowiedzi przez serwlet, ale *przed* wysłaniem jej do klienta).

Kontener decyduje, kiedy należy wywołać filtr na podstawie deklaracji podanych w deskrytorze wdrożenia. W deskrytorze wdrożeniowiec kojarzy filtry ze wzorcami URL. A zatem to właśnie wdrożeniowiec, a nie programista, decyduje, które podzbiory żądań i odpowiedzi będą przetwarzane przez konkretne filtry.



## Ciekawe zastosowania filtrów

Filtry **żądania** można zastosować do:

- wykonywania kontroli bezpieczeństwa,
- zmiany formatu nagłówek i zawartości żądań,
- kontroli lub rejestracji żądań.

Filtry **odpowiedzi** można zastosować do:

- kompresji strumienia odpowiedzi,
- dołączenia lub zmodyfikowania strumienia odpowiedzi,
- stworzenia całkowicie nowej odpowiedzi.



Oglądaj to!

**Istnieje tylko jeden interfejs używany do tworzenia filtrów — Filter.**

Nie ma czegoś takiego jak interfejsy *RequestFilter* lub *ResponseFilter* — istnieje tylko jeden interfejs *Filter*. Kiedy piszemy o filtrach odpowiedzi lub filtrach żądania, w rzeczywistości chodzi nam o sposób ZASTOSOWANIA filtra, a nie o interfejs używany do jego utworzenia. Z punktu widzenia kontenera istnieje tylko jeden rodzaj filtrów — filtrem jest wszystko, co implementuje interfejs *Filter*.

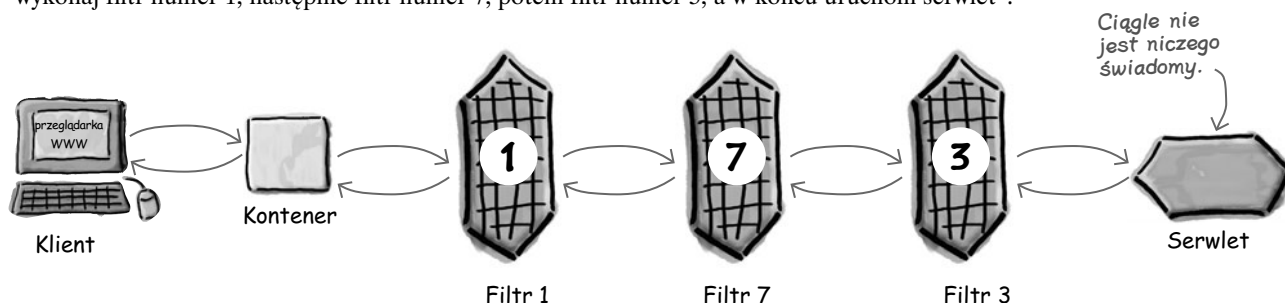
## Filtre są modularne i można je konfigurować w deskrytorze wdrożenia

Filtre można łączyć w łańcuch, tak aby były wykonywane sekwencyjnie jeden po drugim. Filtre zostały zaprojektowane w taki sposób, aby były całkowicie niezależne. Filtra nie obchodzi, czy jakkolwiek inny filtr został uruchomiony *wcześniej* ani czy jakiś filtr będzie uruchomiony *później*\*.

Kolejność, w jakiej będą wykonywane poszczególne filtry, jest określana w deskrytorze wdrożenia; tym zagadnieniem zajmiemy się w dalszej części rozdziału.

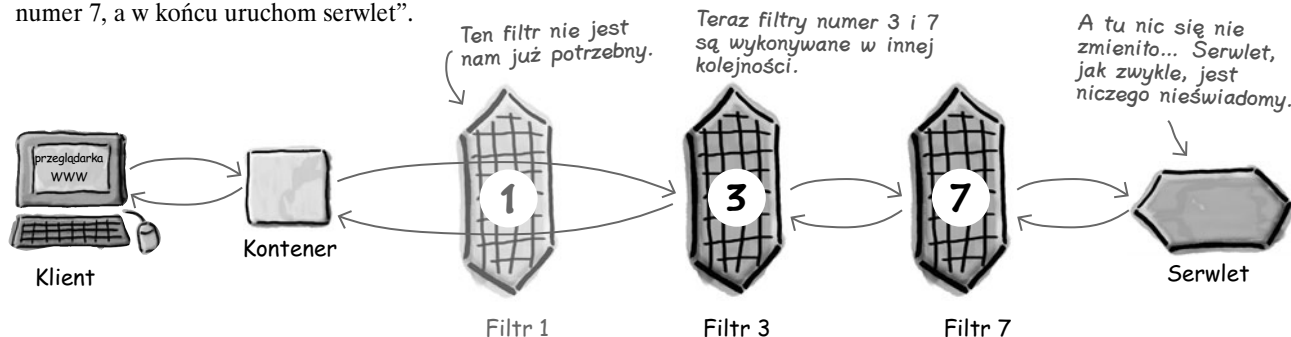
### Pierwsza wersja konfiguracji deskryptora wdrożenia:

W deskrytorze wdrożenia możemy połączyć filtry, nakazując: „Dla takich a takich adresów URL wykonaj filtr numer 1, następnie filtr numer 7, potem filtr numer 3, a w końcu uruchom serwlet”.



### Druga wersja konfiguracji deskryptora wdrożenia:

Potem szybka i prosta zmiana konfiguracji deskryptora pozwala usunąć niepotrzebny filtr i zmienić kolejność pozostałych: „Dla takich a takich adresów URL wykonaj filtr numer 3, następnie filtr numer 7, a w końcu uruchom serwlet”.



\* W tym przypadku nieco rozmiijamy się z prawdą. Często okazuje się, że wdrożeniowiec *musi* skonfigurować kolejność wykonywania poszczególnych filtrów na podstawie skutków ich działania. Na przykład dodanie znaku wodnego do obrazka po wcześniejszym zastosowaniu filtra kompresującego nie byłoby możliwe. W takim przypadku filtr dodający znaki wodne musiałby być użyty, zanim dane zostaną przekazane do filtra kompresującego. Kluczowy jest jednak fakt, iż Ty jako *programista* nie będziesz w swoim kodzie określać jakichkolwiek zależności pomiędzy filtrami.

Skoro filtry przypominają serwlety, to przypuszczam, że tak jak i one muszą być wywoływane przez kontener. Pewnie też mają swój własny cykl życia...



## Trzy podobieństwa filtrów i serwletów

To prawda, filtry działają w obrębie kontenera. Pod wieloma względami przypominają także serwlety. Oto kilka podobieństw pomiędzy tymi dwoma rodzajami „mieszkańców” kontenera:

### Kontener zna ich interfejs programowy

Filtry mają swój własny interfejs programowy. Jeśli klasa Javy implementuje interfejs **Filter**, podpisuje tym samym kontrakt z kontenerem i ze zwyczajnej klasy staje się oficjalnym filtrem J2EE. Inne metody interfejsu programowego filtrów pozwalają im na uzyskanie dostępu do obiektu `ServletContext` oraz na tworzenie połączeń z innymi filtrami.

### Kontener zarządza ich cyklem życia

Podobnie jak serwlety, także i filtry mają swój cykl życia (istnienia). Również dysponują metodami **init()** oraz **destroy()**. Podobnie jak serwlety, które posiadają metody **doPost()** oraz **doGet()**, filtry udostępniają metodę **doFilter()**.

### Są deklarowane w deskrytorze wdrożenia

Aplikacja internetowa może używać **wielu filtrów**, a odebranie jednego żądania może spowodować użycie kilku różnych filtrów. Właśnie w deskrytorze wdrożenia określa się, które filtry i w jakiej **kolejności** zostaną wykonane w odpowiedzi na konkretne żądanie.

## Tworzenie filtra służącego do śledzenia żądań

Naszym zadaniem jest rozbudowanie „aplikacji piwnej” w taki sposób, by wszystkie żądania odwołujące się do zasobów związanych z aktualizacją przepisów były rejestrowane. Poniżej przedstawiliśmy jedną z wielu możliwych postaci takiego filtra.

```
package com.example.web;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.HttpServletRequest;

public class PiwnyFiltrZadania implements Filter {

 private FilterConfig fc;

 public void init(FilterConfig konfig) throws ServletException {
 this.fc = konfig;
 }

 public void doFilter(ServletRequest zad,
 ServletResponse odp,
 FilterChain lancuch)
 throws ServletException, IOException {

 HttpServletRequest httpZad = (HttpServletRequest) zad;

 String nazwa = httpZad.getRemoteUser()

 if (nazwa != null) {
 fc.getServletContext().log("Użytkownik " + nazwa + " aktualizuje.");
 }

 lancuch.doFilter(zad, odp);

 }

 public void destroy() {
 // kończymy działanie filtra, zwalniamy używane zasoby
 }
}
```

Interfejsy `Filter` oraz `FilterChain` są zdefiniowane w pakiecie `javax.servlet`.

Każdy filtr musi implementować interfejs `Filter`.

Musisz zaimplementować metodę `init()`, zazwyczaj zapisujesz w niej jedynie obiekt konfiguracji.

Właściwe zadania filtra są realizowane w metodzie `doFilter()`. Zwróć uwagę na typy argumentów tej metody — nie są nimi obiekty żądania i odpowiedzi HTTP, lecz zwykłe obiekty typów `ServletRequest` oraz `ServletResponse`.

Jednak jesteśmy pewni, że można poprawnie rzutować te obiekty na odpowiednie typy HTTP — `HttpServletRequest` oraz `HttpServletResponse`.

W ten sposób wywołujemy kolejny filtr lub serwet — więcej informacji na ten temat podamy w dalszej części rozdziału.

Musisz zaimplementować metodę `destroy()`, jednak zazwyczaj będzie ona zupełnie pusta.

Filtry nie wiedzą, kto je może wywołać, ani kto zostanie wywołany w następnej kolejności.

# Cykl życia filtrów

Każdy filtr musi implementować trzy metody interfejsu `Filter` — `init()`, `doFilter()` oraz `destroy()`.

## Pierwsza jest metoda `init()`

Kiedy kontener dochodzi do wniosku o konieczności utworzenia obiektu filtra, możemy skorzystać z metody `init()`, która daje nam szansę wykonania dowolnych czynności jeszcze przed właściwym uruchomieniem tego filtra. Najbardziej popularny sposób implementacji tej metody przedstawiono na poprzedniej stronie — jej działanie sprowadza się do zapisania referencji do obiektu `FilterConfig`.

## Za wykonanie najtrudniejszych zadań odpowiada metoda `doFilter()`

Metoda `doFilter()` jest wywoływana za każdym razem, gdy kontener zdecyduje, że do obsługi nadesłanego żądania niezbędne jest zastosowanie filtra. Metoda ta pobiera trzy argumenty:

- typu `ServletRequest` (a nie `HttpServletRequest`);
- typu `ServletResponse` (a nie `HttpServletResponse`);
- typu `FilterChain`.

Metoda `doFilter()` jest naszą szansą na zaimplementowanie możliwości funkcjonalnych filtra. Jeśli nasz filtr ma zapisywać nazwy użytkowników w pliku tekstowym, to czynności te należy wykonywać właśnie w metodzie `doFilter()`. A może chcesz kompresować zwracaną odpowiedź? Także i to możesz zrobić w metodzie `doFilter()`.

## W końcu przychodzi czas na metodę `destroy()`

Kiedy kontener zdecyduje, że należy usunąć obiekt filtra, wywoła metodę `destroy()`, dając nam jednocześnie szansę na wykonanie wszelkich czynności, jakie w tej sytuacji uznamy za niezbędne.

## Nie ma niemądrych pytań

### ❓ A co to jest interfejs `FilterChain`?

❓ Interfejs `FilterChain` jest najlepszym typem spośród wszystkich konstrukcji związanych z tworzeniem i stosowaniem filtrów. Filtry zostały zaprojektowane jako modularne bloki, które można ze sobą łączyć na wiele sposobów w celu uzyskania pożądanej sekwencji wykonywanych czynności. I to właśnie interfejs `FilterChain` w dużej części sprawia, że uzyskanie takiego efektu jest możliwe. *FilterChain to ten gość, który wie, co ma się zdarzyć w następnej kolejności.* Wspominaliśmy już, że filtry (nie mówiąc o serwetach) nie powinny wiedzieć o innych filtrach biorących udział w obsłudze żądania... jednak ktoś musi znać kolejność, w jakiej należy wykonywać poszczególne filtry. Tym kimś jest właśnie obiekt typu `FilterChain`, którego działanie opiera się na informacjach podanych w elementach deskryptora wdrożenia.

Swoją drogą interfejs ten został umieszczony w tym samym pakiecie co interfejs `Filter` — w pakiecie `javax.servlet`.

### ❓ Zauważyłem, że w metodzie `doFilter()` przedstawionej w ostatnim przykładzie znalazło się wywołanie metody lancuch. `doFilter()`... Co wywołanie metody `doFilter()` robi wewnątrz metody `doFilter()`? Chyba nie chcemy stosować w tym momencie wywołań rekurencyjnych, nieprawdaż?

❓ Metoda `doFilter()` interfejsu `FilterChain` działa nieco inaczej niż tak samo nazwana metoda interfejsu `Filter`. Poniżej opisaliśmy podstawowe różnice pomiędzy tymi metodami.

Metoda `doFilter()` interfejsu `FilterChain` odpowiada za określenie filtra, którego metodę `doFilter()` należy wywołać w następnej kolejności (lub, jeśli łańcuch filtrów został już zakończony, za określenie serwletu, którego metodę `service()` należy wywołać). Z kolei metoda `doFilter()` interfejsu `Filter` odpowiada za przeprowadzenie filtrowania — czyli czynności, która była celem stworzenia filtra.

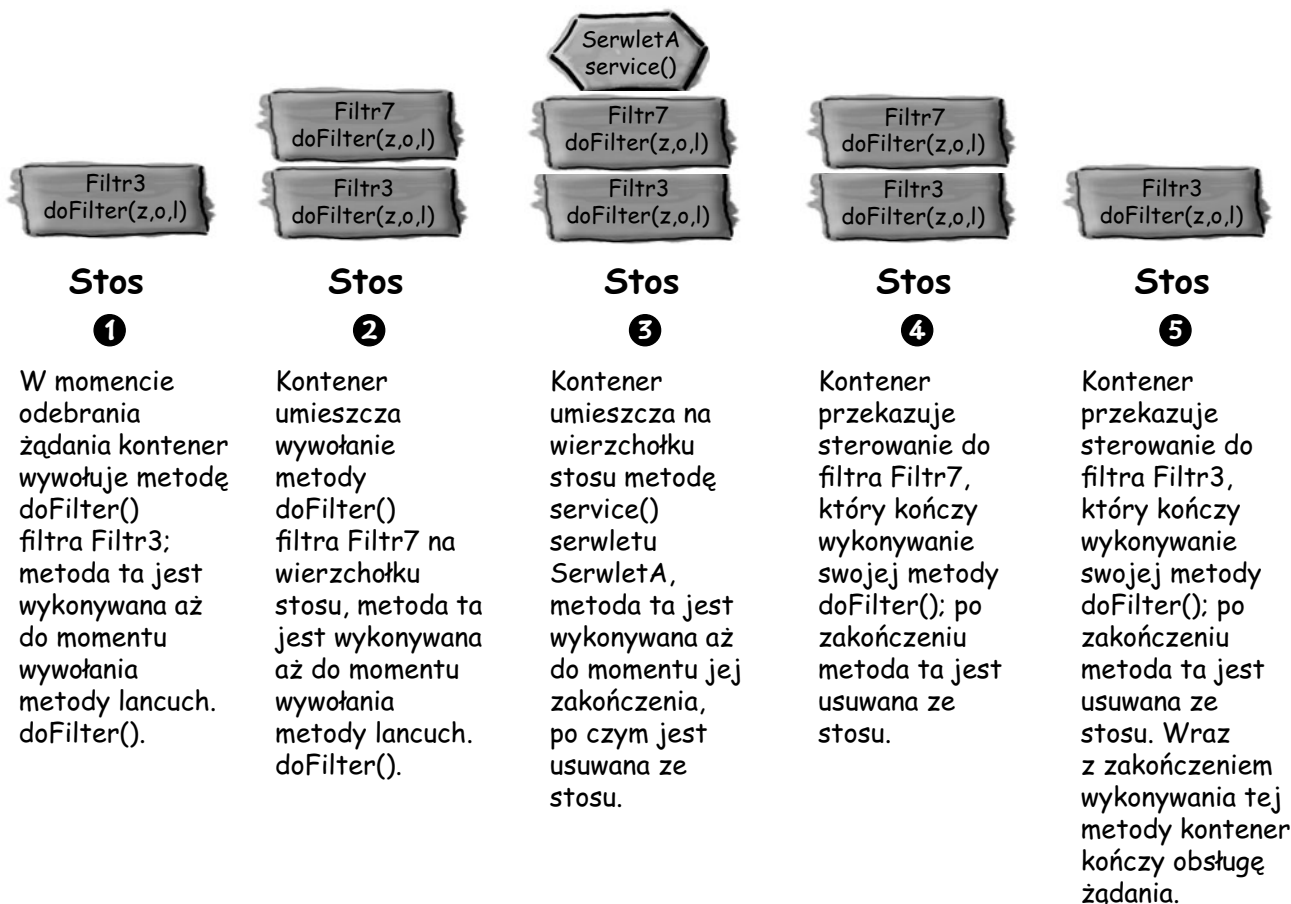
Oznacza to, że obiekt `FilterChain` może wywołać filtr BĄDŹ serwlet, w zależności od tego, w jakim miejscu łańcucha filtrów się znajdujemy. Na końcu tego łańcucha zawsze znajduje się serwlet lub strona JSP (czyli serwlet wygenerowany na podstawie strony JSP), oczywiście jeśli tylko kontener jest w stanie odwzorować żądany adres URL na serwlet lub stronę JSP. (Jeśli kontener nie jest w stanie odnaleźć zasobu wskazanego w żądaniu, to filtr nigdy nie zostanie wykonany).

## Myśl o filtrach jako o czymś, co można „układać jedno na drugim”

Specyfikacja serwletów nie określa, w jaki sposób kontener ma implementować metodę `lancuch.doFilter(zad, odp)`. Jednak w praktyce proces obsługi filtrów można sobie wyobrazić jako sekwencję wywołań zwyczajnych metod umieszczonych na jednym **stosie**. Doskonale wiemy, że tak naprawdę w kontenerze wykonywanych jest znacznie więcej operacji, jednak nie obchodzi nas to, o ile tylko jesteśmy w stanie przewidzieć, w jaki sposób będą wykonywane nasze filtry; a właśnie na to pozwala nam *umowny* stos.

### Przykład umownego stosu wywołań

W poniższym przykładzie żądanie odnoszące się do serwletu `SerwletA` będzie przetworzone przez dwa filtry — `Filtr3` oraz `Filtr7`.



„Umowny stos” jest jedynie sposobem, w jaki możemy sobie wyobrazić wywołania kolejnych filtrów tworzących łańcuch. Nie wiemy ani nie obchodzi nas, w jaki sposób kontener implementuje te mechanizmy — jednak wyobrażenie sobie, iż jest w tym celu używany stos, pomaga przewidzieć sposób działania łańcucha filtrów.

# Deklarowanie i określanie kolejności filtrów

Podczas deklarowania filtrów w deskrypcorze wdrożenia zazwyczaj wykonujemy trzy czynności:

- deklarujemy filtr,
- odwzorowujemy filtr na zasoby aplikacji internetowej, do których odwołanie ma powodować wywołanie filtra,
- określamy porządek tych odwzorowań, definiując w ten sposób sekwencję wywołań filtrów.

## Deklaracja filtra

```
<filter>
 <filter-name>ZadaniePiwa</filter-name>
 <filter-class>com.example.web.PiwnyFiltrZadania</filter-class>
 <init-param>
 <param-name>NazwaPlikuDziennika</param-name>
 <param-value>ListaUzytkownikow.txt</param-value>
 </init-param>
</filter>
```

## Deklaracja odwzorowania filtra na wzorec URL

```
<filter-mapping>
 <filter-name>ZadaniePiwa</filter-name>
 <url-pattern>*.do</url-pattern>
</filter-mapping>
```

## Deklaracja odwzorowania filtra na nazwę serwletu

```
<filter-mapping>
 <filter-name>ZadaniePiwa</filter-name>
 <servlet-name>SerwletDoradcy</servlet-name>
</filter-mapping>
```

## Reguły dotyczące elementu <filter>

- element <filter-name> jest obowiązkowy,
- element <filter-class> jest obowiązkowy,
- element <init-param> jest opcjonalny, liczba tych elementów może być nieograniczona.

## Reguły dotyczące elementu <filter-mapping>

- element <filter-name> jest obowiązkowy i służy on do wskazania odpowiedniego elementu <filter>,
- koniecznie należy podać jeden z elementów <url-pattern> bądź <servlet-name>,
- element <url-pattern> określa dla których zasobów aplikacji internetowej będzie stosowany dany filtr,
- element <servlet-name> określa nazwę konkretnego zasobu aplikacji internetowej, dla którego będzie używany dany filtr.

### TO WAŻNE: Reguły używane przez kontener do określania kolejności wywoływania filtrów:

Jeśli dla jakiegoś zasobu aplikacji zdefiniowano więcej niż jeden filtr, to kontener postępuje według poniższych zasad:

- 1) W pierwszej kolejności odnajdywane są WSZYSTKIE filtry, których wzorce URL odpowiadają adresowi żądanego zasobu. A zatem NIE są to te same reguły używane przez kontener do wybrania „zwycięzcy”, który obsłuży żądanie odwołujące się do konkretnego zasobu. Różnica polega na tym, iż w łańcuchu zostaną umieszczone WSZYSTKIE filtry o pasującym wzorcu URL! Wybrane wzorce są umieszczane w łańcuchu w kolejności, w jakiej zostały zdefiniowane w deskrypcorze wdrożenia.
- 2) Kiedy już wszystkie pasujące filtry zostaną umieszczone w łańcuchu, kontener odnajdzie według tych samych reguł wszystkie filtry z pasującymi ustawieniami elementu <servlet-name>.

Czyż TO nie jest typowe?  
Dają nam możliwość filtrowania żądań  
przesyłanych przez klienta, zapominając  
jednocześnie o tych wszystkich żądaniach, które  
SAMI generujemy — o przekierowaniach i operacjach  
przydziału żądań. Rany! Oni traktują przydzielanie  
żądań jako jakąś podrzędną technikę  
wywoływania?!



## Najnowsze doniesienia — w wersji 2.4 filtry można stosować także dla przydziałów żądań

Zastanów się. Bez dwóch zdań to dobrze, że filtry można stosować do przetwarzania żądań przesyłanych bezpośrednio przez klienta. Ale co z zasobami, do których odwołujemy się za pomocą mechanizmów **przekazywania dalej**, **dołączania**, **przydziału żądań** oraz obsługi błędów? To wszystko zawdzięczamy wersji 2.4 specyfikacji serwetów.

### Deklaracja odwzorowania filtra na zasoby wywoływane przy użyciu mechanizmów przydzielania żądań

```
<filter-mapping>
 <filter-name>FiltrMonitorujący</filter-name>
 <url-pattern>*.do</url-pattern>
 <dispatcher>REQUEST</dispatcher>

 i (lub)

 <dispatcher>INCLUDE</dispatcher>

 i (lub)

 <dispatcher>FORWARD</dispatcher>

 i (lub)

 <dispatcher>ERROR</dispatcher>
</filter-mapping>
```

### Reguły deklarowania

- element `<filter-name>` jest obowiązkowy,
- konieczne jest użycie jednego z elementów: `<url-pattern>` lub `<servlet-name>`,
- można użyć od 0 do 4 elementów `<dispatcher>`,
- wartość `REQUEST` aktywuje filtr w przypadku żądań przesyłanych przez klienta — jeśli nie zostanie użyty żaden element `<dispatcher>`, to domyślnie przyjmowana jest wartość `REQUEST`,
- wartość `INCLUDE` aktywuje filtr w przypadku żądań przydzielanych na skutek wywołania metody `include()`,
- wartość `FORWARD` aktywuje filtr w przypadku żądań przydzielanych na skutek wywołania metody `forward()`,
- wartość `ERROR` aktywuje filtr w przypadku obsługi zasobów, do których odwołujemy się na skutek wystąpienia błędów.

Ćwiczenie z konfiguracji filtrów



Zaostrz ołówek

Opierając się na poniższym fragmencie deskryptora wdrożenia, podaj sekwencję, w jakiej dla poszczególnych żądań będą wywoływane filtry. Przyjmij przy tym, że filtry od `Filtr1` do `Filtr5` zostały zadeklarowane poprawnie. (Rozwiązanie ćwiczenia można znaleźć pod koniec rozdziału).

```
<filter-mapping>
 <filter-name>Filtr1</filter-name>
 <url-pattern>/Przepisy/*</url-pattern>
</filter-mapping>

<filter-mapping>
 <filter-name>Filtr2</filter-name>
 <servlet-name>/Przepisy/ListaChmielu.do</servlet-name>
</filter-mapping>

<filter-mapping>
 <filter-name>Filtr3</filter-name>
 <url-pattern>/Przepisy/Dodaj/*</url-pattern>
</filter-mapping>

<filter-mapping>
 <filter-name>Filtr4</filter-name>
 <servlet-name>/Przepisy/Modyfikuj/RdzPrzepisu.do</servlet-name>
</filter-mapping>

<filter-mapping>
 <filter-name>Filtr5</filter-name>
 <url-pattern>/*</url-pattern>
</filter-mapping>
```

Ścieżka żądania	Sekwencja filtrów:
/Przepisy/RaportChmielu.do	Filtry:
/Przepisy/ListaChmielu.do	Filtry:
/Przepisy/Modyfikuj/RdzPrzepisu.do	Filtry:
/ListaChmielu.do	Filtry:
/Przepisy/Dodaj/DodajPrzepis.do	Filtry:

## Kompresja wyników za pomocą filtra operującego na odpowiedzi

We wcześniejszej części rozdziału przedstawiliśmy bardzo prosty filtr operujący na *żądaniu*. Filtry operujące na *odpowiedziach* są co prawda nieco trudniejsze w tworzeniu, ale też oferują wprost niezwykle możliwości. Za ich pomocą można wykonywać pewne operacje na całej odpowiedzi już po zakończeniu działania serwletu, a jeszcze przed jej przesłaniem do klienta. A zatem, zamiast wykonywać pewne operacje na samym początku — czyli *zanim* serwlet obsłużył żądanie — wykonujemy je na samym końcu — czyli *po* tym, jak serwlet odebrał żądanie i wygenerował odpowiedź.

No... *coś w tym stylu...* zastanów się nad tym. Filtry są zawsze wywoływane w określonej sekwencji *przed* wywołaniem serwletu. Nie ma czegoś takiego jak filtr, który jest wywoływany *po* serwlecie. Ale czy pamiętasz rysunek przedstawiający umowny stos wywołań? **Filtr dostanie jeszcze jedną szansę na powiedzenie swojego „ostatniego słowa” po zakończeniu działania serwletu i usunięciu go ze stosu (wirtualnego)!**

Mój pierwszy filtr spodobał się szefowi tak bardzo, że kazał mi napisać kolejny. Przepustowość firmowego łącza z Internetem jest wykorzystywana w coraz większym stopniu, więc teraz szef chce, żeby wszystkie strumienie wynikowe były *kompresowane...*

Wydaje się, że także w tym przypadku zastosowanie filtra jest rozwiązaniem właściwym... Jednak ponieważ mamy do czynienia z odpowiedziami, muszę umieścić kod kompresujący ZA wywołaniem metody `lancuch.doFilter()...`



# Architektura filtrów operujących na odpowiedzi

Rachel mówi o podstawowej strukturze operacji wykonywanych w metodzie `doFilter()` — w pierwszej kolejności podejmuje działania związane z żądaniem, następnie wywołuje metodę `lancuch.doFilter()`, by wreszcie, już po zakończeniu wykonywania serwletu (oraz innych filtrów tworzących łańcuch), ponownie przekazać sterowanie do początkowej metody `doFilter()`, która tym razem może się zająć obsługą odpowiedzi.

## Pseudokod filtra kompresującego zawartość odpowiedzi autorstwa Rachel

```
class FiltrKompresujacy implements Filter {
 init();

 public void doFilter(zadanie, odpowiedz, lancuch) {
 // tu umieścimy kod obsługujący żądanie
 lancuch.doFilter(zadania, odpowiedz);
 // obsługa kompresji
 }

 destroy();
}
```

*W tym miejscu wkracza od akcji serwlet.*

*Teraz, kiedy serwer już zrobił to, co do niego należało, możemy się zająć kompresją wygenerowanej przez niego odpowiedzi...*

## Umowny stos wywołań

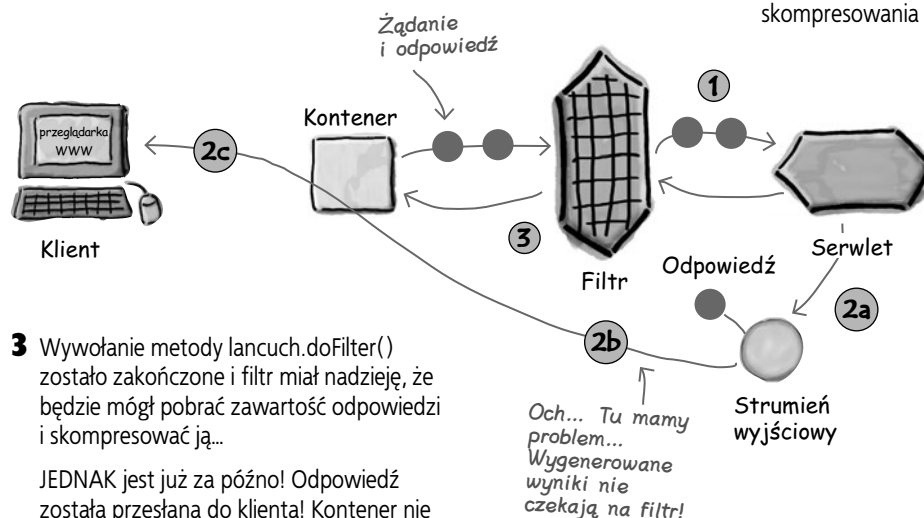


## Czy to naprawdę jest aż tak proste?

Czy kompresja zawartości odpowiedzi naprawdę sprowadza się jedynie do poczekania na zakończenie działania serwletu i kompresji wygenerowanych przez niego wyników? W końcu metoda `doFilter()` filtra ma dostęp do tego samego obiektu odpowiedzi, który jest dostępny w serwlecie, a zatem, teoretycznie rzecz biorąc, filtr powinien mieć dostęp do zawartości odpowiedzi...

```
public void doFilter(zadanie, odpowiedz, lancuch) {
 // tu umieścimy kod obsługujący żądanie
 lancuch.doFilter(zadanie, odpowiedz); ❶ ❷
 // obsługa kompresji ❸
}
```

**1** Filtr przekazuje obiekty żądania i odpowiedzi do serwletu, po czym cierpliwie czeka na możliwość skompresowania odpowiedzi.



**2a** Serwlet realizuje swoje zadania, generuje wyniki i pozostaje nieświadomy tego, iż są to te same wyniki, które mają zostać skompresowane.

**2b** Wyniki są przekazywane do kontenera, a następnie...

**2c** są przesyłane do klienta! Hmm... to może być pewien problem. Filtr oczekiwał, że będzie miał szansę wykonania pewnych operacji na zawartości odpowiedzi (skompresowania jej), zanim trafi ona do klienta.

**3** Wywołanie metody `lancuch.doFilter()` zostało zakończone i filtr miał nadzieję, że będzie mógł pobrać zawartość odpowiedzi i skompresować ją...

JEDNAK jest już za późno! Odpowiedź została przesłana do klienta! Kontener nie buforuje odpowiedzi, czego oczekiwał od niego filtr. A zatem w chwili, gdy metoda `doFilter()` filtra ponownie znajdzie się na wierzchołku (umownego) stosu wywołań, będzie już **zbyt późno, by filtr mógł w jakikolwiek sposób zmodyfikować zawartość odpowiedzi**.

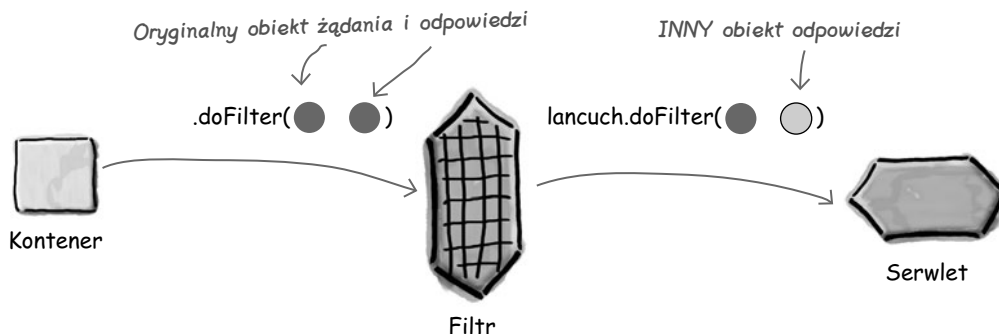
Och... Tu mamy problem... Wygenerowane wyniki nie czekają na filtr!

### Odpowiedź została już wysłana

Nic z tego – to nie będzie działać.  
Nie mogę kompresować czegoś, co wychodzi z serwletu, gdyż wtedy jest już zbyt późno. Wyniki są przesyłane bezpośrednio z serwletu do klienta. Ale cała idea naszego filtra miała polegać właśnie na kompresji zawartości odpowiedzi. A zatem w jaki sposób mogę przejąć kontrolę nad zawartością odpowiedzi, ZANIM zostanie ona przestana do klienta?

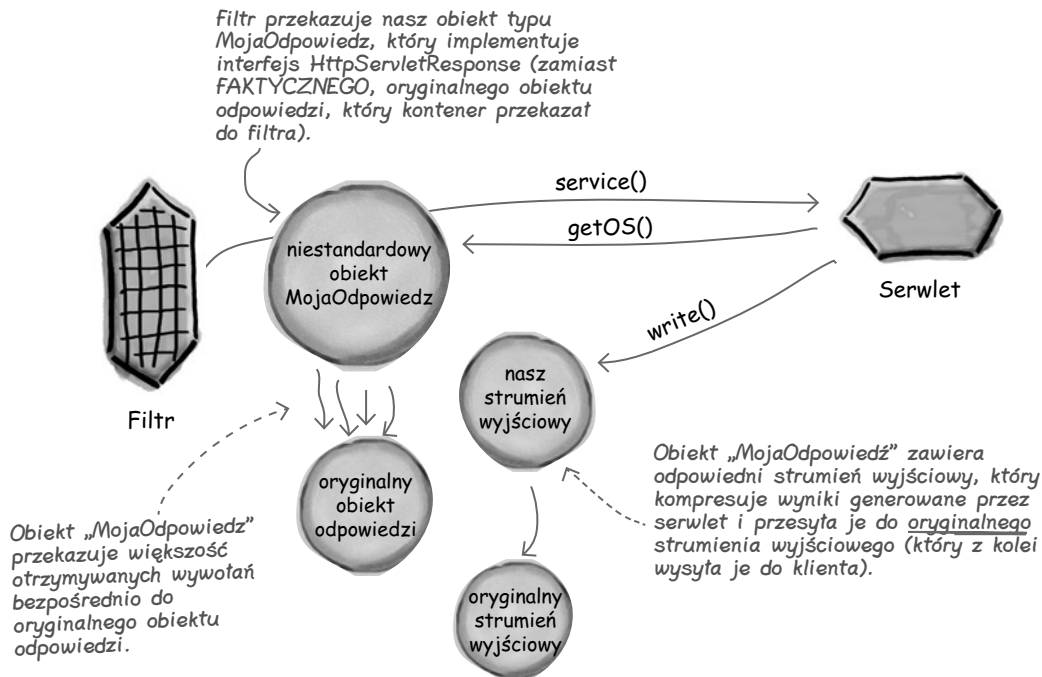


Zastanów się nad tym przez chwilę...  
serwlet otrzymuje od obiektu odpowiedzi strumień wyników lub obiekt Writer. A co się stanie, jeśli zamiast przekazania do serwletu PRAWDZIWEGO obiektu odpowiedzi, filtr przekaże do niego specjalnie przygotowany obiekt zawierający strumień wyjściowy, który możemy kontrolować? W końcu nikt nie mówi, że filtr w momencie wywołania metody `lancuch.doFilter()` ma przekazywać do serwletu PRAWDZIWIY obiekt odpowiedzi...



## Możemy zaimplementować WŁASNĄ odpowiedź

Kontener implementuje już interfejs `HttpServletResponse`; to właśnie obiekty tego typu są przekazywane w wywołaniach metod `doFilter()` oraz `service()`. Jednak, aby uruchomić filtr kompresujący, musimy stworzyć *własną* implementację interfejsu `HttpServletResponse` i to właśnie obiekt tej klasy przekazać w wywołaniu metody `lancuch.doFilter()`. Ta nasza implementacja musi także zawierać *odpowiedni strumień wyjściowy*, gdyż to właśnie on jest niezbędny do osiągnięcia zamierzonego celu — przechwycenia wyników działania serwletu już po ich wygenerowaniu, lecz *przed* przesłaniem do klienta.



**P.** Filtre przekazują do następnego ogniwa łańcucha obiekty typów `ServletRequest` oraz `ServletResponse`, a **NIE** `HttpServletRequest` oraz `HttpServletResponse`! Dlaczego zatem piszecie o implementacji interfejsu `HttpServletResponse`?

**U.** Filtre zostały zaprojektowane w sposób gwarantujący możliwie dużą uniwersalność, zatem z formalnego punktu widzenia masz rację. Gdybyśmy przypuszczali, że jeden z naszych filtrów może być używany przez aplikację *inną* niż internetowa, to nie implementowalibyśmy w nim interfejsu przeznaczonego dla aplikacji internetowych — `HttpServletResponse` — lecz bardziej ogólny — `ServletResponse`. Jednak obecnie szansa na to, że ktoś będzie tworzył serwlety używane przez aplikacje inne niż internetowe, jest praktycznie równa zero, dlatego też w ogóle nie przejmujemy się tym problemem. Co więcej, ponieważ interfejs `HttpServletResponse` dziedziczy po interfejsie `ServletResponse`, przekazywanie obiektu typu `HttpServletResponse` w miejscu, gdzie oczekiwany jest obiekt typu `ServletResponse`, nie stanowi najmniejszego problemu.

`HttpServletResponse` to taki skomplikowany interfejs... gdyby tylko istniał jakiś sposób pozwalający uniknąć implementacji wszystkich jego metod i przekazywania wywołań do prawdziwego obiektu odpowiedzi...



### Ona nie wie jeszcze o istnieniu klas „opakowań”

Stworzenie własnej implementacji interfejsu `HttpServletResponse` byłoby bardzo uciążliwe. Zwłaszcza jeśli weźmiemy pod uwagę, że zależy nam wyłącznie na zaimplementowaniu zaledwie *kilku* spośród jego metod. Co więcej, ponieważ interfejs `HttpServletResponse` dziedziczy po interfejsie `ServletResponse`, implementacja własnej klasy odpowiedzi wymagałaby zaimplementowania *wszystkich* metod obu tych interfejsów — zarówno `HttpServletResponse`, jak i `ServletResponse`.

Jednak na nasze szczęście ktoś w firmie Sun już to za nas zrobił, tworząc klasę pomocniczą implementującą interfejs `HttpServletResponse`. Wszystkie metody tej klasy przekazują wywołania do faktycznego obiektu odpowiedzi utworzonego przez kontener.

### interfejs `ServletResponse`

(`javax.servlet.ServletResponse`)

<<interfejs>> <b><i>ServletResponse</i></b>
<code>getBufferSize()</code> <code>setContentType()</code> <code>getOutputStream()</code> <code>getWriter()</code> <i>// WIELE innych metod</i>

### interfejs `HttpServletResponse`

(`javax.servlet.http.HttpServletResponse`)

<<interfejs>> <b><i>HttpServletResponse</i></b>
<code>addCookie()</code> <code>addDateHeader()</code> <code>addHeader()</code> <code>encodeRedirectURL()</code> <code>encodeURL()</code> <code>sendError()</code> <code>sendRedirect()</code> <code>setDateHeader()</code> <code>setHeader()</code> <code>setStatus()</code> <i>// inne metody</i>

Pamiętaj, że aby zaimplementować interfejs `HttpServletResponse`, trzeba zaimplementować **WSZYSTKIE** metody zdefiniowane zarówno w nim, jak i w jego interfejsie bazowym — `ServletResponse`.

## Opakowania są świetne!

Klasy opakowań dostępne w API serwetów budzą zachwyt i szacunek — implementują one wszystkie niezbędne metody, przekazując wywołania do faktycznych obiektów odpowiedzi i żądania. A zatem na nas spoczywa jedynie konieczność stworzenia klasy dziedziczącej po odpowiedniej klasie opakowania i nadpisania wybranych metod tej klasy — tych, które są niezbędne do osiągnięcia zamierzonych celów.

Oczywiście klasy pomocnicze można także znaleźć w J2SE API, gdzie ich przykładami mogą być klasy odbiorców zdarzeń używane przy tworzeniu graficznego interfejsu użytkownika. Można je także znaleźć w API JSP — są nimi klasy pomocnicze służące do obsługi znaczników niestandardowych. Niemniej jednak, choć zarówno wspomniane przed chwilą klasy, jak i klasy opakowań można określić ogólnie jako „klasy pomocnicze”, to jednak opakowania różnią się od innych klas, gdyż... *opakowują* obiekty tego typu, który implementują. Innymi słowy, klasy te nie tylko dostarczają *implementacji interfejsu*, lecz w rzeczywistości zawierają referencję do obiektu tego samego typu interfejsu i przekazują do niego wywołania wszystkich metod. (Swoją drogą te „opakowania” nie mają absolutnie nic wspólnego z „klasami opakowań” dostępnymi w J2SE i stanowiącymi obiektową reprezentację typów podstawowych, czyli klasami Integer, Boolean, Double itd.).

Tworzenie wyspecjalizowanych wersji obiektów żądania i odpowiedzi jest tak częstą praktyką w procesie stosowania filtrów, że firma Sun stworzyła aż cztery klasy „pomocnicze”:

- ServletRequestWrapper
- HttpServletRequestWrapper
- ServletResponseWrapper
- HttpServletResponseWrapper



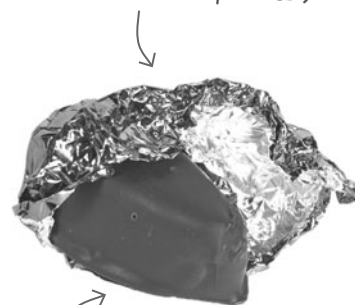
Oglądaj to!

**Na egzaminie może się pojawić pytanie o „dekorator”, choć to zagadnienie nie znalazło się na oficjalnej liście celów egzaminacyjnych.**

Jeśli znasz zagadnienia związane ze zwyczajnymi wzorcami projektowymi stosowanymi w języku Java (które nie mają nic wspólnego z technologią J2EE), zapewne dostrzeżesz łudzące podobieństwo omawianych „opakowań” do implementacji wzorca nazwanego Decorator (dekorator) — niektórzy stosują jednak alternatywną nazwę Wrapper (opakowanie). Wzorec ten dekoruje (lub opakowuje) pewien typ obiektu, udostępniając jego implementację o „poszerzonych” możliwościach. Oznacza to, że nowy obiekt robi wszystko to, co robi obiekt „dekorowany”, a przy tym dodaje pewne nowe możliwości.

To tak jakby stwierdzić: „Jestem LEPSZĄ wersją obiektu, który opakowuję — robię wszystko to, co on, a nawet więcej”. Jedną z cech charakterystycznych dekoratorów (opakowań) jest to, iż przekazują one wszystkie wywołania do dekorowanego (opakowywanego) obiektu, a nie implementują tych możliwości samodzielnie.

Opakowanie (Twój obiekt odpowiedzi)



Obiekt opakowany (oryginalny obiekt odpowiedzi utworzony przez kontener)

Zawsze, gdy będziesz chciał stworzyć własny niestandardowy obiekt żądania lub odpowiedzi, należy stworzyć klasę dziedziczącą po jednej z pomocniczych klas „opakowań”.

Opakowanie zawiera RZECZYWISTY obiekt żądania lub odpowiedzi i przekazuje do niego wszystkie wywołania metod, a jednocześnie pozwala na wykonywanie własnych operacji niezbędnych do odpowiedniego przetworzenia żądania lub odpowiedzi.

## Dodanie prostego opakowania do projektu aplikacji

Zmodyfikujmy pierwszą wersję pseudokodu Rachel, dodając do niego opakowania.

### Druga wersja filtra kompresującego (pseudokod)

```
class OpakowanieKompresjiOdpowiedzi extends HttpServletResponseWrapper {
 // przesłaniamy metody, które chcemy zmodyfikować
}
```

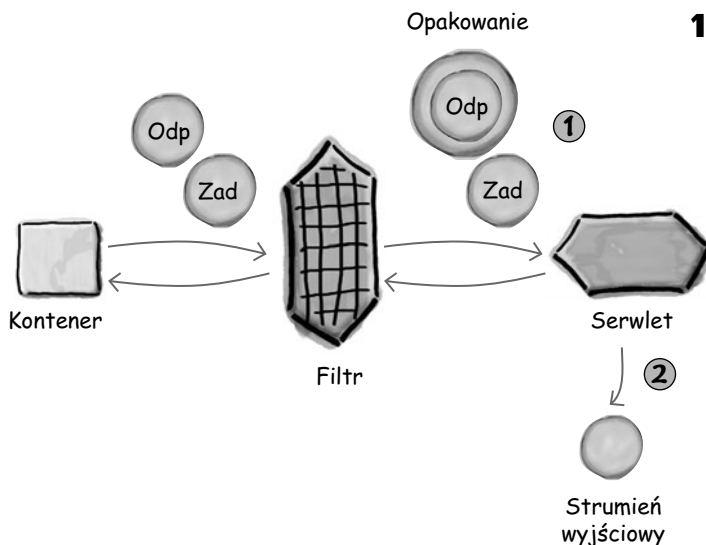
*W celu zrealizowania naszych diabelskich celów stwórzmy klasę dziedziczącą po tej klasie opakowania...*

*Już niedługo naprawdę zajmiemy się nadpisywaniem metod!*

```
class FiltrKompresujacy implements Filter {
 public void init(FilterConfig cfg) { }
 public void doFilter(zadanie, odpowiedz, lancuch) {
 OpakowanieKompresjiOdpowiedzi opakowanieOdp
 = new OpakowanieKompresjiOdpowiedzi(odpowiedz);
 lancuch.doFilter(zadanie, opakowanieOdp);
 // tu wykonujemy kompresję
 }
 public void destroy() { }
}
```

*Oto akt „opakowywania” odpowiedzi naszą klasą.*

*Teraz przesyłamy nasz obiekt odpowiedzi do dalszej części łańcucha. Żaden ze znajdujących się tam komponentów nie będzie wiedzieć, że używany przez niego obiekt odpowiedzi nie jest standardowy.*



**1** Filtr przekazuje do serwletu obiekt żądania oraz niestandardowy obiekt odpowiedzi.

**2** Ponieważ w klasie opakowania nie nadpisaliśmy żadnej metody, strumień wyjściowy nie jest w żaden sposób modyfikowany... przynajmniej na razie.

## Dodanie opakowania strumienia wyjściowego

Dodajmy zatem drugie opakowanie...

### Filtr kompresujący, wersja 3. (pseudokod)

```
class OpakowanieKompresjiOdpowiedzi extends HttpServletResponseWrapper {
 public ServletOutputStream getOutputStream() throws... {
 ...
 strumienGzipOS = new GzipSOS(odp.getOutputStream());
 return strumienGzipOS;
 }
 // może nadpiszemy jeszcze inne metody
}
```

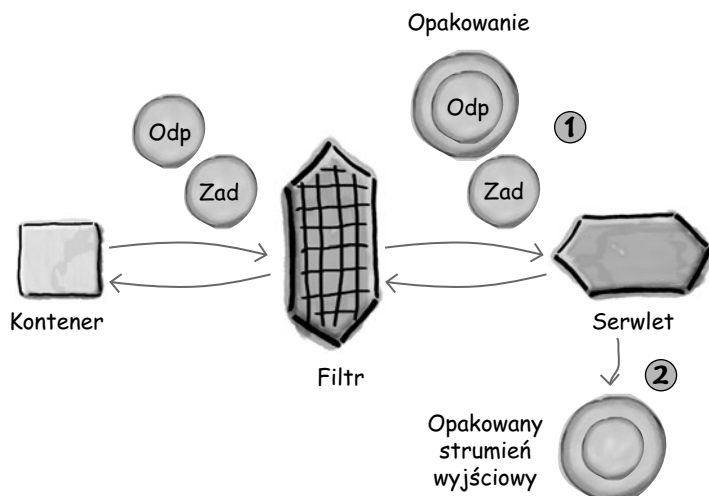
Nadpisz tę metodę, aby zwrócić niestandardowy strumień wyjściowy.

„Opakowanie” strumienia ServletOutputStream przy wykorzystaniu niestandardowej klasy opakowania. Na razie załóżmy, że klasa GzipSOS dziedziczy po klasie ServletOutputStream.

Wszystkim, którzy nas o to poproszą, zwracamy „specjalny” obiekt ServletOutputStream.

```
class FiltrKompresujacy implements Filter {
 public void init(FilterConfig cfg) { }
 public void doFilter(zadanie, odpowiedz, lancuch) {
 OpakowanieKompresjiOdpowiedzi opakowanie0dp
 = new OpakowanieKompresjiOdpowiedzi(odpowiedz);
 lancuch.doFilter(zadanie, opakowanie0dp);
 // tu wykonujemy kompresję
 }
 public void destroy() { }
}
```

**1** Filtr przekazuje do serwletu obiekt żądania oraz **niestandardowy** obiekt odpowiedzi. Przekazany obiekt odpowiedzi dysponuje specjalną metodą `getOutputStream()`.



**2** Kiedy serwlet poprosi o strumień wyjściowy, nie będzie WIEDZIAŁ, że strumień, który dostanie, będzie „specjalny”.

# Prawdziwy kod filtra kompresji odpowiedzi

Nadszedł czas na przedstawienie kodu filtra. Na końcu niniejszego rozdziału przedstawiamy kod filtra kompresującego odpowiedzi, jak również używanego przez niego „opakowania”. Przedstawiony kod jest nieco bardziej rozbudowany w stosunku do tego, o czym pisaliśmy wcześniej, jednak w przeważającej mierze jest to zwyczajny kod Javy.

Przedstawiony poniżej filtr potrafi kompresować zawartość odpowiedzi. Filtry tego typu z powodzeniem można stosować do kompresji dowolnej zawartości tekstowej, takiej jak dokumenty HTML; nie można ich jednak używać do przetwarzania większości formatów multimedialnych, takich jak PNG lub MPEG, gdyż taka zawartość już jest kompresowana.

```
package com.example.web;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.zip.GZIPOutputStream;

public class FiltrKompresujacy implements Filter {

 private ServletContext ctx;
 private FilterConfig cfg;

 public void init(FilterConfig cfg) throws ServletException {
 this.cfg = cfg;
 ctx = cfg.getServletContext();
 ctx.log("Filtr " + cfg.getFilterName() + " został zainicjalizowany");
 }

 public void doFilter(ServletRequest zad,
 ServletResponse odp,
 FilterChain lancuch)
 throws IOException, ServletException {

 HttpServletRequest zadanie = (HttpServletRequest) zad;
 HttpServletResponse odpowiedz = (HttpServletResponse) odp;

 String dopuszczalne_kodowania = request.getHeader("Accept-Encoding");
 if (dopuszczalne_kodowania.indexOf("gzip") > -1) {
```

```
 OpakowanieKompresjiOdpowiedzi opakowanieOdp
 = new OpakowanieKompresjiOdpowiedzi(odpowiedz);
```



### Relax

Przystępując do egzaminu, nie musisz analizować tego kodu.

Dalsza część przykładu stanowi przykład konstrukcji filtra operującego na odpowiedzi. Zamieściliśmy go wyłącznie po to, abyś mógł zobaczyć przykład nieco bardziej praktyczny i nadający się do wykorzystania. Jeśli planujesz przystąpić do egzaminu, nie musisz ani studiować, ani próbować zrozumieć tego przykładu, zatem całą dalszą część rozdziału możesz potraktować jako opcjonalną.

Idea działania tego filtra polega na opakowaniu obiektu odpowiedzi za pomocą dekoratora. Nasz dekorator opakowuje strumień wyjściowy w ramach niestandardowego strumienia kompresującego.

Kompresja danych wyjściowych jest wykonywana wyłącznie pod warunkiem, że klient przestał nagłówek Accept-Encoding zawierający ciąg „gzip” (informując tym samym, że jest w stanie obsługiwać kompresję).

Czy klient obsługuje dane skompresowane w formacie GZIP?

Jeśli tak, to opakowujemy obiekt odpowiedzi, używając do tego celu opakowania kompresującego.

## Filtr kompresujący, ciąg dalszy

### Wskazówka testowa!

Aby przetestować działanie tego filtra, umieść ten wiersz w komentarzu — w przeglądarce powinieneś wówczas zobaczyć niezrozumiałą, skompresowaną zawartość odpowiedzi.

```
opakowanieOdp.setHeader("Content-Encoding", "gzip");
```

Deklarujemy, że zawartość odpowiedzi jest skompresowana w formacie GZIP.

```
lancuch.doFilter(zadanie, opakowanieOdp);
```

Wykonujemy dalszą część łańcucha filtrów.

```
GZIPOutputStream gzos = opakowanieOdp.getGZIPOutputStream();
gzos.finish();
```

```
ctx.log(cfg.getFilterName() + ": zakończono obsługę żądania.");
```

Kompresujący strumień GZIP należy „zakończyć”, co jednocześnie powoduje opróżnienie jego bufora i wystanie zawartości do oryginalnego strumienia wyjściowego.

Wszystkie dalsze operacje obsługuje już kontener.

```
} else {
 ctx.log(cfg.getFilterName() + ": nie przeprowadzono kodowania odpowiedzi.");
 lancuch.doFilter(zadanie, odpowiedz);
}
}
```

```
public void destroy() {
 // usunięcie składowych
 cfg = null;
 ctx = null;
}
```

```
}
```

## Własnymi ścieżkami

### Kompresja a protokół HTTP

Skąd serwer wie, że może przesłać do klienta skompresowaną zawartość? Skąd przeglądarka wie, że odpowiedź, którą odbiera zawiera skompresowaną treść? Okazuje się, że protokół HTTP przystosowano do obsługi kompresji. Poniżej krótko opisano najważniejsze elementy odpowiednich mechanizmów:

- Jeden z nagłówków przesyłanych przez przeglądarkę (a konkretnie nagłówek Accept-Encoding: gzip) informuje serwer o możliwościach przeglądarki w zakresie obsługi rozmaitych rodzajów treści.
- Jeśli serwer stwierdzi, że przeglądarka jest w stanie obsługiwać skompresowane dane, to wykona kompresję i doda do odpowiedzi stosowny nagłówek (a konkretnie — Content-Encoding: gzip).
- Kiedy przeglądarka odbierze odpowiedź, nagłówek Content-Encoding: gzip poinformuje ją, iż przed wyświetleniem danych należy je rozpakować.

Jak na razie wszystko idzie dobrze. Jak trudne może być napisanie takiego małego opakowania? (Słynne ostatnie słowa...)



# Kod opakowania kompresji odpowiedzi

Przedstawiliśmy już kod filtra kompresującego, teraz z kolei przyjrzymy się klasie opakowania używanego przez ten filtr. To jedno z najbardziej skomplikowanych zagadnień związanych z serwetami, a zatem nie wpadaj w panikę, jeśli nie zrozumiesz wszystkiego za pierwszym razem.

Poniższe opakowanie „dekoruje” oryginalny obiekt odpowiedzi, podmieniając oryginalny strumień wyjściowy strumieniem umożliwiającym kompresję.

```
package com.example.web;
```

```
// związane z serwetami
import javax.servlet.http.*;
import javax.servlet.*;
// związane z wejściem-wyjściem
import java.io.*;
import java.util.zip.GZIPOutputStream;
```

```
class OpakowanieKompresjiOdpowiedzi extends HttpServletResponseWrapper {
```

```
 private GZIPServletOutputStream strumienGzipOut = null;
```

Skompresowany strumień wyjściowy stosowany dla odpowiedzi serwletu.

```
 private PrintWriter pw = null;
```

Obiekt PrintWriter operujący na skompresowanym strumieniu wyjściowym.

```
 OpakowanieKompresjiOdpowiedzi(HttpServletResponse odp) {
 super(odp);
 }
```

Konstruktor `super()` wykonuje obowiązki dekoratora związane z koniecznością przechowania referencji do dekorowanego obiektu, czyli w tym przypadku obiektu `HttpServletResponse`.

```
 public void setContentLength(int dl) { }
```

Tę metodę możemy zignorować, ponieważ wyniki będą kompresowane.

```
 public GZIPOutputStream getGZIPOutputStream() {
 return this.strumienGzipOut.internalGzipOS;
 }
```

Ta metoda dekoratora, używana przez filtr, przekazuje filtrowi „uchwyt” do wyjściowego strumienia GZIP, tak by filtr mógł „zakończyć” kompresję i opróżnić bufor wyjściowy.

## Kod opakowania kompresującego, ciąg dalszy

```

private Object uzywanyStrumien = null;

public ServletOutputStream getOutputStream() throws IOException {
 if ((uzywanyStrumien != null) && (uzywanyStrumien != pw)) {
 throw new IllegalStateException();
 }

 if (strumienGzipOut == null) {
 strumienGzipOut
 = new GZIPServletOutputStream(getResponse.getOutputStream());
 uzywanyStrumien = strumienGzipOut;
 }
 return strumienGzipOut;
}

public PrintWriter getWriter() throws IOException {
 if ((uzywanyStrumien != null) && (uzywanyStrumien != strumienGzipOut)) {
 throw new IllegalStateException();
 }

 if (pw == null) {
 strumienGzipOut
 = new GZIPServletOutputStream(getResponse.getOutputStream());
 OutputStreamWriter osw
 = new OutputStreamWriter(strumienGzipOut,
 getResponse().getCharacterEncoding());

 pw = new PrintWriter(osw);
 uzywanyStrumien = pw;
 }
 return pw;
}

```

Metoda zapewnia dostęp do dekorowanego strumienia wyjściowego serwletu.

Dajemy serwletowi możliwość dostępu do strumienia wyjściowego wyłącznie w przypadku, gdy serwlet wcześniej nie pobrał obiektu `PrintWriter`.

Opakujemy oryginalny strumień wyjściowy serwletu wyjściowym strumieniem kompresującym.

Ta metoda zapewnia dostęp do udekorowanego obiektu `PrintWriter`.

Dajemy serwletowi możliwość pobrania obiektu `PrintWriter` wyłącznie w przypadku, gdy serwlet wcześniej nie pobrał wyjściowego strumienia kompresującego.

Aby utworzyć obiekt `PrintWriter`, w pierwszej kolejności musimy opakować strumień wyjściowy serwletu, a następnie opakować wyjściowy strumień kompresujący dwoma dodatkowymi strumieniami wyjściowymi — `OutputStreamWriter` (który konwertuje znaki na bajty) oraz obiektem `PrintWriter` pracującym ponad obiektem `OutputStreamWriter`.

# Opakowanie kompresujące, kod klasy pomocniczej

Klasa przedstawiona na tej stronie jest dekoratorem abstrakcyjnej klasy `ServletOutputStream` i odpowiada za przekazywanie wywołań związanych z zapisywaniem kompresowanej treści do faktycznego strumienia wyjściowego `GZIPOutputStream`.

W klasie `ServletOutputStream` istnieje tylko jedna metoda abstrakcyjna — `write(int)` — którą należy zaimplementować w prezentowanej klasie pomocniczej. To właśnie w niej dokonuje się cała magia związana z przekazywaniem wywołania.

```
class GZIPServletOutputStream extends ServletOutputStream {
```

```
 GZIPOutputStream wewnetrznyStrumienGZIP;
```

```
 /** Konstruktor dekoratora */
```

```
 GZIPServletOutputStream(ServletOutputStream sos) throws IOException {
 this.wewnetrznyStrumienGZIP = new GZIPOutputStream(sos);
 }
```

```
 public void write(int param) throws java.io.IOException {
 wewnetrznyStrumienGZIP.write(param);
 }
```

```
}
```

W tej składowej będziemy przechowywali referencję do standardowego strumienia `GZIPOutputStream`. Składowa ma charakter prywatny na poziomie pakietu, dzięki czemu jest dostępna dla kompresującego opakowania odpowiedzi.

Ta metoda stanowi właściwą implementację dekoratora, ponieważ przekazuje wywołanie metody `write()` do strumienia typu `GZIPOutputStream`, który opakowuje oryginalny strumień wyjściowy typu `ServletOutputStream` (który z kolei jest opakowaniem strumienia wyjściowego odpowiedzialnego za przesyłanie danych do klienta za pośrednictwem sieci TCP).



## Zaostrz ołówek

### ODPOWIEDZI

Dla każdego z przedstawionych żądań zapisz sekwencję, w jakiej będą wykonywane filtry. Przyjmij przy tym, że filtry od Filtr1 do Filtr5 zostały poprawnie zadeklarowane.

```
<filter-mapping>
 <filter-name>Filtr1</filter-name>
 <url-pattern>/Przepisy/*</url-pattern>
</filter-mapping>

<filter-mapping>
 <filter-name>Filtr2</filter-name>
 <servlet-name>/Przepisy/ListaChmielu.do</servlet-name>
</filter-mapping>

<filter-mapping>
 <filter-name>Filtr3</filter-name>
 <url-pattern>/Przepisy/Dodaj/*</url-pattern>
</filter-mapping>

<filter-mapping>
 <filter-name>Filtr4</filter-name>
 <servlet-name>/Przepisy/Modyfikuj/RdzPrzepisu.do</servlet-name>
</filter-mapping>

<filter-mapping>
 <filter-name>Filtr5</filter-name>
 <url-pattern>/*</url-pattern>
</filter-mapping>
```

Ścieżka żądania	Sekwencja filtrów:
/Przepisy/RaportChmielu.do	Filtry: 1, 5
/Przepisy/ListaChmielu.do	Filtry: 1, 5, 2
/Przepisy/Modyfikuj/RdzPrzepisu.do	Filtry: 1, 5, 4
/ListaChmielu.do	Filtry: 5
/Przepisy/Dodaj/DodajPrzepis.do	Filtry: 1, 3, 5



- 
- 1 Które z poniższych stwierdzeń dotyczących filtrów są prawdziwe? (Należy zaznaczyć wszystkie poprawne opcje).
- ☐ A. Filtry mogą operować wyłącznie na obiekcie odpowiedzi bądź na obiekcie żądania, ale nie na obu tych obiektach jednocześnie.
  - ☐ B. Metoda **destroy()** zawsze jest metodą zwrrotną kontenera.
  - ☐ C. Metoda **doFilter()** zawsze jest metodą zwrrotną kontenera.
  - ☐ D. Jedynie umieszczenie odpowiedniej deklaracji w deskrytorze wdrożenia powoduje, że filtr będzie używany.
  - ☐ E. Kolejny filtr w łańcuchu filtrów można wskazać albo w poprzedzającym go filtrze, albo w deskrytorze wdrożenia.
- 
- 2 Które z poniższych stwierdzeń dotyczących deklarowania filtrów w deskrytorze wdrożenia są prawdziwe. (Należy zaznaczyć wszystkie poprawne opcje).
- ☐ A. W przeciwieństwie do serwetów, filtry NIE MOGĄ deklarować parametrów inicjalizacji.
  - ☐ B. Kolejność poszczególnych filtrów w łańcuchu jest zawsze określana na podstawie kolejności deklaracji tych filtrów w deskrytorze wdrożenia.
  - ☐ C. Klasa dziedzicząca po klasie opakowania żądania lub odpowiedzi musi zostać zadeklarowana w deskrytorze wdrożenia.
  - ☐ D. Klasa dziedzicząca po klasie opakowania żądania lub odpowiedzi jest przykładem wzorca projektowego Interception Filter (filtr przechwytyjący).
  - ☐ E. Kolejność filtrów w łańcuchu zależy do tego, czy odwzorowania filtrów są deklarowane przy użyciu elementów **<url-pattern>** czy też **<servlet-name>**.

2 Przeanalizuj kod poniższej metody należącej do poprawnej implementacji filtra:

```
20. public void doFilter(ServletRequest zad,
21. ServletResponse odpowiedz,
22. FilterChain lancuch)
22. throws IOException, ServletException {
23. HttpServletRequest zadanie = (HttpServletRequest) zad;
24. HttpSession sesja = zadanie.getSession();
25. Object uzytkownik = sesja.getAttribute("uzytkownik");
26. if (uzytkownik != null) {
27. ZadanieUzytkownika zadU = new ZadanieUzytkownika(zadanie, uzytkownik);
28. lancuch.doFilter(zadU, odpowiedz);
29. } else {
30. RequestDispatcher rd = zadanie.getRequestDispatcher("/logowanie.jsp");
31. rd.forward(zadanie, odpowiedz);
32. }
33. }
```

po czym wybierz poprawną opcję.

- ☐ A. Wykonanie wiersza 31. zawsze spowoduje zgłoszenie wyjątku.
- ☐ B. Wiersz 28. zawiera błąd, gdyż pierwszym argumentem zawsze musi być **zadanie** (obiekt żądania).
- ☐ C. Wywołanie **lancuch.doFilter(zadanie,odpowiedz)** musi być umieszczone gdzieś w bloku **else**.
- ☐ D. Przedstawiona metoda nie jest poprawną implementacją metody **Filter.doFilter()**, ponieważ jej sygnatura jest nieprawidłowa.
- ☐ E. Żadna z powyższych odpowiedzi nie jest poprawna.

**4** Przeanalizuj przedstawiony poniżej fragment deskryptora wdrożenia:

```
11. <filter>
12. <filter-name>Moj Filtr</filter-name>
13. <filter-class>com.example.MojFiltr</filter-class>
14. </filter>
15. <filter-mapping>
16. <filter-name>Moj Filtr</filter-name>
17. <url-pattern>/moje</url-pattern>
18. </filter-mapping>
19. <servlet>
20. <servlet-name>Moj Servlet</servlet-name>
21. <servlet-class>com.example.MojServlet</servlet-class>
22. </servlet>
23. <servlet-mapping>
24. <servlet-name>Moj Servlet</servlet-name>
25. <url-pattern>/moje</url-pattern>
26. </servlet-mapping>
```

a następnie wskaż stwierdzenia, które są prawdziwe. (Należy zaznaczyć wszystkie poprawne opcje).

- ☐ A. Deskryptor nie jest prawidłowy, ponieważ ten sam wzorec URL, **/moje**, skojarzono zarówno z filtrem, jak i z serwletem.
- ☐ B. Deskryptor nie jest prawidłowy, ponieważ ani w nazwach serwletów, ani w nazwach filtrów nie mogą występować znaki spacji.
- ☐ C. Dla każdego żądania pasującego do wzorca **/moje** filtr **MojFiltr** zostanie wywołany po serwlecie **MojServlet**.
- ☐ D. Dla każdego żądania pasującego do wzorca **/moje** filtr **MojFiltr** zostanie wywołany przed serwletem **MojServlet**.
- ☐ E. Deskryptor nie jest poprawny, gdyż element **<filter>** musi zawierać element **<servlet-name>** określający, dla jakiego serwletu filtr zostanie zastosowany.

5 Które z poniższych stwierdzeń dotyczących filtrów są prawdziwe? (Należy zaznaczyć wszystkie poprawne opcje).

- ☐ A. Filtrów można używać do tworzenia opakowań żądań i odpowiedzi.
- ☐ B. Opakowań można używać do tworzenia filtrów operujących na żądaniach i odpowiedziach.
- ☐ C. W przeciwieństwie do serwletów, cały kod inicjalizujący filtrów należy umieszczać w ich konstruktorach, ponieważ filtry nie dysponują metodą **init()**.
- ☐ D. Filtry dysponują mechanizmem inicjalizacji składającym się z metody **init()**, która musi zostać wywołana, zanim filtr zostanie użyty do obsługi żądania.
- ☐ E. Metoda **doFilter()** filtra musi wywołać metodę **doFilter()** wejściowego obiektu **FilterChain**, aby zapewnić możliwość wykonania wszystkich filtrów w łańcuchu.
- ☐ F. W wywołaniu metody **doFilter()** wejściowego obiektu **FilterChain** trzeba przekazać te same obiekty **ServletRequest** oraz **ServletResponse**, które zostały przekazane w wywołaniu metody **doFilter()** filtra.
- ☐ G. Metoda **doFilter()** filtra może przerwać dalsze przetwarzanie żądania.

6 Które z poniższych stwierdzeń dotyczących klas opakowań serwletów są prawdziwe? (Należy zaznaczyć wszystkie poprawne opcje).

- ☐ A. Klasy te zapewniają jedyną możliwość opakowania obiektów typu **ServletResponse**.
- ☐ B. Można je wykorzystywać do „dekorowania” klas implementujących interfejs **Filter**.
- ☐ C. Można z nich korzystać nawet w przypadkach, gdy tworzona aplikacja NIE jest aplikacją internetową.
- ☐ D. Interfejs programowy udostępnia klasy opakowań dla klas **ServletRequest**, **ServletResponse** oraz **FilterChain**.
- ☐ E. Klasy opakowań implementują wzorec projektowy Intercepting Filter (filtr przechwytyjący).
- ☐ F. Tworząc klasy dziedziczące po klasie opakowania, konieczne jest przesłonięcie przynajmniej jednej metody zastosowanej klasy opakowania.



## BAR KAWOWY

### Examin próbny — odpowiedzi

- 1 Które z poniższych stwierdzeń dotyczących filtrów są prawdziwe?  
(Należy zaznaczyć wszystkie poprawne opcje).

(Specyfikacja serwletów,  
wersja 2.4, sekcja 6.).

- ☐ A. Filtry mogą operować wyłącznie na obiekcie odpowiedzi bądź na obiekcie żądania, ale nie na obu tych obiektach jednocześnie.
- ☒ B. Metoda **destroy()** zawsze jest metodą zwrrotną kontenera.
- ☐ C. Metoda **doFilter()** zawsze jest metodą zwrrotną kontenera.
- ☒ D. Jedynie umieszczenie odpowiedniej deklaracji w deskrytorze wdrożenia powoduje, że filtr będzie używany.
- ☐ E. Kolejny filtr w łańcuchu filtrów można wskazać albo w poprzedzającym go filtrze, albo w deskrytorze wdrożenia.

– Odpowiedź C jest błędna, ponieważ metoda `doFilter()` jest zarówno metodą zwrrotną, jak i wbudowaną, rozwijaną w miejscu wywołania (ang.: inline).

– Odpowiedź E jest błędna, kolejność wykonywania poszczególnych filtrów zawsze jest określana przez deskryptor wdrożenia.

- 2 Które z poniższych stwierdzeń dotyczących deklarowania filtrów w deskrytorze wdrożenia są prawdziwe. (Należy zaznaczyć wszystkie poprawne opcje).

(Specyfikacja serwletów,  
wersja 2.4, sekcja 6.).

- ☐ A. W przeciwieństwie do serwletów, filtry NIE MOGĄ deklarować parametrów inicjalizacji.
- ☐ B. Kolejność poszczególnych filtrów w łańcuchu jest zawsze określana na podstawie kolejności deklaracji tych filtrów w deskrytorze wdrożenia.
- ☐ C. Klasa dziedzicząca po klasie opakowania żądania lub odpowiedzi musi zostać zadeklarowana w deskrytorze wdrożenia.
- ☐ D. Klasa dziedzicząca po klasie opakowania żądania lub odpowiedzi jest przykładem wzorca projektowego Interception Filter (filtr przechwytyjący).
- ☒ E. Kolejność filtrów w łańcuchu zależy do tego, czy odwzorowania filtrów są deklarowane przy użyciu elementów `<url-pattern>` czy też `<servlet-name>`.

– Odpowiedź B jest błędna, gdyż skojarzenie określone przy użyciu elementu `<url-pattern>` zostanie uwzględnione przed skojarzeniem zdefiniowanym przy użyciu elementu `<servlet-name>`.

– Odpowiedź D jest błędna, gdyż opakowania są przykładami zastosowania wzorca Decorator (dekorator).

- 3 Przeanalizuj kod poniższej metody należącej do poprawnej implementacji filtra: (Specyfikacja serwletów, wersja 2.4, str. 49).

```

20. public void doFilter(ServletRequest zad,
21. ServletResponse odpowiedz,
22. FilterChain lancuch)
22. throws IOException, ServletException {
23. HttpServletRequest zadanie = (HttpServletRequest) zad;
24. HttpSession sesja = zadanie.getSession();
25. Object uzytkownik = sesja.getAttribute("uzytkownik");
26. if (uzytkownik != null) {
27. ZadanieUzytkownika zadU = new ZadanieUzytkownika(zadanie, uzytkownik);
28. lancuch.doFilter(zadU, odpowiedz);
29. } else {
30. RequestDispatcher rd = zadanie.getRequestDispatcher("/logowanie.jsp");
31. rd.forward(zadanie, odpowiedz);
32. }
33. }

```

po czym wybierz poprawną opcję.

- ☐ A. Wykonanie wiersza 31. zawsze spowoduje zgłoszenie wyjątku. *– Odpowiedź A jest błędna, gdyż filtr może przekazywać żądanie.*
- ☐ B. Wiersz 28. zawiera błąd, gdyż pierwszym argumentem zawsze musi być **zadanie** (obiekt żądania). *– Odpowiedź B jest błędna, gdyż filtr może „opakowywać” żądanie (należy zauważyć, że klasa ZadanieUzytkownika musi implementować interfejs ServletRequest).*
- ☐ C. Wywołanie **lancuch.doFilter(zadanie,odpowiedz)** musi być umieszczone gdzieś w bloku **else**. *– Odpowiedź C jest błędna, gdyż NIE ma obowiązku wywoływania metody lancuch.doFilter() wewnątrz metody doFilter() filtra.*
- ☐ D. Przedstawiona metoda nie jest poprawną implementacją metody **Filter.doFilter()**, ponieważ jej sygnatura jest nieprawidłowa. *– Odpowiedź D jest błędna, gdyż sygnatura metody jest poprawna.*
- ☒ E. Żadna z powyższych odpowiedzi nie jest poprawna.

**4** Przeanalizuj przedstawiony poniżej fragment deskryptora wdrożenia: (Specyfikacja serwletów, wersja 2.4, str. 53).

```

11. <filter>
12. <filter-name>Moj Filtr</filter-name>
13. <filter-class>com.example.MojFiltr</filter-class>
14. </filter>
15. <filter-mapping>
16. <filter-name>Moj Filtr</filter-name>
17. <url-pattern>/moje</url-pattern>
18. </filter-mapping>
19. <servlet>
20. <servlet-name>Moj Serwlet</servlet-name>
21. <servlet-class>com.example.MojSerwlet</servlet-class>
22. </servlet>
23. <servlet-mapping>
24. <servlet-name>Moj Serwlet</servlet-name>
25. <url-pattern>/moje</url-pattern>
26. </servlet-mapping>

```

a następnie wskaż stwierdzenia, które są prawdziwe. (Należy zaznaczyć wszystkie poprawne opcje).

- ☐ A. Deskryptor nie jest prawidłowy, ponieważ ten sam wzorec URL, **/moje**, – Odpowiedź A jest błędna, gdyż to poprawna składnia służąca do kojarzenia filtra z tym samym wzorcem URL, z którym jest już skojarzony serwlet.
- ☐ B. Deskryptor nie jest prawidłowy, ponieważ ani w nazwach serwletów, ani w nazwach filtrów nie mogą występować znaki spacji. – Odpowiedź B jest błędna, gdyż takie ograniczenie nie istnieje.
- ☐ C. Dla każdego żądania pasującego do wzorca **/moje** filtr **MojFiltr** zostanie wywołany po serwlecie **MojSerwlet**. – Odpowiedź C jest błędna, gdyż filtry są wykonywane przed serwletami, a nie po nich.
- ☒ D. Dla każdego żądania pasującego do wzorca **/moje** filtr **MojFiltr** zostanie wywołany przed serwletem **MojSerwlet**.
- ☐ E. Deskryptor nie jest poprawny, gdyż element **<filter>** musi zawierać element **<servlet-name>** określający, dla jakiego serwletu filtr zostanie zastosowany. – Odpowiedź E jest błędna, gdyż wewnątrz elementu **<filter-mapping>** musi się znaleźć albo element **<servlet-name>**, albo **<url-pattern>**.

5 Które z poniższych stwierdzeń dotyczących filtrów są prawdziwe? (Specyfikacja serwetów, wersja 2.4, str. 51).  
(Należy zaznaczyć wszystkie poprawne opcje).

- ☒ A. Filtrów można używać do tworzenia opakowań żądań i odpowiedzi.
- ☐ B. Opakowań można używać do tworzenia filtrów operujących na żądaniach i odpowiedziach. – Odpowiedź B jest błędna, gdyż sytuacja jest odwrotna.
- ☐ C. W przeciwieństwie do serwetów, cały kod inicjalizujący filtrów należy umieszczać w ich konstruktorach, ponieważ filtry nie dysponują metodą **init()**. – Odpowiedź C jest błędna, gdyż metoda **init()**, której można używać do inicjalizacji filtrów, istnieje.
- ☒ D. Filtry dysponują mechanizmem inicjalizacji składającym się z metody **init()**, która musi zostać wywołana, zanim filtr zostanie użyty do obsługi żądania.
- ☐ E. Metoda **doFilter()** filtra musi wywołać metodę **doFilter()** wejściowego obiektu **FilterChain**, aby zapewnić możliwość wykonania wszystkich filtrów w łańcuchu.
- ☐ F. W wywołaniu metody **doFilter()** wejściowego obiektu **FilterChain** trzeba przekazać te same obiekty **ServletRequest** oraz **ServletResponse**, które zostały przekazane w wywołaniu metody **doFilter()** filtra. – Odpowiedź E jest błędna, gdyż wywoływanie metody **doFilter()** nie jest konieczne, jeśli filtr chce zablokować dalsze przetwarzanie żądania.
- ☒ G. Metoda **doFilter()** filtra może przerwać dalsze przetwarzanie żądania. – Odpowiedź F jest błędna, gdyż filtr może zdecydować się na „opakowanie” obiektu żądania lub odpowiedzi i przekazania tych „opakowanych” obiektów.

6 Które z poniższych stwierdzeń dotyczących klas opakowań serwetów są prawdziwe? (API)  
(Należy zaznaczyć wszystkie poprawne opcje).

- ☐ A. Klasy te zapewniają jedyną możliwość opakowania obiektów typu **ServletResponse**. – Odpowiedź A jest błędna, gdyż można samodzielnie tworzyć klasy opakowań.
- ☐ B. Można je wykorzystywać do „dekorowania” klas implementujących interfejs **Filter**. – Odpowiedź B jest błędna, gdyż te klasy służą do „opakowywania” obiektów żądania i odpowiedzi.
- ☒ C. Można z nich korzystać nawet w przypadkach, gdy tworzona aplikacja NIE jest aplikacją internetową.
- ☐ D. Interfejs programowy udostępnia klasy opakowań dla klas **ServletRequest**, **ServletResponse** oraz **FilterChain**. – Odpowiedź D jest błędna, gdyż w API NIE jest dostępne opakowanie klasy **FilterChain**.
- ☐ E. Klasy opakowań implementują wzorec projektowy Intercepting Filter (filtr przechwytyjący). – Odpowiedź E jest błędna, ponieważ opakowania implementują wzorec projektowy Decorator.
- ☐ F. Tworząc klasy dziedziczące po klasie opakowania, konieczne jest przesłonięcie przynajmniej jednej metody zastosowanej klasy opakowania.



## 14. Wzorce i Struts

# Korporacyjne wzorce projektowe



**Ktoś to już wcześniej zrobił.** Jeśli właśnie zaczynasz tworzyć aplikacje internetowe w języku Java, to masz dużo szczęścia. Możesz wykorzystać wspólną mądrość dziesiątków tysięcy programistów, którzy już do dawna się tym zajmują i mają już firmową koszulkę. Wykorzystując wzorce projektowe, zarówno te związane z platformą J2EE, jak i *wszelkie* inne, możesz uprościć swój kod i swoje życie. W przypadku aplikacji internetowych najważniejszym wzorcem projektowym jest wzorec Model-Widok-Kontroler (ang. *Model-View-Controller*, w skrócie *MVC*). Właśnie wzorec MVC wykorzystano w bardzo popularnym szkielecie nazwanym Struts, który pomaga przy tworzeniu elastycznych i łatwych w utrzymaniu serwletów Kontrolera Frontowego (ang. *Front Controller*). Wykorzystanie pracy *innych* jesteś winien samemu sobie — dzięki temu będziesz mógł poświęcić więcej czasu na ważniejsze sprawy (takie jak jeżdżenie na nartach, gra w golfa, tańczenie salsy, gra w piłkę nożną, pokera, muzykowanie na akordeonie...).



### Wzorce J2EE

- 11.1.** Na podstawie opisu scenariusza składającego się z listy problemów wskaż wzorzec, który można zastosować w jego rozwiązaniu. Oto lista wzorców, które musisz znać: Intercepting Filter (filtr przechwytyjący), Model-View-Controller (Model-Widok-Kontroler), Front Controller (kontroler frontowy), Service Locator (lokalizator usług), Business Delegate (delegat biznesowy) oraz Transfer Object (obiekt transferu).
- 11.2.** Każdy z następujących wzorców: Intercepting Filter (filtr przechwytyjący), Model-View-Controller (Model-Widok-Kontroler), Service Locator (lokalizator usług), Business Delegate (delegat biznesowy) oraz Transfer Object (obiekt transferu) dopasuj do stwierdzeń opisujących potencjalne korzyści, które można dzięki nim osiągnąć.

### Uwagi wyjaśniające:

*Wszystkie z celów egzaminacyjnych zamieszczonych w lewej kolumnie zostały szczegółowo opisane w niniejszym rozdziale. Choć w zasadzie stwierdzenie to nie jest zgodne z prawdą — zostały opisane BARDZIEJ niż wyczerpująco. Pytania egzaminacyjne związane z wzorcami są najmniej podchwytliwe i trudne spośród wszystkich pytań pojawiających się na egzaminie. A zatem, czytając niniejszy rozdział, prawie możesz się odprężyć.*

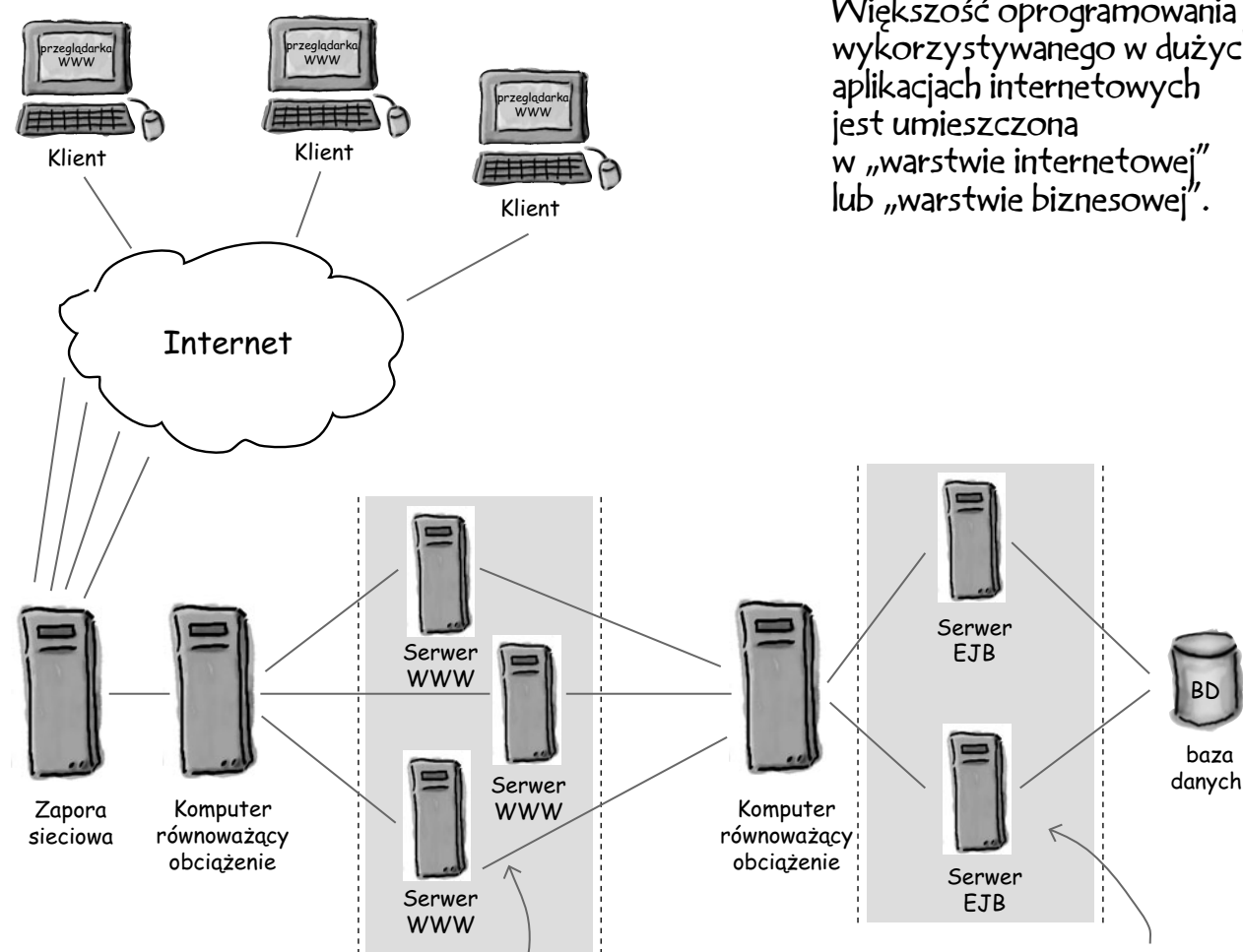
*Jeśli dysponujesz już jakimiś wiadomościami na temat podstawowych korporacyjnych wzorców projektowych, najprawdopodobniej już teraz mógłbyś z powodzeniem przystąpić do tej części egzaminu.*

*Mimo że zagadnienia związane ze stosowaniem frameworku Struts na pewno nie pojawiają się na egzaminie, w tym rozdziale zdecydowaliśmy się zamieścić podstawowe informacje na jego temat, ponieważ właśnie Struts jest obecnie najbardziej popularnym frameworkiem tworzenia aplikacji internetowych zgodnie z modelem MVC.*

## Sprzęt obsługujący witryny internetowe może być bardzo skomplikowany

W praktyce aplikacje internetowe mogą być bardzo skomplikowane. Popularna witryna internetowa może codziennie rejestrować setki tysięcy odwiedzin. Aby móc obsługiwać takie natężenie ruchu, najpopularniejsze witryny WWW wykorzystują skomplikowane architektury sprzętowe, w których oprogramowanie oraz dane są rozproszone na wielu różnych komputerach.

Jedna z najpopularniejszych architektur, z którą zapewne się już spotkałeś, polega na rozmieszczeniu komputerów w warstwach odpowiadających pełnionym przez nie funkcjom. Dodawanie kolejnych komputerów do tej samej warstwy nosi nazwę **skalowania poziomego** i jest uważane za jeden z najlepszych sposobów zwiększania przepustowości



Większość oprogramowania wykorzystywanego w dużych aplikacjach internetowych jest umieszczona w „warstwie internetowej” lub „warstwie biznesowej”.

„Warstwa internetowa” lub „warstwa prezentacji”. To właśnie w niej umieszczane są serwlety i strony JSP. Wraz ze wzrostem liczby odwiedzin witryny można do niej dodawać kolejne serwery, aby przystosować architekturę do większego natężenia ruchu.

„Warstwa biznesowa”. To właśnie ona zawiera całą logikę biznesową. Jeśli witryna musi obsługiwać coraz to większe natężenie ruchu, można do niej dodawać kolejne serwery.

# Oprogramowanie obsługujące aplikacje internetowe może być skomplikowane

Mogliśmy się już przekonać, że aplikacje internetowe bardzo często składają się z wielu różnych rodzajów komponentów programowych. Warstwa internetowa zazwyczaj zawiera strony HTML, JSP, serwlety, kontrolery, komponenty modelu, obrazy i tak dalej. Z kolei warstwa biznesowa może zawierać komponenty EJB, starsze aplikacje, rejestry używane podczas wszelkiego typu operacji wyszukiwania, a w większości przypadków także sterowniki baz danych oraz bazy danych.

sterowniki      modele      obrazy  
filtry      strony JSP      kontrolery  
serwlety      komponenty EJB  
widoki      JND      bazy danych



To jest doba  
Internetu, Złotko. A ten  
kod ma już kilka tygodni...  
Najwyższy czas dodać  
do niego jakieś nowe  
możliwości!

W jaki sposób mam  
zorganizować i zapewnić porządek  
tych wszystkich elementów aplikacji.  
W jaki sposób mogę zapewnić wysoką  
szybkość jej działania?



## Na szczęście mamy wzorce J2EE

Na szczęście bardzo *wiele* osób używało już kontenerów J2EE w celu rozwiązania problemów, które także i Ty najprawdopodobniej napotkasz. Osoby te odkryły powtarzalne wzorce charakterystyczne dla tych problemów i na tej podstawie opracowały rozwiązania wielokrotnego użytku. Te **wzorce projektowe** były stosowane, testowane i ulepszane przez innych programistów, dzięki czemu Ty nie będziesz musiał wymyślać ich raz jeszcze.

### Powszechne wymagania

Najważniejszym zadaniem aplikacji internetowej jest zapewnienie jej użytkownikom godnych zaufania, przydatnych i odpowiednich możliwości. Innymi słowy, program musi spełniać *wymagania funkcjonalne*, takie jak „wybierz rodzaj piwa” bądź „dodaj słód do mojego koszyka”. Kiedy się upewnisz, że system obsługuje niezbędne przypadki użycia, to prawdopodobnie będziesz musiał obsłużyć kolejną grupę wymagań — dotyczących tego, co się dzieje „za kulisami”, czyli, na przykład, wymagań *niefunkcjonalnych*.



### Zaostrz ołówek

#### Jakie są „możliwości”?

Proszę podać niektóre spośród niefunkcjonalnych wymagań systemu, nad którymi pracowałeś (lub nad którymi pracę możesz sobie wyobrazić). (Wskazówka: większość tych wymagań będzie zawierała słowa zakończone literami „...ość”, na przykład: „łatwość utrzymania”).

Programowe wzorce projektowe to „powtarzające się rozwiązania problemów, które często pojawiają się podczas tworzenia oprogramowania”.

## Różnego typu „ości”

Poniżej przedstawiliśmy trzy najczęściej spotykane wymagania niefunkcjonalne, z którymi zapewne się spotkasz:

### ① Wydajność

Jeśli Twoja witryna jest zbyt wolna, to (bez wątpienia) spowoduje to utratę użytkowników. W tym rozdziale pokażemy, jak wzorce projektowe mogą pomóc w skróceniu **czasu odpowiedzi** oraz w zwiększeniu liczby jednocześnie obsługiwanych użytkowników (czyli w zwiększeniu **przepustowości**). (Więcej informacji na ten temat podamy przy okazji prezentacji wzorca *Transfer Object*).

### ② Modularność

Aby różne elementy aplikacji mogły być jednocześnie wykonywane na różnych komputerach, tworzone oprogramowanie musi być modularne... co więcej: modularne w *odpowiedni sposób*.

### ③ Elastyczność, łatwość utrzymania i łatwość rozbudowy

**Elastyczność:** Musisz zmodyfikować swój system bez przechodzenia przez ten sam duży cykl produkcyjny. Być może musisz podmienić w aplikacji komponenty do obsługi „specjalnej oferty obowiązującej przez krótki czas” na potrzeby planowanej dużej promocji. Pewnie znalazłeś jakieś błędy w nowym komponencie i musisz zastąpić go tymczasowo jego wcześniejszą wersją. A zatem Twój system musi być elastyczny.

**Łatwość utrzymania:** Być może musisz zmienić producenta bazy danych i szybko zaktualizować system. Albo, gdy w aplikacji pojawiają się jakieś dziwne błędy, musisz jak najszybciej zlokalizować źródło tych problemów. A może administratorzy zdecydowali się na modyfikację firmowej usługi nazewnicznej i musisz się dostosować do tych zmian **natychmiast!** A zatem Twój system musi być łatwy w utrzymaniu.

**Łatwość rozbudowy (rozszerzalność):** Pracownicy działu marketingu mogą potrzebować nowych możliwości, aby zdobyć dużego i ważnego klienta. Albo użytkownicy witryny mogą żądać dodania nowych możliwości, które obsługują ich przeglądarki. Lepiej, żeby Twój system zapewniał łatwość rozbudowy!

Jeśli wzorce J2EE mogą mi pomóc rozwiązać wszystkie te problemy, to stanę się lokalnym, firmowym bohaterem. A to może mi zapewnić większy udział w akcjach spółki rozdawanych pracownikom. A kiedy nadejdzie kolejna bańka dotcomów, być może okaże się, że te akcje są coś warte.



## Wyjaśnienie terminologii...

Wszystkie wzorce J2EE w dużej mierze zależą od popularnych zasad projektowania oprogramowania, które najprawdopodobniej doskonale znasz. Na paru następnych stronach przedstawimy kilka terminów związanych z tymi zasadami projektowania. Terminy te mogą być w różny sposób i z odmiennej perspektywy opisywane przez rozmaite osoby lub przedstawiane w książkach, dlatego też zamieszczamy nasze definicje tych terminów, abyś wiedział, co mamy na myśli.

## Kodowanie według interfejsów

Jak sobie zapewne przypominasz, interfejs jest pewnym **kontraktem pomiędzy dwoma obiektami**. Kiedy klasa implementuje interfejs, to w rzeczywistości stwierdza: „Moje obiekty mogą się porozumiewać w twoim języku”. Kolejną ogromną zaletą interfejsów jest możliwość wykorzystania **polimorfizmu**. Ten sam interfejs może być implementowany przez wiele klas. Obiekt wywołujący nie musi zwracać uwagi na to, jaki konkretnie obiekt wywołuje, o ile tylko spełnia on ustalony kontrakt. Na przykład kontener może używać dowolnych obiektów implementujących interfejs `Servlet`.

## Wyodrębnianie problemów i spójność

Wszyscy wiemy, że specjalizacja możliwości komponentów programowych sprawia, iż stają się one łatwiejsze do stworzenia, utrzymania i wielokrotnego użycia. Naturalną konsekwencją wyodrębniania problemów jest zwiększenie poziomu **spójności**. Spójność oznacza, na ile skutecznie udało się tak zaprojektować daną klasę, by rozwiązywała lub realizowała jeden — *spójny* — problem lub zadanie.

## Ukrywanie złożoności

Ukrywanie złożoności często jest ściśle powiązane z rozdzielaniem problemów. Jeśli na przykład nasz system musi się komunikować z usługą wyszukiującą, najlepszym rozwiązaniem jest ukrycie całej złożoności tego problemu oraz jego rozwiązania w jednym komponencie i zapewnienie innym komponentom, które muszą używać tej usługi, możliwości korzystania z tego wyspecjalizowanego komponentu. Takie rozwiązanie przekłada się na większą prostotę wszystkich komponentów w jakikolwiek sposób związanych z naszą usługą wyszukiującą.

# Kolejne zasady projektowe...

## Niski stopień powiązań

Systemy obiektowe z natury rzeczy składają się ze wzajemnie komunikujących się obiektów. Kodowanie według interfejsów pozwala na zmniejszenie ilości informacji, jakie jedna klasa musi posiadać o drugiej, aby móc się z nią komunikować. Im mniej dwie klasy wiedzą na swój temat, tym mniejszy jest **stopień powiązań** pomiędzy nimi. Kiedy klasa A chce korzystać z metod klasy B, to bardzo często jest stosowane rozwiązanie polegające na utworzeniu interfejsu pomiędzy tymi klasami. Jeśli klasa B zaimplementuje odpowiedni interfejs, klasa A będzie mogła wywoływać jej metody właśnie za pośrednictwem tego interfejsu. Rozwiązanie takie jest bardzo wygodne i przydatne, gdyż później można zastąpić klasę B jej zmodyfikowaną wersją, bądź też całkowicie inną klasą, o ile tylko będzie ona spełniać warunki kontraktu określonego przez interfejs.

## Zdalne obiekty pośredniczące

Obecnie w przypadku zwiększającego się ruchu w aplikacji internetowej stosuje się rozwiązanie polegające nie na zwiększaniu możliwości jedyne, monolitycznego serwera, który tę aplikację obsługuje, lecz na dodawaniu nowych serwerów. Rozwiązanie to sprawia, że obiekty Javy wykonywane na różnych komputerach, na zupełnie niezależnych stertach, muszą się ze sobą komunikować.

Zdalne obiekty pośredniczące dodatkowo zwiększają potencjał interfejsów. Zdalne obiekty pośredniczące to w rzeczywistości obiekty lokalne względem „klienta”, które *udają*, że są obiektami zdalnymi. (Obiekt pośredniczący jest „zdalny” tylko względem emulowanego obiektu). Obiekty klienta komunikują się z pośrednikami, które z kolei obsługują wszystkie niuanse komunikacji sieciowej z faktycznym obiektem świadczącym usługi. Z punktu widzenia obiektu klienta komunikuje się on z obiektem lokalnym.

## Zwiększenie kontroli deklaratywnej

Deklaratywna kontrola nad aplikacją jest bardzo ważną cechą kontenerów J2EE. W większości przypadków kontrola ta jest realizowana za pośrednictwem deskryptora wdrożenia aplikacji. Modyfikacja tego deskryptora daje nam możliwość zmiany działania całej aplikacji bez konieczności wprowadzania jakichkolwiek modyfikacji w jej kodzie. Deskryptor wdrożenia jest plikiem XML, który może być modyfikowany przez osoby niemające żadnego doświadczenia programistycznego. W im większym stopniu wykorzystujesz możliwości deskryptora wdrożenia, pisząc aplikacje internetowe, tym bardziej abstrakcyjny i ogólny staje się Twój kod.

## Wzorce wspomagające zdalne komponenty modelu

Pisaliśmy, chociaż bardzo teoretycznie, w jaki sposób wzorce projektowe J2EE mogą pomóc w uproszczeniu bardzo złożonych aplikacji internetowych. Pisaliśmy także o zasadach projektowania obiektowego leżących u podstaw tych wzorców. Dysponując tymi informacjami, możemy zająć się konkretnymi i przedstawić kilka najprostszych spośród tych wzorców. Wszystkie trzy wzorce, które mamy zamiar przedstawić, mają to samo zadanie — ułatwienie zarządzania zdalnymi komponentami *modelu*.

### Bajeczka: Nasza „aplikacja piwna” się rozrasta

Dawno, dawno temu była sobie mała firma internetowa, która zajmowała się prowadzeniem witryny prezentującej przepisy dotyczące samodzielnego warzenia piwa, porady z tym związane oraz pozwalając na zamawianie i kupowanie niezbędnych składników. Ponieważ firma była mała (choć miała wielkie plany), dysponowała tylko jednym serwerem produkcyjnym obsługującym aplikację. Niemniej jednak w firmie istniały dwa niezależne zespoły programistów zajmujących się rozwojem aplikacji. Pierwszy z nich — zespół „projektantów stron” — koncentrował się na komponentach *widoku*, natomiast drugi — zespół „biznesowy” — koncentrował swoją uwagę na komponentach *kontrolera* (tym zajmowała się Rachel) oraz komponentach *modelu* (te należały do obowiązków Kima).

Ten facet zajmuje się wyglądem aplikacji...  
Interesują nas jedynie arkusze stylów. Nie z nudzaj nas sprawami związanymi z warstwą biznesową — my tworzymy sztukę.



Projektanci stron/aktorzy/kelnerzy

Zaczyna mnie to męczyć...



Serwer

Wydatność zaczyna być naprawdę poważnym problemem. Obecnie nie mamy dużego budżetu na sprzęt, jednak wiem, że musimy być przygotowani na moment, w którym trzeba będzie rozdzielić różne elementy aplikacji.

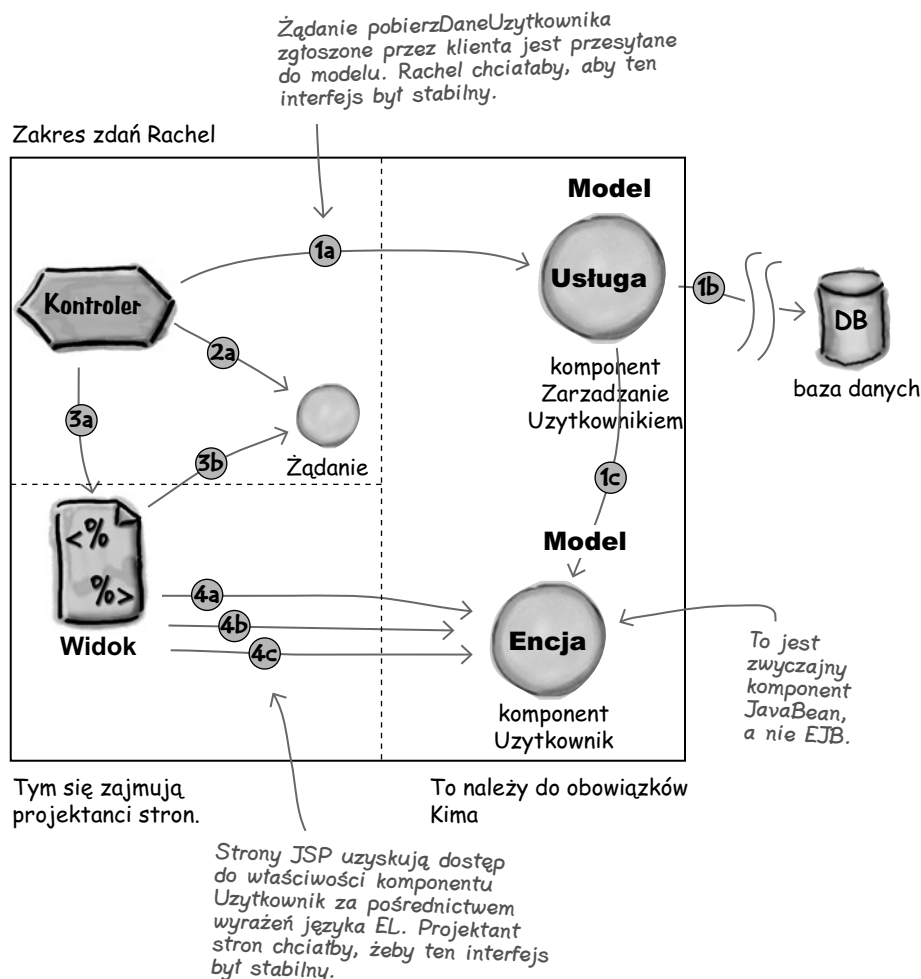


Rachel i Kim — zespół biznesowy

# Jak zespół biznesowy wspomaga projektantów w sytuacji, gdy wszystkie komponenty modelu MVC działają na jednej wirtualnej maszynie Javy?

O ile tylko członkowie zespołu biznesowego zapewnią spójność interfejsów swoich komponentów modelu, wszyscy będą szczęśliwi. Dwoma kluczowymi punktami ich interfejsów jest pierwszy kontakt *kontrolera* z komponentem *modelu* (kroki 1. i 2. na poniższym rysunku) oraz interakcja pomiędzy *widokiem* JSP i używanym przez niego komponentem (kroki 3. i 4.).

### Pobieranie danych o użytkowniku na potrzeby odpowiedzi...



- 1 Po otrzymaniu żądania dotyczącego informacji o użytkowniku, **kontroler** odwołuje się do komponentu **ZarządzanieUżytkownikiem** (**modelu**). Komponent ten wykorzystuje JDBC w celu pobrania informacji z bazy danych, a następnie, na ich podstawie, tworzy komponent **Użytkownik** (który nie jest komponentem EJB, lecz zwyczajnym komponentem JavaBean).
- 2 Kontroler dodaje referencję do komponentu **Użytkownik** do obiektu żądania jako jeden z jego atrybutów.
- 3 Kontroler przekazuje sterowanie do **widoku** JSP. Strona JSP pobiera referencję do komponentu **Użytkownik** z obiektu żądania.
- 4 Widok wykorzystuje język EL do uzyskiwania właściwości komponentu **Użytkownik** k niezbędnych do obsługi oryginalnego żądania.

## W jaki sposób będą obsługiwane obiekty zdalne?

Sytuacja wygląda w miarę prosto, kiedy wszystkie komponenty aplikacji internetowej (model, widok oraz kontroler) znajdują się na jednym komputerze i są wykonywane przez tę samą wirtualną maszynę Javy. W tym przypadku wszystkie operacje sprowadzają się do zwyczajnych operacji Javy — pobrania referencji, wywołania metody. Jednak *teraz* Kim i Rachel muszą się dowiedzieć, co zrobić w sytuacji, gdy ich komponenty modelu będą *zdalne* — czyli oddzielone od aplikacji internetowej.

### JNDI oraz RMI, krótka prezentacja

Java oraz J2EE udostępniają mechanizmy pozwalające na rozwiązanie dwóch najczęściej spotykanych problemów pojawiających się w przypadku, gdy obiekty muszą się ze sobą komunikować przez sieć. Chodzi mianowicie o **lokalizację** obiektów zdalnych oraz obsługę całej sieciowej **komunikacji** niskiego poziomu pomiędzy obiektami. (Innymi słowy, chodzi o to, jak *odszukać* obiekt zdalny oraz jak *wywoływać* jego metody).

#### JNDI w zarysie

*JNDI* to skrót angielskich słów: *Java Naming and Directory Interface* (interfejs nazewnictwa i katalogowy Javy). *JNDI* to interfejs programowy (API) udostępniający usługi nazewnictwa i katalogowe. *JNDI* zapewnia scentralizowane miejsce, w którym komputery połączone siecią mogą poszukiwać „danych”. Jeśli dysponujesz obiektami, do których inne programy działające w sieci muszą się odwoływać lub których muszą używać, to możesz zarejestrować je w *JNDI*. Jeśli jakiś inny program będzie chciał *użyć* takiego obiektu, odnajdzie go właśnie z wykorzystaniem interfejsu *JNDI*.

*JNDI* ułatwia także zmianę położenia poszczególnych komponentów w sieci. Po zmianie położenia komponentu wystarczy przekazać interfejsowi *JNDI* informacje o nowym położeniu przeniesionego komponentu. Dzięki temu wszelkie inne komponenty muszą jedynie wiedzieć, jak odnaleźć usługi *JNDI*, nie muszą natomiast orientować się, gdzie są przechowywane obiekty *zarejestrowane* w *JNDI*.

#### RMI w zarysie

*RMI* to skrót angielskich słów *Remote Methode Invocation* (wywoływanie metod zdalnych). Jest to mechanizm, który w ogromnym stopniu ułatwia proces komunikowania się obiektów przez sieć. Odwróć stronę, a znajdziesz krótkie przypomnienie tego, o czym jest mowa i jak działa *RMI*... zamieściliśmy je na wypadek, gdybyś musiał odświeżyć pamięć. Dlaczego w ogóle wspominamy tutaj o *RMI*? Ponieważ właśnie technologia *RMI* pomoże nam w zrozumieniu i docenieniu dwóch wzorców projektowych prezentowanych w tym rozdziale.

A zatem musimy przenieść część naszych komponentów modelu z komputera, na którym działa serwer WWW, i розміścić je w serwerach warstwy biznesowej. Pewnie zdajesz sobie sprawę, że ta operacja nie będzie wykonywana tylko jeden jedyny raz...



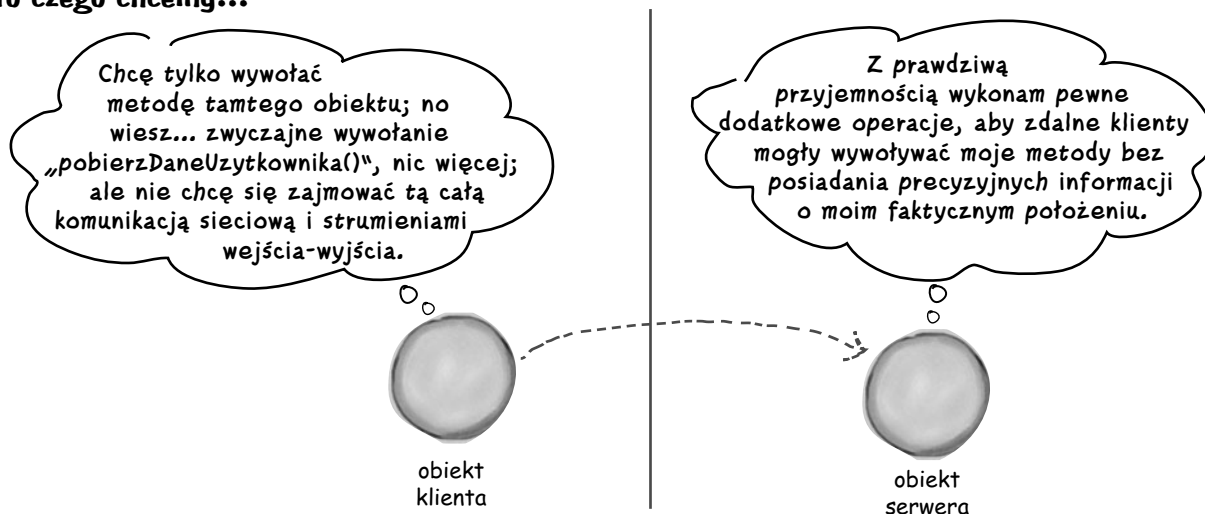
Dokładnie! Na dodatek, mogę się założyć, że w końcu będziemy musieli przenieść bardzo wiele obiektów. Lepiej, żeby nasz projekt komunikacji sieciowej był możliwie jak najprostszy.



## RMI ułatwia nam życie

Chcesz, aby Twoje obiekty komunikowały się ze sobą przez sieć. Innymi słowy, chcesz, aby obiekty wykonywane przez jedną wirtualną maszynę Javy wywoływały metody obiektów **zdalnych** (czyli wykonywanych przez *inną* wirtualną maszynę Javy), a jednocześnie chcesz *udawać*, że wywoływane są metody obiektów lokalnych. Właśnie takie możliwości daje Ci RMI — pozwala udawać (co prawda nie do końca), że wykonujesz zwyczajne, lokalne wywołania metod.

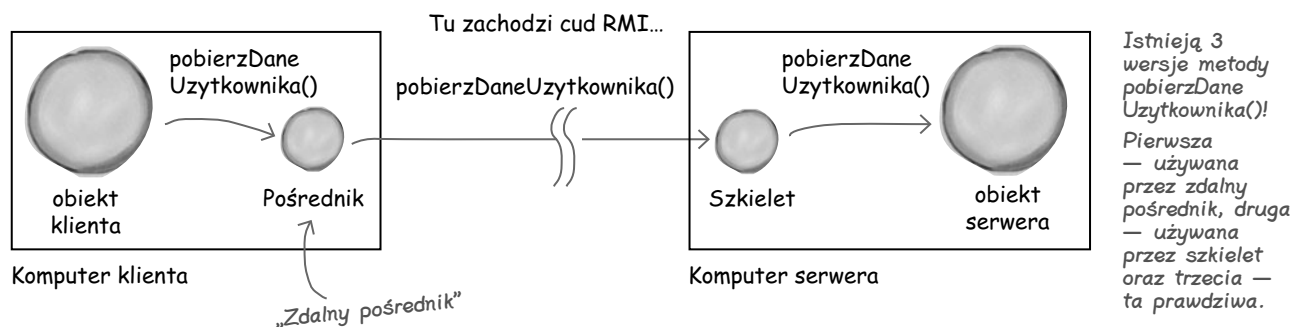
### Oto czego chcemy...



### Jak działa RMI?

Załóżmy, że zespół biznesowy wykonał swoje zadania i teraz chcesz udostępnić obiekty zdalnym klientom. Korzystając z RMI, utworzysz **pośredniki** i **zarejestrujesz** swój obiekt w pewnym rejestrze. Każdy klient, który będzie chciał wywoływać metody Twojego obiektu, będzie musiał odszukać go w rejestrze i pobrać kopię obiektu zdalnego *pośrednika*. Właśnie obiekt zdalnego **pośrednika** obsługuje wszystkie operacje związane z komunikacją — a więc gniazda, strumienie wejścia-wyjścia, TCP/IP, serializację i deserializację argumentów metod oraz zwracanych przez nie wartości, obsługę wyjątków i tak dalej.

(A swoją drogą... podobny obiekt znajduje się także po stronie serwera (nosi on nazwę „szkieletu” i wykonuje te same zadania co pośrednik tylko po stronie serwera).



## Nieco dokładniejsze informacje o RMI

Bez zbędnego przedstawiania wyczerpujących informacji o RMI\* przyjrzymy się kilku dodatkowym zagadnieniom związanym z tą technologią, tak by się upewnić, że „nadajemy na tych samych falach”. W szczególności przyjrzymy się aspektom wykorzystania RMI po stronie klienta oraz serwera.

### Cztery etapy stosowania RMI po stronie serwera

(Ogólny opis czynności niezbędnych do stworzenia zdalnej usługi modelu działającej na serwerze).

- ❶ Stwórz **zdalny interfejs**. To właśnie w nim będą umieszczone sygnatury metod takich jak **pobierzDaneUzytkownika()**. Interfejs ten będzie implementowany zarówno przez **obiekt pośrednika**, jak i faktyczną **usługę** modelu (czyli zdalny obiekt).
- ❷ Stwórz **zdalną implementację**, czyli faktyczny obiekt modelu, który będzie wykonywany na serwerze. Na tym etapie będziesz musiał stworzyć kod rejestrujący model w znanej Ci usłudze katalogowej, takiej jak JNDI lub RMI.
- ❸ Wygeneruj klasę pośrednika oraz (ewentualnie) szkielet. RMI udostępnia specjalny kompilator, o nazwie **rmic**, służący właśnie do tworzenia pośredników.
- ❹ Uruchom usługę modelu (która zarejestruje się w katalogu i będzie oczekiwać na wywołania przesyłane przez zdalnych klientów).

### Po stronie klienta, z wykorzystaniem RMI oraz bez niego

Porównajmy teraz pseudokod klienta, który wykorzystuje RMI, oraz klienta, który z RMI NIE korzysta.

#### Klient, który nie korzysta z RMI

```
public void obslugaKlienta() {
 try {
 // utworzenie obiektu Socket
 // pobranie strumienia OutputStream
 // połączenie go ze strumieniem ObjectOutputStream
 // przesłanie kodu operacji i argumentów
 // opróżnienie strumienia wyjściowego
 // pobranie strumienia InputStream
 // połączenie go ze strumieniem ObjectInputStream
 // odczytanie wartości wynikowej i (lub)
 // obsługa wyjątków
 // zakończenie operacji
 } // przechwycenie i obsługa zdalnych wyjątków
}
```

#### Klient używający RMI

```
public void obslugaKlienta() {
 try {
 // odszukanie zdalnego obiektu (pośrednika)

 // wywołanie metody zdalnego obiektu
 } // przechwycenie i obsługa zdalnych wyjątków
}
```

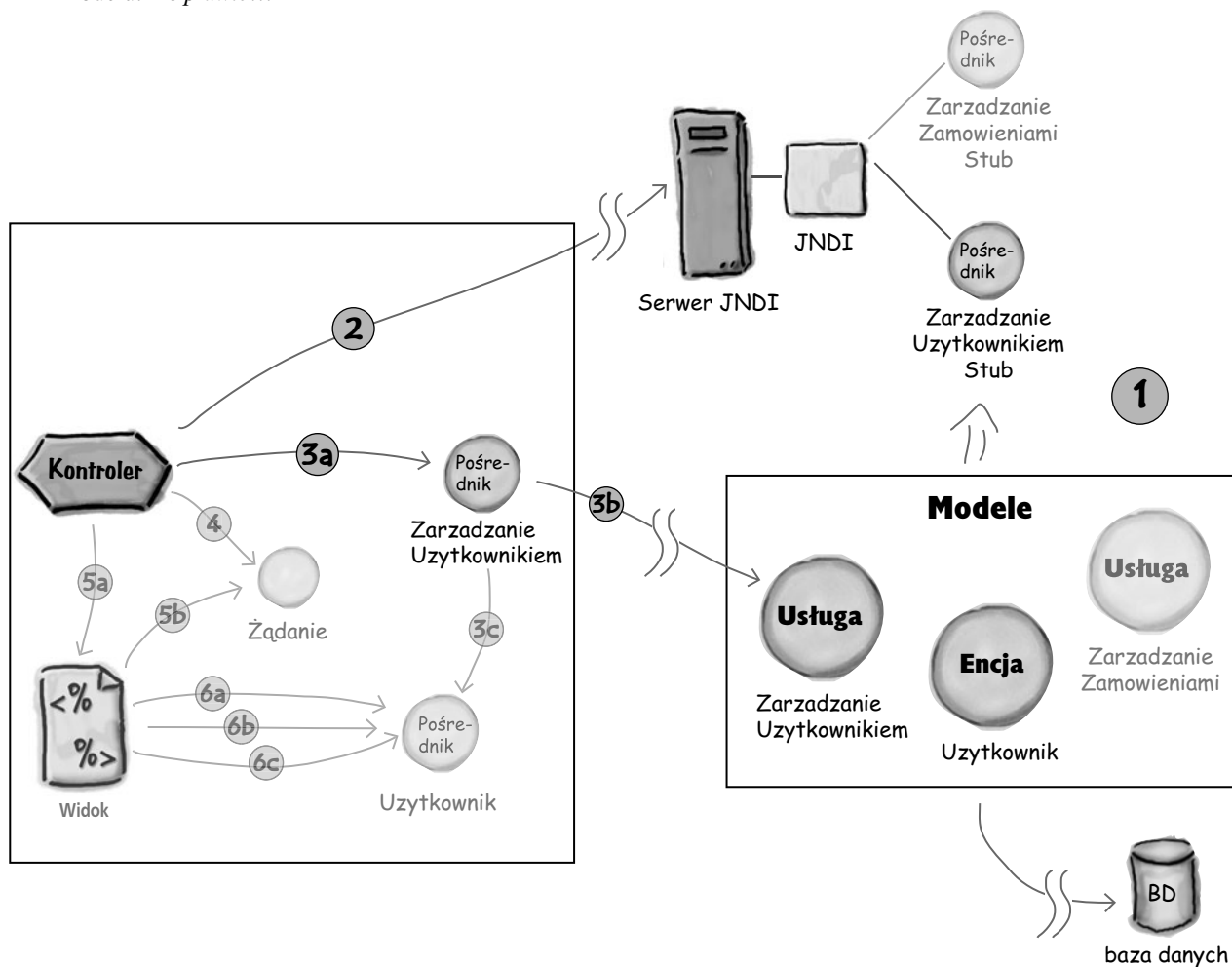
\* Jeśli jeszcze nie znasz RMI, to wybierz się do najbliższej księgarni, weź (ale nie *kupuj*) egzemplarz książki *Head First Java* i po prostu przeczytaj fragmenty poświęcone RMI. Następnie odłóż książkę na półkę, tak by okładka była dobrze widoczna i zasłaniała inne książki o Javie, które mogłyby Cię zainteresować. Upewnij się, że okładka nie jest zakurzona i uważaj, by nie pochłapać jej kawą.

### Dodawanie RMI i JNDI do kontrolera

Skoncentrujmy się na tym, co można zrobić, aby w jak największym stopniu uprościć życie Rachel. Innymi słowy, zajmijmy się wpływem, jaki dodanie JNDI i RMI wywiera na kontroler.

#### Trzy etapy stosowania zdalnych obiektów

- 1 Kim, zajmujący się komponentami modelu, *rejestruje* je w usłudze *JNDI*.
- 2 Kiedy kontroler Rachel odbiera żądanie, jego kod wykonuje operację *wyszukiwania* JNDI w celu odnalezienia pośrednika zdalnej usługi Kima.
- 3 Kontroler wykonuje wywołania metod biznesowych używając w tym celu pośrednika, przy czym realizuje je w taki sposób, jak gdyby pośrednik był rzeczywistym obiektem modelu. No *prawie*...



Hmm... Jak widać, wywołania metod są prawie takie same jak w sytuacji, gdy komponenty modelu były dostępne lokalnie, niemniej jednak i tak muszę zmodyfikować kod kontrolera i umieścić w nim wszystkie czynności związane z wyszukiwaniem JNDI. Miałam nadzieję na znalezienie rozwiązania, które pozwoli mi używać tego samego kontrolera niezależnie od tego, czy model jest dostępny lokalnie czy zdalnie.



Zaostrz ołówek

## W jaki sposób można ulepszyć to rozwiązanie?

- 1 - Jakich problemów przysparza to rozwiązanie (wymień przynajmniej dwa)?
- 2 - W jaki sposób można zmienić to rozwiązanie w celu wyeliminowania podanych wcześniej problemów?

**Problemy:**

**Rozwiązanie:**

# A może by zastosować jakiś obiekt „pośredniczący”?

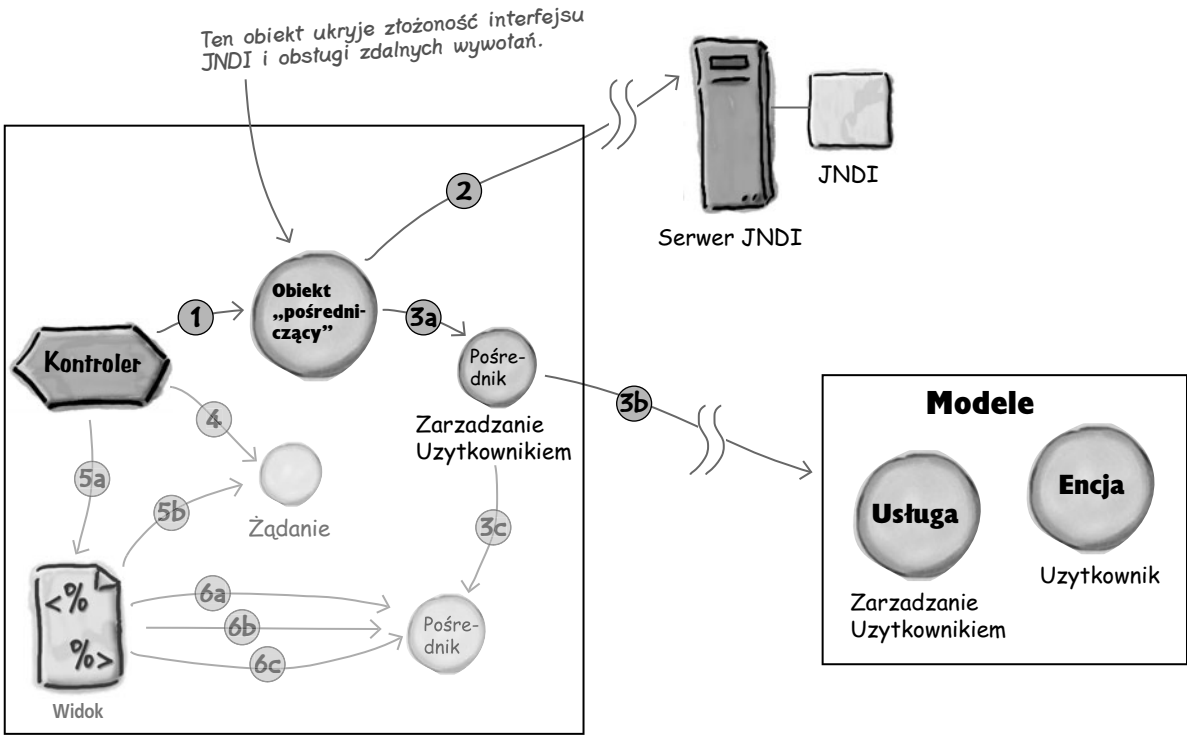
Popularnym rozwiązaniem problemów, nad którymi miałeś się zastanowić na poprzedniej stronie, jest zastosowanie nowego obiektu — pojedynczego obiektu „pośredniczącego”, z którego mógłby korzystać kontroler i który wyeliminowałby konieczność umieszczania w kontrolerze kodu niezbędnego do bezpośredniej obsługi *zdalnego* komponentu modelu.

## Problem 1. Ukrycie skomplikowanego wyszukiwania JNDI

Jeśli kontroler Rachel pozwoli, by to obiekt „pośredniczący” obsługiwał wyszukiwanie JNDI, kod samego kontrolera będzie można znacznie uprościć, eliminując mechanizmy odpowiedzialne za to, gdzie (i jak) odszukać komponent modelu.

## Problem 2. Ukrycie złożoności związanej ze stosowaniem komponentów zdalnych

Jeśli obiekt „pośredniczący” będzie w stanie współpracować z pośrednikiem, to kontroler Rachel zostanie całkowicie ochroniony i odizolowany od wszelkich zagadnień związanych z używaniem komponentów zdalnych, w tym także z koniecznością obsługi zdalnych *wyjątków*.



## Ten obiekt „pośredniczący” jest delegatem biznesowym

Przeanalizujmy pseudokod typowego delegata biznesowego (ang. *Business Delegate*) oraz sposób wdrażania tych delegatów na serwerze.

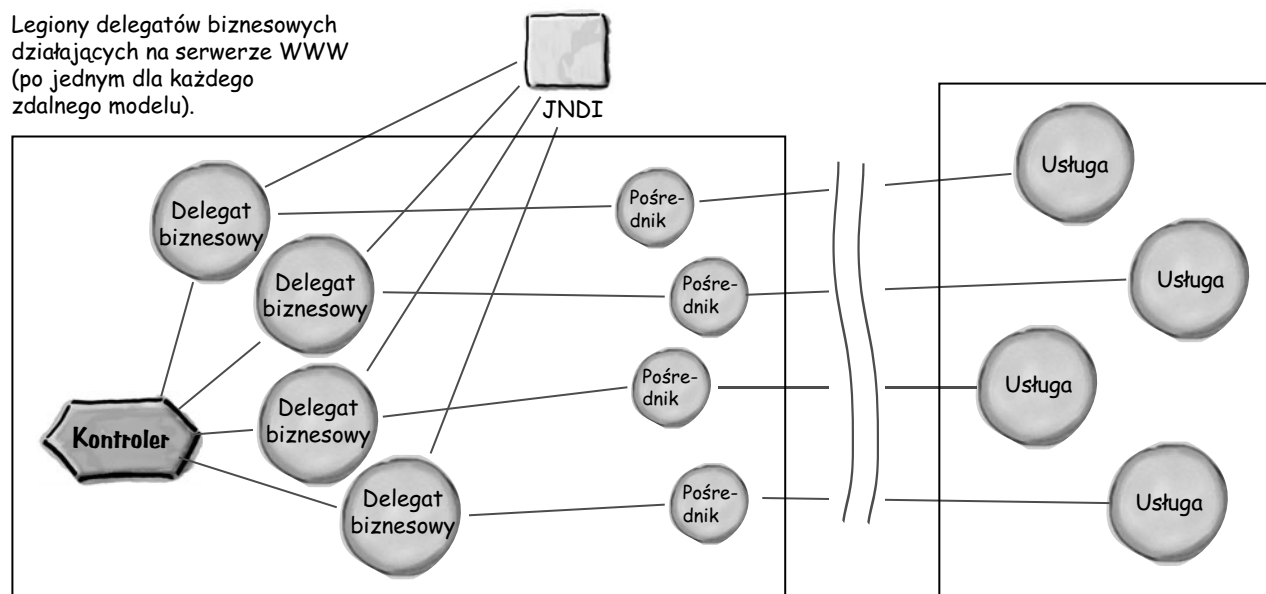
Warto zwrócić uwagę na to, że w warstwie internetowej zazwyczaj będzie używanych bardzo wiele takich delegatów biznesowych.

### Pseudokod delegata biznesowego

```
// pobranie żądania i wykonanie wyszukiwania JNDI
// przekazanie sterowania do pośrednika

// wywołanie metody biznesowej
// obsługa wszelkich zdalnych wyjątków
// przesłanie wartości wynikowej do kontrolera
```

Legiony delegatów biznesowych działających na serwerze WWW (po jednym dla każdego zdalnego modelu).



### Zaostrz ołówek



Uwaga! Uwaga! **Wykryto powtarzający się kod!**

(Wskaż miejsca, w których występuje powtarzający się kod, i zaproponuj rozwiązanie tego problemu).

## Uproszczenie używanych delegatów biznesowych poprzez zastosowanie lokalizatora usługi

Jeśli delegaty biznesowe nie będą współpracować z lokalizatorem usługi (ang. *Service Locator*), to konieczne będzie wykorzystanie w nich powtarzającego się kodu obsługującego wyszukiwanie JNDI.

Aby zaimplementować lokalizator usługi, ze wszystkich delegatów biznesowych należy usunąć logikę związaną z obsługą wyszukiwania JNDI i umieścić ją w *jednym* obiekcie lokalizatora.

Zazwyczaj w aplikacjach J2EE będzie istnieć wiele komponentów używających tych samych usług JNDI. Choć złożone aplikacje mogą używać kilku różnych rejestrów, takich jak JNDI oraz UDDI (przechowujących informacje o punktach końcowych usług internetowych), to jednak konkretny *komponent* będzie zazwyczaj korzystał tylko z *jednego* rejestru. Ogólnie rzecz biorąc, konkretny lokalizator usługi będzie obsługiwał tylko jeden określony rejestr.

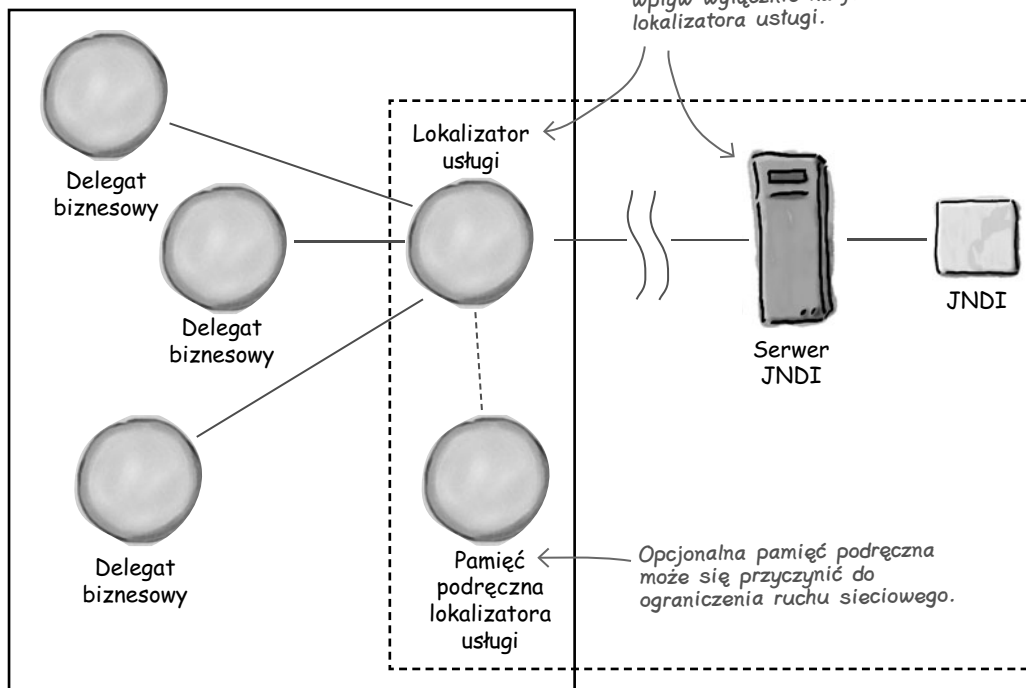
Jeśli delegaty **biznesowe** będą obsługiwać wyłącznie metody **biznesowe** i nie będą musieć zajmować się *także* zagadnieniami związanymi z przeszukiwaniem rejestru, to wpłynie to na zwiększenie stopnia ich spójności.

### Pseudokod lokalizatora usługi

```
// uzyskanie obiektu InitialContext
// wykonanie zdalnego przeszukania
// obsługa komunikacji sieciowej
// opcjonalne zachowanie referencji
// w pamięci podręcznej
```

Zapewniamy zwiększenie stopnia spójności wszystkich tych delegatów biznesowych.

Pobranie pośrednika jest obecnie obsługiwane przez lokalizator usługi. Delegaty mogą zatem skoncentrować się wyłącznie na stosowaniu metod biznesowych udostępnianych przez pośredników.



Web Server

## Nie ma niemądrych pytań

**P.** W dotychczasowych rozważaniach przyjmowaliśmy, że korzystamy z RMI. A co, jeśli moja firma używa technologii CORBA?

**O.** Wszystkie wzorce opisywane w tym rozdziale można, w większym lub mniejszym stopniu, zaimplementować niezależnie od technologii J2EE. W przypadku wykorzystania technologii J2EE ich implementacja będzie łatwiejsza, lecz można ich także używać w sytuacjach, gdy technologie te nie są stosowane.

**P.** Czy to samo dotyczy możliwości zastosowania innej usługi niż JNDI?

**O.** Cóż... Oprócz JNDI istnieją także i inne rejestry stosowane w programach pisanych w Javie — na przykład RMI lub Jini. Jednak prawdopodobnie właśnie JNDI sprawdza się najlepiej w obszarze aplikacji internetowych, ponieważ jest stosunkowo prosty w obsłudze i oferuje spore możliwości. (Choć autorzy niniejszej książki *osobiście* bardzo by chcieli, aby technologia Jini zajęła właściwe jej miejsce w świecie przetwarzania rozproszonego). Można się także spotkać z innymi rejestrami, które nie zostały stworzone w Javie, takimi jak UDDI. Niezależnie od zastosowanego rejestru same *wzorce* się nie zmieniają, choć oczywiście zmienią się kod ich implementacji.

**P.** Można odnieść wrażenie, że wzorce w nieskończoność dodają do architektury rozwiązania nowe warstwy obiektów. Skąd tak duża popularność tego podejścia?

**O.** Masz rację, iż dodawanie nowych warstw obiektów jest częstym zjawiskiem w wielu wzorcach projektowych. Zakładając, że opracowany projekt jest dobry, pomyśl o korzyściach projektowych, jakie niesie ze sobą rozwiązanie proponowane przez wzorec.

**P.** No dobrze... Pierwszą, która mi przychodzi do głowy, jest zwiększenie stopnia spójności...

**O.** Dobrze! Wzorce Business Delegate i Service Locator stosowane łącznie poprawiają **spójność** obiektów, z którymi współpracują. Ich kolejną zaletą jest **ukrywanie operacji sieciowych**. Dodanie kolejnej warstwy często sprawia, że istniejące obiekty nie muszą być już zależne od operacji sieciowych. To oczywiście ściśle się wiąże ze **spójnością** i **rozdzieleniem zadań**.

**P.** Rozdzielenie zadań daje mi...?

**O.** Przeanalizujmy przykład lokalizatora usługi. Jeśli zmieni się adres sieciowy używanego rejestru bądź jego interfejs, to znacznie łatwiej będzie zmodyfikować jeden lokalizator usługi niż całą armadę delegatów biznesowych. Ogólnie rzecz biorąc, rozdzielenie zadań zwiększa elastyczność i łatwość utrzymania kodu.

**P.** W przykładach przedstawionych do tej pory używane wcześniej **POJO** działające lokalnie zmienialiście na obiekty działające zdalnie. Jednak czy nie bardziej prawdopodobna jest sytuacja, w której będę musiał zintegrować swoją aplikacją internetową z już istniejącymi komponentami EJB?

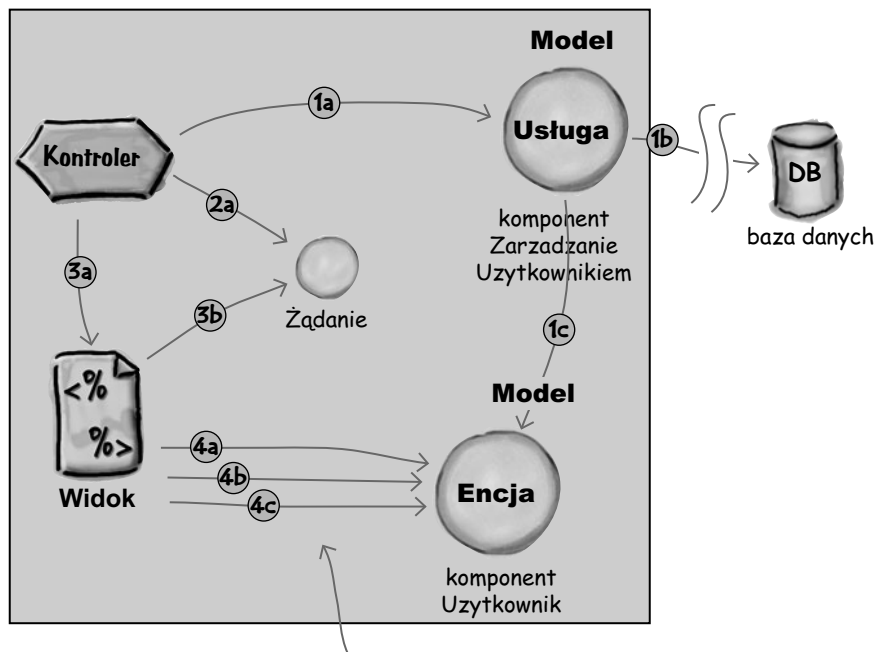
**O.** Zakładamy, że używając skrótu POJO, masz na myśli zwyczajne, tradycyjne obiekty Javy (ang: *plain, old Java objects*). Owszem — jest całkiem prawdopodobne, że będziesz musiał integrować ze swoją aplikacją komponenty EJB, jednak ten fakt jest kolejnym powodem przemawiającym za stosowaniem dwóch opisywanych wzorców projektowych... Twój kontroler (oraz widok) nigdy nie powinien wiedzieć, czy używany model jest lokalnym komponentem JavaBean, zdalnym obiektem Javy czy też komponentem (EJB). W razie braku delegata biznesowego i lokalizatora usługi różnice pomiędzy wymienionymi rodzajami obiektów zyskują duże znaczenie — komponenty EJB oraz zwyczajne obiekty Javy działające zdalnie odszukuje się przy wykorzystaniu zupełnie innych metod. Dzięki opisanym wzorcom można z powodzeniem izolować wszelkie zagadnienia związane z tym, gdzie i jak przebiega proces poszukiwania modelu. Oznacza to, że w razie wprowadzenia przez zespół biznesowy jakichś modyfikacji w komponentach warstwy biznesowej lub zmiany ich położenia nie będzie konieczne modyfikowanie kodu kontrolera. Niezbędna będzie jedynie zmiana lokalizatora usługi oraz (być może) delegata biznesowego.

## Ochrona stron JSP przed złożonością korzystania z komponentów zdalnych

Dzięki zastosowaniu wzorców projektowych Business Delegate (delegata biznesowego) i Service Locator (lokalizatora usług) udało nam się uchronić kontroler tworzony przez Rachel przed problemami związanymi ze stosowaniem zdalnych komponentów modelu. Sprawdźmy, czy jesteśmy w stanie w podobny sposób ochronić strony JSP.

### Krótkie przypomnienie wcześniejszego, lokalnego rozwiązania — kod JSP wykorzystywał język EL do uzyskiwania informacji z lokalnych komponentów modelu.

Powinieneś już znać poniższy schemat, gdyż był on przedstawiony we wcześniejszej części rozdziału. Strony JSP pobierają referencję do komponentów z obiektu żądania (krok 3.), a następnie wywołują odpowiednie metody get (krok 4.).



To może być proste wyrażenie EL, takie jak:

```
${uzytkownik.nazwisko}
```

**1** Po otrzymaniu żądania dotyczącego informacji o użytkowniku **Kontroler** odwołuje się do komponentu modelu **ZarzadzanieUzytkownikiem**. Wywoływany komponent modelu wykorzystuje interfejs JDBC do uzyskania niezbędnych informacji z bazy danych, a następnie, na ich podstawie, tworzy komponent **Uzytkownik**.

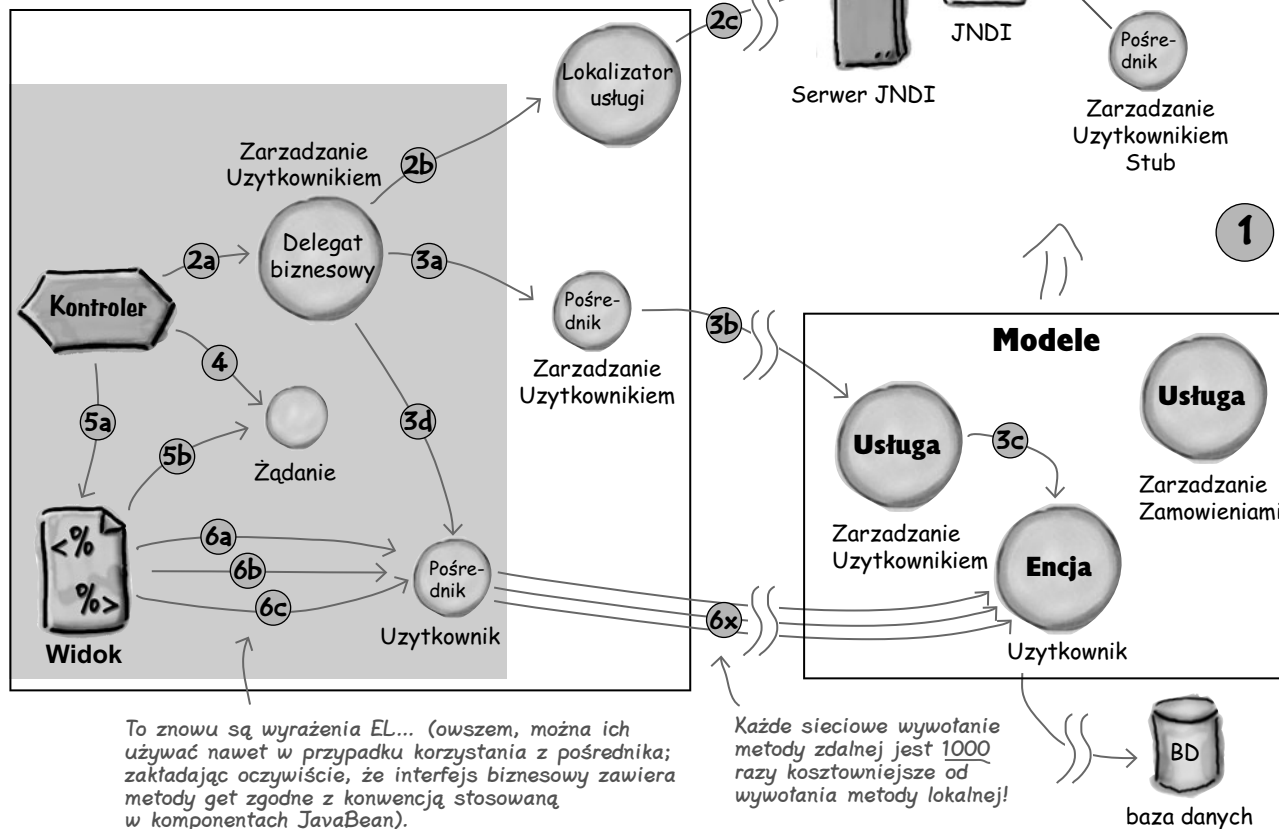
**2** Kontroler dołącza do obiektu żądania (w formie atrybutu) referencję do komponentu **Uzytkownik**.

**3** Kontroler przekazuje sterowanie do **Widoku** JSP. Strona JSP pobiera referencję do komponentu **Uzytkownik** z obiektu żądania.

**4** Widok wykorzystuje język EL do uzyskania właściwości komponentu **Uzytkownik** niezbędnych do obsługi przesłanego żądania.

## Porównaj schemat rozwiązania wykorzystującego model lokalny oraz model zdalny

Obszar, który na poniższym rysunku został wyróżniony szarym tłem, powinien być BARDZO podobny do rysunku przedstawionego na poprzedniej stronie, zwłaszcza jeśli weźmiemy pod uwagę to, że **DelegatBiznesowy** „udaje” komponent modelu **ZarządzanieUżytkownikiem**.



### Przegląd sześciu etapów:

- 1** Zarejestruj swoje usługi w JNDI.
- 2** Użyj delegata biznesowego i lokalizatora usługi do pobrania z JNDI obiektu pośrednika komponentu **ZarządzanieUżytkownikiem**.
- 3** Postępując się delegatem biznesowym oraz pośrednikiem, pobierz „komponent **Użytkownik**”, którym w tym przypadku jest kolejny pośrednik. Przekaż do kontrolera referencję do tego komponentu.
- 4** Dodaj referencję do pośrednika komponentu **Użytkownik** do obiektu żądania.

**5** Kontroler przekazuje obsługę żądania do **Widoku** JSP. Kod JSP pobiera z obiektu żądania referencję do (pośrednika) komponentu **Użytkownik**.

**6** Widok może wykorzystywać wyrażenia EL w celu uzyskania dostępu do właściwości komponentu **Użytkownik** i pobierać z nich wartości niezbędne do obsłużenia zgłoszonego żądania.

**WAŻNA UWAGA:** Za każdym razem, gdy kod JSP wywoła jedną z metod **get**, pośrednik komponentu **Użytkownik** i wykona odpowiednie sieciowe wywołanie metody zdalnej.

# Dobre i złe wiadomości...

Architektura przedstawiona na poprzedniej stronie z powodzeniem ukrywała wszelkie złożoności zarówno przed kontrolerem, jak i stronami JSP. Co więcej, z powodzeniem wykorzystywała wzorce projektowe Business Delegate i Service Locator.

## Złe wiadomości:

W momencie, w którym strona JSP chce pobrać dane, pojawiają się dwa problemy. Oba mają związek z tym, że komponent, na którym operuje kod JSP, w rzeczywistości jest tylko *pośrednikiem* *zdalnego obiektu*.

1. **Wszystkie te drobne wywołania sieciowe najprawdopodobniej będą miały poważny wpływ na wydajność aplikacji.** Każde wyrażenie EL powoduje wywołanie zdalnej metody. Takie wywołania nie tylko wiążą się z dodatkowym obciążeniem łączy sieciowych (i nieuniknionymi opóźnieniami), ale także przysparzają pewnych problemów serwerowi.

2. Kod JSP **NIE jest dobrym miejscem** do obsługi wyjątków, które mogą się pojawić w przypadku awarii zdalnego serwera.

## A dlaczego strona JSP nie może używać zwyczajnego komponentu zamiast pośrednika?

**P.** Skoro kod JSP ma się komunikować z komponentem **JavaBean**, skąd wziąć ten komponent?

**U.** Zazwyczaj komponenty te były zwracane przez *lokalny* obiekt modelu (usługi), a zatem czy nie można by ich uzyskiwać od *zdalnych* obiektów modelu (usługi)?

**P.** A w jaki sposób chciałbyś przesłać komponent przez sieć?

**U.** Ej... przecież jeśli tylko komponent można serializować, to RMI nie będzie miało najmniejszych problemów z jego przesłaniem.

**P.** Ale jakie korzyści mogłoby nam zapewnić takie rozwiązanie?

**U.** Przede wszystkim zamiast wielu małych wywołań zdalnych metod musielibyśmy wykonać tylko jedno duże wywołanie. Po drugie, strona JSP odwoływałaby się do obiektu lokalnego, dzięki czemu nie musielibyśmy się przejmować koniecznością obsługi zdalnych wyjątków.

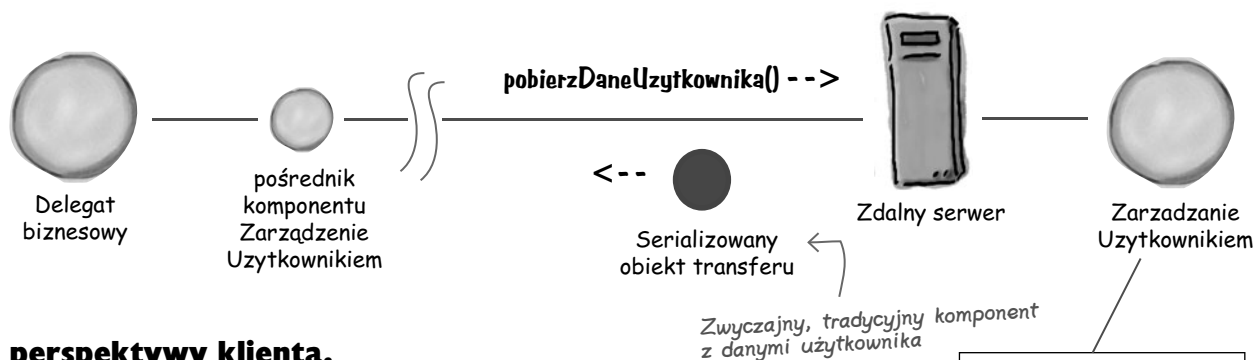
**P.** Zaraz, chwileczkę... zauważyłem jeden problem. Może nawet duży problem. Jeśli używamy komponentu po stronie klienta, to czy przechowywane w nim dane nie stają się nieaktualne w momencie ich wystania z serwera?

**U.** Owszem, masz rację, ale taka JEST cena kompromisu — wydajność a aktualność danych. Sam musisz zdecydować, co jest ważniejsze, opierając się przy tym na wymaganiach konkretnej aplikacji. Jeśli dane prezentowane przez widok w każdej chwili muszą bezwzględnie, bezwarunkowo i bez wyjątków przedstawiać aktualny stan bazy danych, to musisz się posługiwać zdalną referencją. Na przykład, jeśli wykonujesz trzy wywołania dotyczące danych użytkownika: `getNazwisko()`, `getImie()` oraz `getTelefon()`, to możesz dojść do wniosku, że dane te nie zmieniają się na tyle często, aby *ponowne* odwoływanie się (przy wykorzystaniu zdalnego obiektu) do bazy danych było usprawiedliwione — bowiem prawdopodobieństwo zmiany numeru telefonu użytkownika POMIĘDZY wywołaniami metody `getNazwisko()` oraz `getImie()` jest znikome.

Z drugiej strony, w przypadku tworzenia środowisk bardzo dynamicznych, w których użytkownicy wykonują transakcje 24 godziny na dobę przez 7 dni w tygodniu, możesz zdecydować, że wyświetlanie najbardziej aktualnych informacji JEST konieczne. Przesłanie komponentu do klienta oznaczałoby, iż będzie on dysponować kopią bazy danych z chwili tworzenia tego komponentu. Jednak dane te momentalnie zaczynają tracić swą aktualność, gdyż komponent nie dysponuje żadnym połączeniem z bazą danych.

## Czas zastosować wzorzec Transfer Object?

Jest całkiem prawdopodobne, że usługa biznesowa zostanie poproszona o wysłanie wszystkich lub prawie wszystkich danych wchodzących w skład dużego komunikatu; usługi często oferują tego rodzaju możliwości w swoim interfejsie programowym. Zazwyczaj usługi biznesowe tworzą duże obiekty Javy, które nadają się do serializacji i zawierają bardzo wiele składowych. Firma Sun nazywa je **Transfer Objects** (obiektami transferu); wszyscy pozostali określają ten model mianem wzorca **Data Transfer Object** (obiektu transferu danych). I wiesz co? Obiekty te spełniają dokładnie taką funkcję. (Tak... my też tak to oceniamy...).



### Z perspektywy klienta, wewnątrz delegata biznesowego

```
try {
 // Typ komponentu (i obiektu transferu).
 // Pobieramy obiekt transferu od pośrednika.
 Uzytkownik u = posrUzytkownika.pobierzDaneUzytkownika(idU);
} catch (RemoteException ex) {
 // Przechwytyjemy zdalne wyjątki i opakowujemy je w ramach wyjątków wyższego poziomu.
 throw new UzytkownikException();
}
```

To wszystko. W zupełnie niewidoczny sposób obiekt transferu jest serializowany, wysyłany i w końcu deserializowany na lokalnej stercie wirtualnej maszyny Javy na komputerze klienta. Od tej pory niczym się on nie różni od innych lokalnych komponentów JavaBean.



#### Dane zgromadzone w obiekcie transferu stają się nieaktualne!

Po przestaniu obiektu transferu na inny komputer traci on wszelkie związki ze źródłem danych, a przechowywane w nim informacje nie uwzględniają ewentualnych zmian w bazie danych, z której pochodzą. Dla każdego z przypadków użycia będziesz musiał zdecydować, czy zysk wydajności działania aplikacji jest wart utraty integralności i synchronizacji danych.

### Zarówno wzorzec Service Locator, jak i wzorzec Business Delegate upraszcza komponenty modelu

*Posłuchaj, jak nasi dwaj posiadacze czarnych pasów spierają się, który ze wzorców jest lepszy — Service Locator (lokalizator usługi) czy Business Delegate (delegat biznesowy).*

Service  
Locator



Business  
Delegate

Service Locator jest doskonałym wzorcem projektowym. Przede wszystkim, w odróżnieniu od wzorca Business Delegate, jeden obiekt lokalizatora usługi jest w stanie obsłużyć całą warstwę aplikacji.

Lokalizator usługi efektywniej obsługuje wywołania sieciowe. Po zlokalizowaniu pośredników lub pośredników usług jest w stanie gromadzić referencje do nich w pamięci podręcznej, redukując tym samym liczbę wywołań sieciowych, które trzeba będzie wykonać w przyszłości.

Trudne zadanie? Twoje proste dane biznesowe w ogóle nie robią na mnie żadnego wrażenia.

Ech, może programiści na tym skorzystają, ale Twój prosty wzorzec zapomina, że czasami działa w środowisku *sieciowym*. Dlatego też będzie wykonywać wiele wywołań kierowanych do usług biznesowych, zupełnie nie zwracając uwagi na dodatkowe *koszty* związane z realizacją wywołań sieciowych.

O tak, Twój słabiutki wzorzec wymaga *pomocy*, wszyscy o tym wiemy. Jednak nawet jeśli będziesz współpracować z obiektem transferu, to mogą dopaść Cię inne demony... nie zapomniałeś jeszcze swoich problemów z brakiem aktualności danych, nieprawdaż?

To fakt, ale lokalizator usługi musi komunikować się tylko z *jedną* zdalną encją, natomiast delegat biznesowy musi obsługiwać *wiele* różnych obiektów.

Z całym szacunkiem, ale zapominasz, że lokalizator usługi ma o wiele prostsze zadanie. Delegat biznesowy musi obsługiwać bardzo trudne zadanie, jakim jest komunikacja sieciowa z obiektami dynamicznymi, których zawartość może się zmieniać w dowolnej chwili.

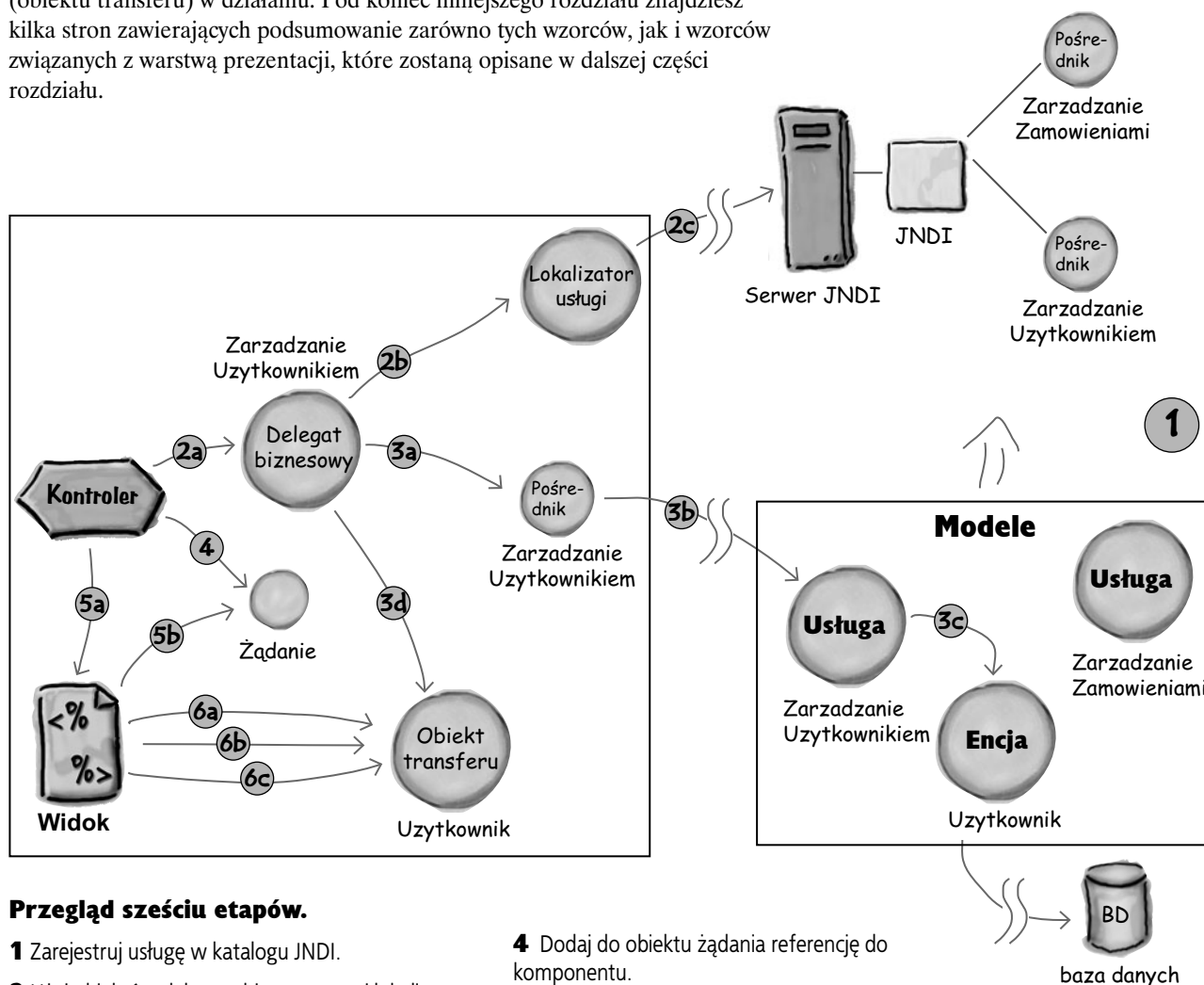
Delegaty biznesowe dają twórcom aplikacji internetowych znacznie większe *korzyści* niż lokalizatory usług.

Oho! Wzorzec Business Delegate wcale nie wstydzi się swoich związków ze wzorcem Transfer Object! Dzięki tej współpracy moje obiekty nie tylko pomagają programistom, ale też minimalizują liczbę wywołań sieciowych.

Nie, nie zapomniałem o tym. Jednak problemy te będzie można rozwiązać w chwili, kiedy zaczną się pojawiać. Nie możesz oczekiwać oszałamiających rezultatów bez choćby odrobiny dodatkowego wysiłku. W J2EE nic i nigdy nie jest czarno-białe.

## Wzorce warstwy biznesowej — krótki przegląd

W ramach podsumowania dyskusji o wzorcach warstwy biznesowej poniżej zamieściliśmy rysunek przedstawiający wzorce Business Delegate (delegata biznesowego), Service Locator (lokalizatora usługi) oraz Transfer Object (obiektu transferu) w działaniu. Pod koniec niniejszego rozdziału znajdziesz kilka stron zawierających podsumowanie zarówno tych wzorców, jak i wzorców związanych z warstwą prezentacji, które zostaną opisane w dalszej części rozdziału.



### Przegląd sześciu etapów.

- 1 Zarejestruj usługę w katalogu JNDI.
- 2 Użyj obiektów delegata biznesowego i lokalizatora usługi do uzyskania pośrednika komponentu ZarządzanieUżytkownikami.
- 3 Użyj obiektu delegata biznesowego i pośrednika do uzyskania „komponentu Użytkownik”, czyli w rzeczywistości odpowiedniego obiektu transferu. Przekaż kontrolerowi referencję do tego obiektu transferu.

- 4 Dodaj do obiektu żądania referencję do komponentu.

- 5 Kontroler przekazuje obsługę żądania do **widoku** JSP. Kod JSP pobiera z żądania referencję do obiektu transferu komponentu Użytkownik.

- 6 Widok JSP wykorzystuje wyrażenia EL do pobierania ze składowych obiektu transferu komponentu Użytkownik danych, które są mu potrzebne do obsłużenia żądania.

## Nasz pierwszy wzorzec po raz wtóry — MVC

Szczęśliwym zbiegiem okoliczności ten sam wzorzec projektowy, którego używaliśmy w niniejszej książce, pojawia się także w pytaniach egzaminacyjnych. Ostatnie dwa wzorce prezentowane w tym rozdziale są związane z warstwą prezentacji, podobnie jak opisywany wcześniej wzorzec Intercepting Filter (filtra przechwytyującego). W pierwszej kolejności dokończymy prezentację wzorca Model-View-Controller (MVC, model-widok-kontroler), kontynuując ją od miejsca, w którym wcześniej została przerwana. Tym razem prowadzone rozważania powinny nas doprowadzić do frameworku Struts oraz wzorca Front Controller (kontrolera frontowego).

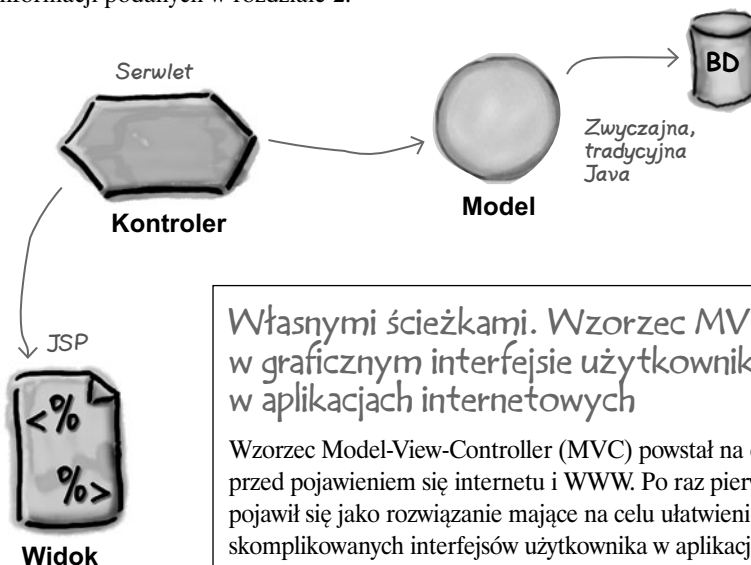
### Na czym skończyliśmy...

Dokonajmy szybkiego przypomnienia informacji podanych w rozdziale 2.

#### KONTROLER

Pobiera dane wprowadzone przez użytkownika (i przekazane w żądaniu), po czym próbuje określić znaczenie tych danych dla modelu.

Wydaje modelowi polecenie aktualizacji jego zawartości i udostępnia jego stan widokowi (stronom JSP), a następnie przekazuje obsługę żądania do stron JSP.



#### WIDOK

Jest odpowiedzialny za prezentację. Pobiera stan modelu z Kontrolera (choć nie robi tego w sposób bezpośredni — Kontroler umieszcza dane modelu w miejscu, w którym widok może je odnaleźć).

#### MODEL

Zawiera faktyczną logikę biznesową oraz przechowuje stan. Innymi słowy, model zna reguły dotyczące pobierania i aktualizacji stanu.

Zawartość Koszyka (oraz reguły określające, co z nią można zrobić) stanowiłyby część Modelu.

Model jest jedyną częścią aplikacji, która komunikuje się z bazą danych.

### Własnymi ścieżkami. Wzorzec MVC w graficznym interfejsie użytkownika oraz w aplikacjach internetowych

Wzorzec Model-View-Controller (MVC) powstał na długo przed pojawieniem się internetu i WWW. Po raz pierwszy pojawił się jako rozwiązanie mające na celu ułatwienie tworzenia skomplikowanych interfejsów użytkownika w aplikacjach tworzonych w języku Smalltalk; jedną z podstawowych cech tego wzorca była możliwość informowania widoku o wszelkich zmianach zachodzących w modelu.

Później jednak wzorzec ten znalazł zastosowanie w rozwiązaniach internetowych, choć w tym przypadku widok znajduje się w przeglądarce WWW i nie ma możliwości jego automatycznej aktualizacji w razie pojawienia się jakichś zmian modelu należących do warstwy internetowej. W tej książce w całości skoncentrujemy się właśnie na internetowej wersji wzorca MVC.

Warto na koniec wspomnieć, że w prezentowanych rozważaniach zawsze mamy na myśli wzorzec MVC w wersji 2., a nie we wcześniejszych wersjach — 1. lub 1.5.

## Model MVC w prawdziwych aplikacjach internetowych

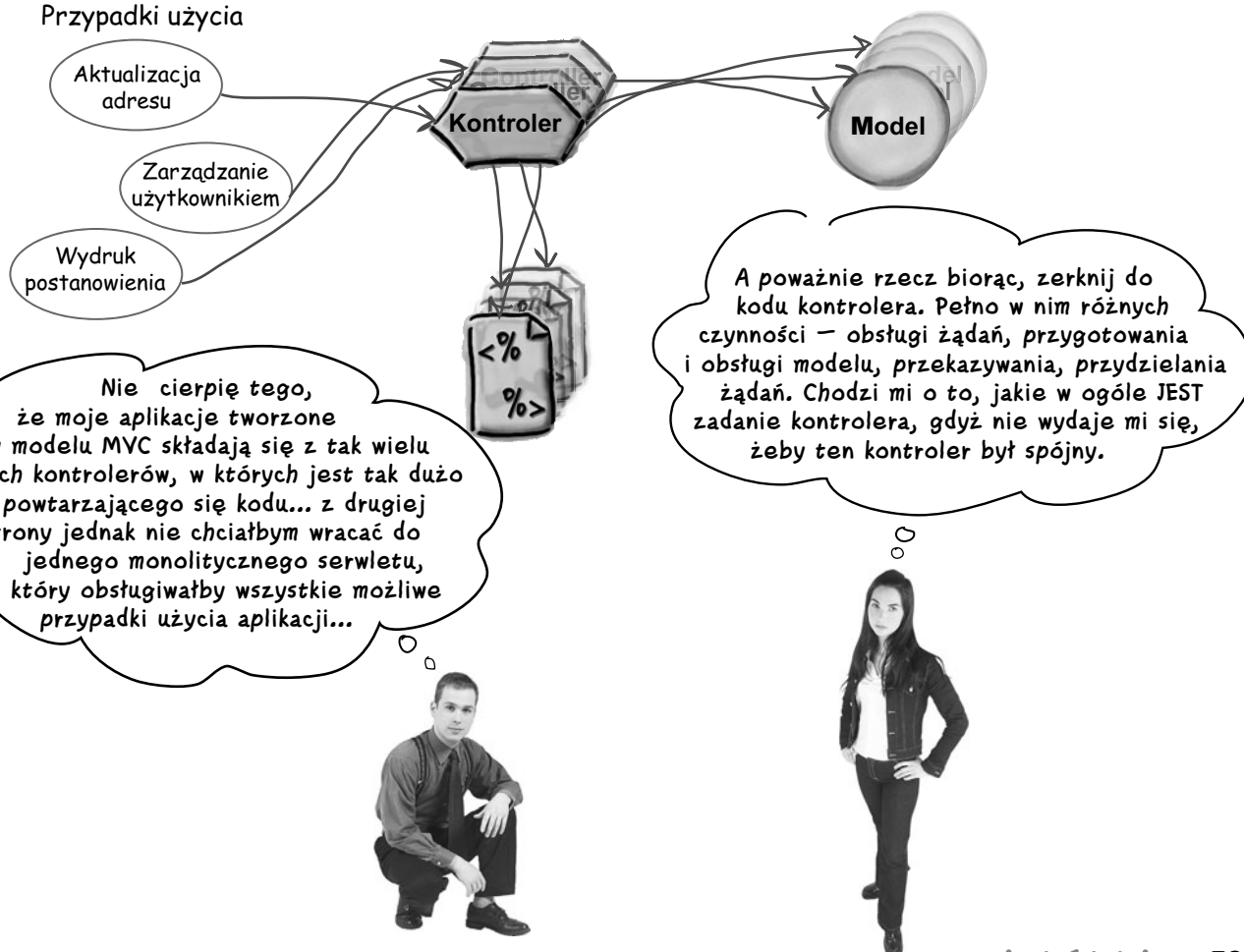
Dawno temu, w drugim rozdziale niniejszej książki, przedstawiliśmy ćwiczenie z cyklu „WYTEŻ UMYŚŁ” dotyczące potencjalnych problemów związanych z architekturą MVC zastosowaną w aplikacji swatającej. Zobaczmy, w którym miejscu przerwaliśmy nasze rozważania, i zacznijmy poszukiwać odpowiedzi na pytanie, które niewątpliwie dręczyło Cię przez wszystkie te rozdziały — jakie rozwiązanie mogłoby być lepsze od modelu MVC?

Dla każdego przypadku użycia przeglądarki będzie istnieć odpowiedni zbiór komponentów Modelu, Widoku oraz Kontrolera. Komponenty te można dowolnie mieszać i łączyć, aby zaspokoić wymagania różnych przypadków użycia.

W aplikacji randkowej występował problem polegający na tym, iż dysponowaliśmy wieloma różnymi kontrolerami. Choć rozwiązanie to wydawało się poprawne z obiektowego punktu widzenia, to jednak we wszystkich tych kontrolerach w całej naszej aplikacji występowały identyczne fragmenty kodu, co bynajmniej nie napawało nas optymizmem, gdy myśleliśmy o utrzymywaniu aplikacji oraz jej elastyczności.

### Jedna aplikacja w modelu MVC będzie mieć wiele modeli, widoków i kontrolerów.

Przypadki użycia



# Prezentacja kontrolera MVC

Sprawdźmy, czy zgodzimy się z tym, co zostało powiedziane na temat kontrolerów. W pierwszej kolejności przypomnimy, jakie jest zadanie serwletu kontrolera.

### Pseudokod ogólnego kontrolera MVC:

```
public class SerwletKontrolera extends HttpServlet {
 public void doPost(zadanie, odpowiedz) {
 ❶ String c = zadanie.getParameter("dataPocatkowa");
 // wykonanie konwersji daty dla wartości uzyskanej z parametru
 // sprawdzenie, czy data mieści się w dostępnym zakresie
 // jeśli wykryto błędy, należy skierować użytkownika
 // na trwale zakodowaną stronę "spróbuj jeszcze raz".
 // wywołujemy określone na stałe komponenty modelu
 ❷ // dodajemy wyniki wykonania komponentów modelu do żądania
 // może to być, na przykład, referencja do komponentu
 // przekazujemy obsługę żądania do widoku JSP
 ❸ // (który także jest podany na stałe).
 }
}
```

Obsługa  
parametrów żądania

Obsługa modelu

Obsługa widoku



### Zaostrz ołówek

#### Jakie zasady narusza ten komponent?

Podaj przynajmniej trzy (lub więcej) zasad związanych z projektowaniem oprogramowania, które narusza przedstawiony komponent.

# Ulepszanie kontrolerów MVC

Oprócz braku spójności kontroler przedstawiony na poprzedniej stronie jest ściśle związany z komponentami modelu i widoku. Co więcej, pojawia się jeszcze jedno ostrzeżenie związane z powtarzającym się kodem.

Trzy podstawowe zadania kontrolera	Lepszy sposób ich realizacji
1 Uzyskanie i obsługa parametrów żądania	To zadanie należy przekazać dodatkowemu, wyspecjalizowanemu komponentowi weryfikującemu, który odczyta informacje podane w formularzu, dokona ich konwersji, sprawdzi ich poprawność, obsłuży ewentualne błędy związane z poprawnością danych oraz stworzy obiekt, w którym zostaną zapisane wartości parametrów.
2 Wywołanie modelu	Hmm... nie podoba nam się trwałe kodowanie informacji o modelu w serwlecie kontrolera. Może dałoby się określać je w sposób deklaratywny, podając nazwy odpowiednich modeli w jakimś własnym deskrypcorze? Kontroler mógłby odczytywać zawartość tego deskryptora i na podstawie żądania określać, jakie modele należy zastosować.
3 Przekazanie sterowania do widoku	Dlaczego także tych informacji nie podawać w sposób deklaratywny? W ten sposób, bazując na adresie URL przekazanym w żądaniu, kontroler mógłby określić (na podstawie naszego deskryptora), do jakiego widoku przekazać obsługę żądania.

## Nowy (i krótszy) pseudokod kontrolera

```
public class ServletKontrolera extends HttpServlet {

 public void doPost(zadanie, odpowiedz) {
 String c = zadanie.getParameter("dataPoczatkowa");

 // wywołaj określony deklaratywnie komponent weryfikujący
 // (niech ten kontroler obsługuje także błędy, które
 // pojawiły się podczas sprawdzania poprawności danych)

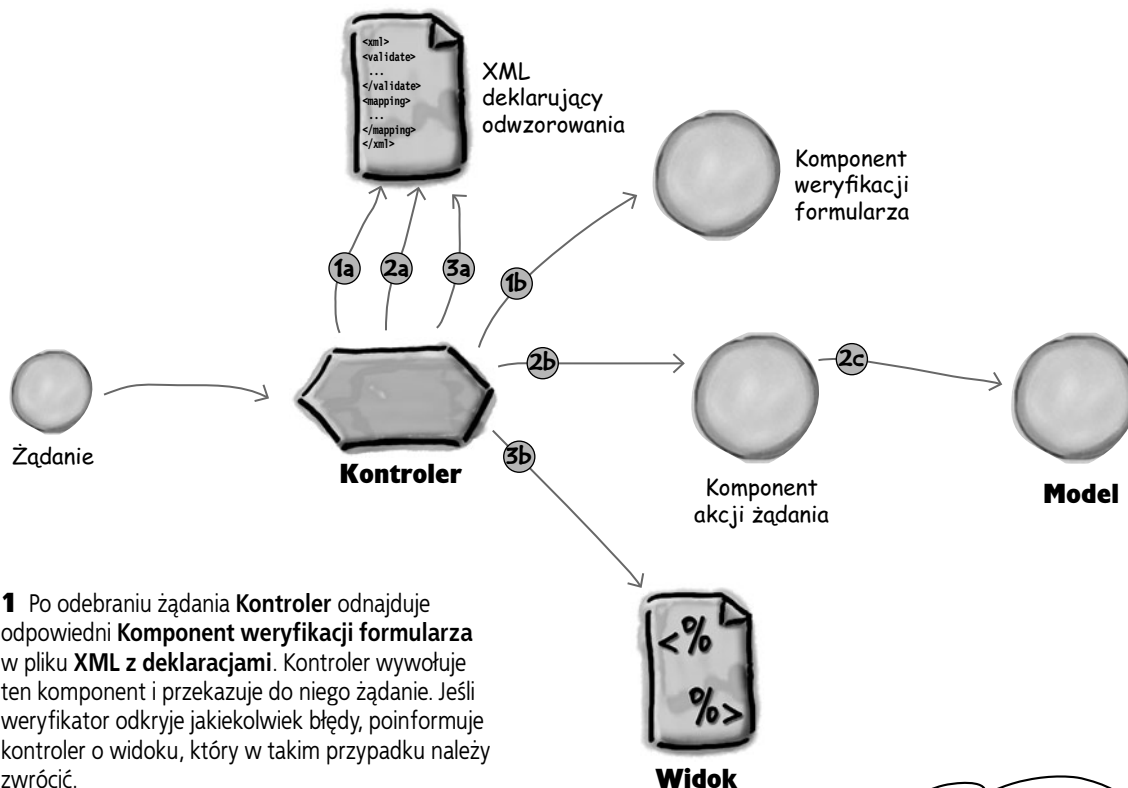
 // wywołaj określony deklaratywnie komponent służący do
 // przetwarzania żądania i tak samo określone komponenty
 // modelu

 // przekaz obsługę żądania do deklaratywnie wskazanego widoku
 }
}
```



# Projektowanie naszego fantastycznego kontrolera

Narysujmy jeden z naszych niesłownych schematów architektury, aby się przekonać, jak wygląda nasz obecny kontroler oraz wszystkie jego elementy pomocnicze.

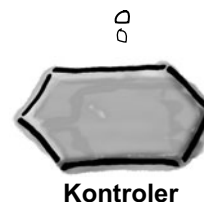


**1** Po odebraniu żądania **Kontroler** odnajduje odpowiedni **Komponent weryfikacji formularza** w pliku **XML z deklaracjami**. Kontroler wywołuje ten komponent i przekazuje do niego żądanie. Jeśli weryfikator odkryje jakiegokolwiek błąd, poinformuje kontroler o widoku, który w takim przypadku należy zwrócić.

**2** Bazując na informacjach podanych w pliku XML z deklaracjami, kontroler odnajduje i wywołuje **Komponent akcji żądania**, który z kolei wywołuje model.

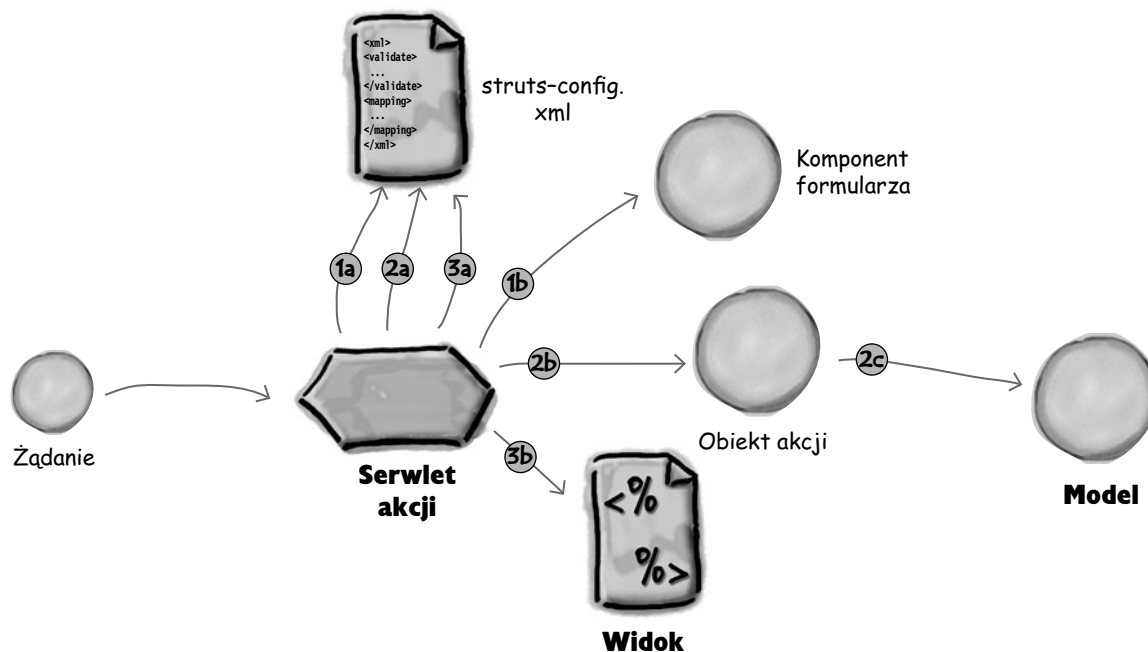
**3** Bazując na informacjach podanych w pliku XML z deklaracjami, kontroler odnajduje i wywołuje **Widok**.

Zaraz, chwilę... Już to gdzieś widziałem! Próbuje ukryć framework STRUTS!



# Tak! To Struts w zarysie

Oczywiście to tylko krótka prezentacja, w której pominięto wiele istotnych szczegółów, jednak właśnie na tym polega idea leżąca u podstaw stworzenia frameworku Struts. Przyjrzyjmy się bardziej szczegółowo temu frameworkowi, uwzględniając przy tym zmianę kilku ważnych nazw...



## Podstawowe komponenty frameworku Struts

**Serwlet akcji** — to serwlet typu `ActionServlet`; w jednej aplikacji będziesz potrzebował tylko **jednego** takiego serwletu. Najlepsze jest jednak to, że nawet nie będziesz musiał go pisać — **Struts oferuje gotowe rozwiązania**.

**Komponent formularza** — dla każdego formularza obsługiwane przez aplikację będziesz musiał napisać jeden taki komponent. Są to komponenty Javy, a kiedy serwlet akcji wywoła już metody `set` takiego komponentu (w celu zapisania w nim parametrów formularza), kolejną czynnością będzie wywołanie metody `validate()`. Metoda ta doskonale nadaje się do umieszczenia kodu konwertującego dane oraz obsługującego wszelkie błędy z nimi związane.

**Obiekt akcji** — ogólnie rzecz biorąc, akcja odpowiada pojedynczej czynności wchodzącej w skład przypadku użycia. Obiekt akcji dysponuje metodą `execute()`, przypominającą nieco metody zwrotne, która doskonale nadaje się do *pobrania* zweryfikowanych parametrów formularza i wywołania komponentów modelu. Obiekty tego rodzaju można sobie wyobrazić jako „uproszczone wersje” serwletów.

**struts-config.xml** — to swoisty deskryptor wdrożenia frameworku Struts. Będą w nim umieszczane: odwzorowania **adresów URL oraz akcji, akcji oraz komponentów formularzy**, jak również **akcji i widoków**.



## Czy Struts jest kontenerem?

Oficjalnie Struts jest uznawany za framework do tworzenia aplikacji.

Frameworki to kolekcje interfejsów i klas zaprojektowanych z myślą o wzajemnej współpracy w celu rozwiązywania problemów określonego typu. W przypadku frameworku Struts przestrzenią tych problemów są aplikacje internetowe. Jego celem jest „niesienie programistom pomocy w tworzeniu i utrzymaniu złożonych aplikacji internetowych”.

A zatem Struts nie jest kontenerem, choć pod pewnymi względami jest do niego podobny.

### Pięć największych podobieństw pomiędzy frameworkiem Struts a kontenerem serwletów

**1 Deklaratywność** — zarówno Struts, jak i kontener serwletów wykorzystują plik XML do deklaratywnej konfiguracji aplikacji.

**2 Cykl życia** — zarówno Struts, jak i kontener serwletów zapewniają cykl życia określonych typów obiektów.

**3 Metody zwrotne** — zarówno Struts, jak i kontener serwletów w automatyczny sposób wywołują kluczowe metody zwrotne związane z cyklem istnienia obiektów.

**4 Interfejs programowy** — zarówno Struts, jak i kontener serwletów udostępniają ściśle zdefiniowany interfejs programowy najważniejszych obsługiwanych typów obiektów.

**5 Kontrola aplikacji** — zarówno Struts, jak i kontener serwletów udostępniają kontrolowane środowiska, w których są wykonywane aplikacje. Stanowią zatem „okno na świat” dla tworzonych aplikacji internetowych.



**Relax**

Na egzaminie nie ma niczego na temat frameworku Struts

Przystępując do egzaminu, MASZ wiedzieć, jakie jest przeznaczenie i funkcje wzorca Front Controller (a Struts jest tylko wzbogaconą wersją implementacji tego wzorca). Jednak na pewno nie pojawią się na nim żadne pytania dotyczące samego frameworku Struts. A zatem możesz się rozluźnić i spokojnie czytać dalsze informacje zamieszczone w tym rozdziale bez konieczności zapamiętywania jakichkolwiek szczegółów.

Nasuwa mi się jedna analogia... napisaliście, że Struts wykorzystuje „metody zwrotne” oraz deskryptor wdrożenia. A zatem czy Struts nie jest takim minikontenerem?



We frameworku Struts awansowałem — teraz jestem „serwletem akcji”. Czasami nazywają mnie także kontrolerem frontowym. (Swoją drogą, akurat to jest na egzaminie).



**Serwlet  
akcji**

## Jak dopasować wzorzec Front Controller do naszego modelu?

Ach tak. Front Controller (kontroler frontowy) jest kolejnym wzorcem J2EE i tak się składa, że pytania na jego temat pojawiają się na egzaminie. Prawdę mówiąc, **Struts jest w istocie wyszukany przykładem zastosowania wzorca Front Controller**. Podstawową ideą tego wzorca jest zastosowanie jednego komponentu (zazwyczaj jest nim serwet, lecz równie dobrze może nim być strona JSP), który będzie działać jako pojedynczy punkt kontrolny obsługujący warstwę prezentacji aplikacji internetowej. W przypadku zastosowania tego wzorca, wszystkie żądania kierowane do aplikacji trafiają do jednego kontrolera, który obsługuje przekazywanie żądań w odpowiednie miejsca aplikacji.

W praktyce rzadko spotyka się implementacje wzorca Front Controller istniejące całkowicie samodzielnie. Nawet w naprawdę bardzo prostych implementacjach zazwyczaj dodatkowo stosuje się inny wzorzec projektowy J2EE nazywany **Application Controller** (kontrolerem aplikacji). Struts zawiera klasę o nazwie `RequestProcessor`, która jest w całości odpowiedzialna za obsługę żądań HTTP.

Choć na egzaminie mogą się pojawić pytania dotyczące wzorca Front Controller, w praktyce wystarczy zapamiętać zalety frameworku Struts oraz to, że Struts jest właśnie kontrolerem frontowym wyposażonym w kilka dodatkowych „wodotrysków”.

### Osiem możliwości, które Struts dodaje do wzorca Front Controller

**1. Kontrola deklaratywna.** Struts pozwala na tworzenie deklaratywnych odwzorowań pomiędzy adresami URL przekazywanymi w żądaniu, obiektami weryfikującymi, obiektami wywołującymi komponenty modelu oraz widokami.

**2. Automatyczne przekazywanie żądań.** Metoda `Action.execute()` zwraca symboliczne polecenie `ActionForward` informujące serwet akcji, do którego widoku należy skierować dane żądanie. Takie rozwiązanie tworzy kolejną warstwę abstrakcji (i przyczynia się do zmniejszenia stopnia powiązań) pomiędzy kontrolerem a komponentami widoku.

**3. Źródła danych.** Struts jest w stanie zarządzać źródłami danych.

**4. Znaczniki niestandardowe.** Struts udostępnia dziesiątki znaczników niestandardowych.

**5. Obsługa aplikacji wielojęzycznych.** Zarówno klasy związane z obsługą błędów, jak i znaczniki niestandardowe są przystosowane do obsługi wielu języków.

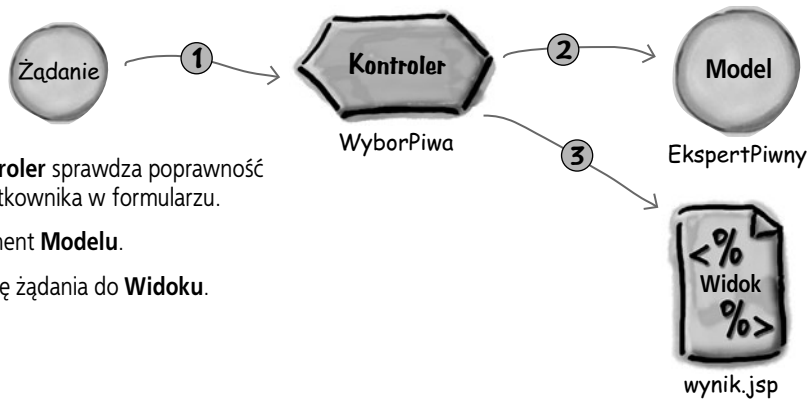
**6. Deklaratywna weryfikacja poprawności.** Struts zawiera mechanizmy weryfikacji, dzięki którym umieszczanie kodu weryfikującego poprawność danych w komponentach formularzy nie jest konieczne. Reguły weryfikacji są podawane w pliku XML i można je zmieniać bez wprowadzania jakichkolwiek zmian w kodzie komponentów formularzy.

**7. Globalna obsługa wyjątków.** Struts udostępnia deklaratywny mechanizm obsługi błędów podobny do elementu `<error-page>` stosowanego w deskrytorze wdrożenia. Okazuje się jednak, że we frameworku Struts wyjątki można definiować dla kodu aplikacji umieszczonego w obiektach akcji.

**8. Pluginy.** Struts udostępnia interfejs `Plugin` definiujący dwie metody: `init()` oraz `destroy()`. Można zatem tworzyć własne wtyczki (ang. *plug-ins*) rozszerzające możliwości aplikacji tworzonych z wykorzystaniem tego frameworku. Taka wtyczka jest wykorzystywana między innymi do inicjalizacji mechanizmów weryfikacji danych.

# Przystosowanie aplikacji piwnej do korzystania z frameworku Struts

Dosyć tej teorii, napiszmy w końcu jakąś aplikację korzystającą z frameworku Struts. Przede wszystkim przypomnijmy naszą aplikację piwną stworzoną na bazie modelu MVC, a przedstawioną w rozdziale 3. Przystosowanie aplikacji do korzystania z frameworku Struts wymaga wprowadzenia modyfikacji tylko w jednym miejscu — w kontrolerze MVC. (Ani model, ani widok nie zmieniają się).



- 1 Po otrzymaniu żądania **Kontroler** sprawdza poprawność informacji podanych przez użytkownika w formularzu.
- 2 Kontroler wywołuje komponent **Modelu**.
- 3 Kontroler przekazuje obsługę żądania do **Widoku**.

## Kod kontrolera MVC (z rozdziału 3.)

```
package com.example.web;
```

```
import com.example.model.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;
```

```
public class WyborPiwa extends HttpServlet {
```

```
 public void doPost(HttpServletRequest request,
 HttpServletResponse response)
 throws IOException, ServletException {
```

```
 String c = request.getParameter("kolor"); ← Weryfikacja poprawności danych nie
 jest tu przesadnie rozbudowana. :)
```

```
 EkspertPiwny ep = new EkspertPiwny();
 List wynik = ep.getMarki(c);
```

```
 request.setAttribute("style", wynik); ← Wywołanie modelu.
```

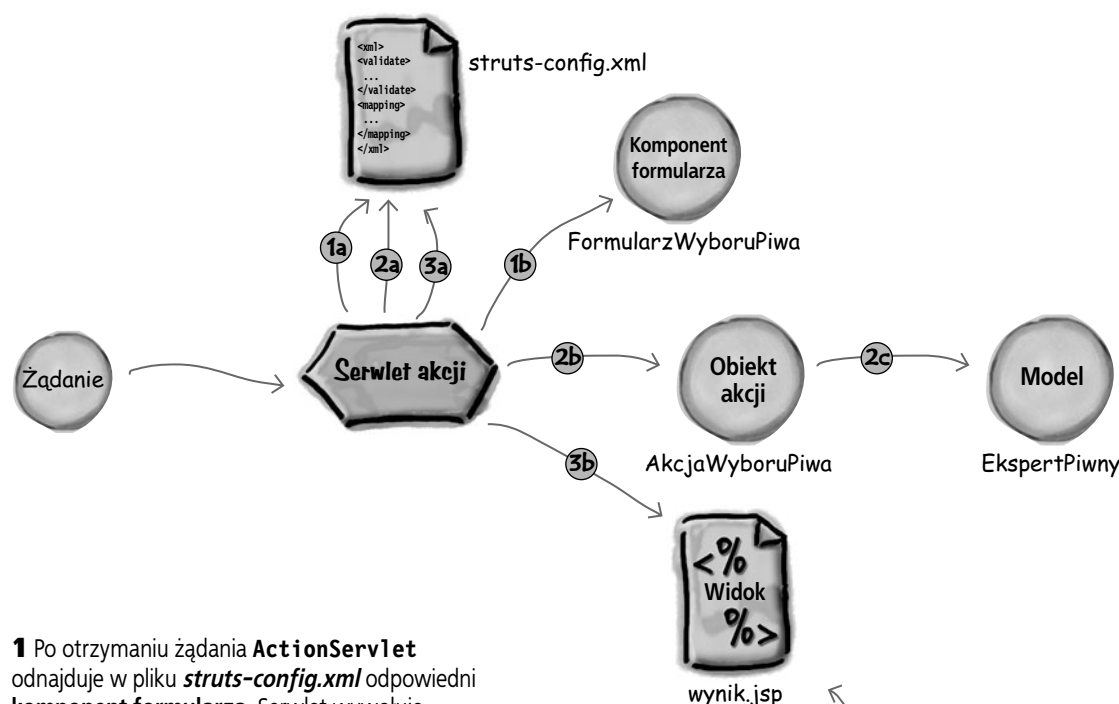
```
 RequestDispatcher disp = request.getRequestDispatcher("wynik.jsp");
```

```
 disp.forward(request, response);
```

```
 ↑ Przekazanie żądania do trwale
 zakodowanego Widoku.
```

# Architektura aplikacji piwnej w wersji bazującej na frameworku Struts

Poniżej przedstawiliśmy architekturę aplikacji piwnej w całości korzystającej z frameworku Struts...



**1** Po otrzymaniu żądania **ActionServlet** odnajduje w pliku **struts-config.xml** odpowiedni **komponent formularza**. Serwlet wywołuje następnie logikę związaną z weryfikacją informacji podanych w formularzu. Jeśli komponent znajdzie jakiegokolwiek błędy, to informacje na ich temat zostaną zapisane w obiekcie **ActionErrors**.

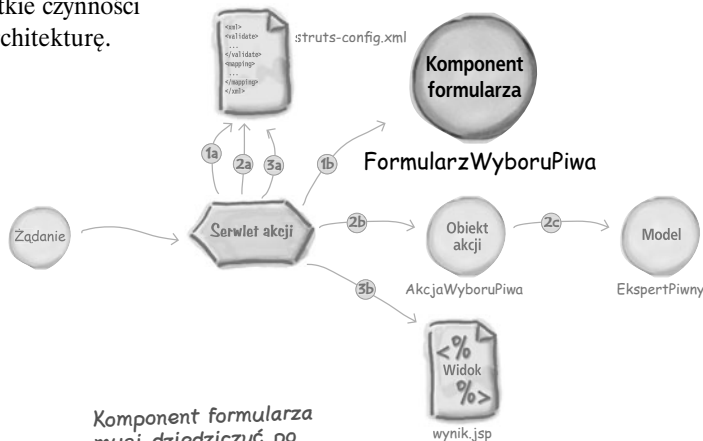
**2** Na bazie informacji podanych w pliku **struts-config.xml** **ActionServlet** określa i wywołuje odpowiedni obiekt **Action**, który z kolei wywołuje model i zwraca serwletowi akcji obiekt **ActionForward**.

**3** Dysponując niezbędnymi informacjami o odwzorowaniu, pobranymi wcześniej z pliku **struts-config.xml**, **ActionServlet** używa obiektu **ActionForward** do przekazania obsługi żądania do odpowiedniego komponentu widoku.

No dobrze, w porządku, tak naprawdę to w aplikacji bazującej na frameworku Struts widok "ulegnie" zmianie. Przede wszystkim Struts udostępnia bibliotekę znaczników zawierającą znacznik `<html:error/>` prezentujący błędy, które wystąpiły podczas weryfikacji informacji podanych przez użytkownika w formularzu. Co więcej, biblioteka znaczników HTML udostępnia znaczniki pozwalające na wypełnienie formularza oryginalną zawartością w przypadku wystąpienia błędów podczas weryfikacji.

# Komponent formularza bez tajemnic

Pamiętaj, że zadaniem komponentu formularza jest sprawdzenie, czy informacje podane przez użytkownika w formularzu są poprawne. Niewątpliwą zaletą frameworku Struts jest to, że wszystkie czynności związane z tą weryfikacją zostały wbudowane w jego architekturę.



```
package com.example.web;
```

```
// Struts – instrukcje importu
import org.apache.struts.action.ActionMapping;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionMessage;
import org.apache.struts.action.ActionErrors;
```

```
import javax.servlet.http.HttpServletRequest;
```

```
public class FormularzWyboruPiwa extends ActionForm {
```

```
 private String kolor;
 public void setKolor(String kolor) {
 this.kolor = kolor;
 }
```

```
 public String getKolor() {
 return kolor;
 }
```

```
 private static final String POPRAWNE_KOLORY = "jasne,ciemne,brązowe,bursztynowe";
 public ActionErrors validate(ActionMapping odwzorowanie,
 HttpServletRequest zadanie) {
 ActionErrors błedy = new ActionErrors();
```

```
 if (POPRAWNE_KOLORY.indexOf(kolor) == -1) {
 błedy.add("kolor", new ActionError("bład.poleKolor.złaWartosc"));
 }
 return błedy;
 }
```

```
}
```

Komponent formularza musi dziedziczyć po klasie ActionForm.

Zazwyczaj będziesz chciał, by komponenty formularzy posiadały metody set i get odpowiadające wszystkim parametrom formularza.

Serwlet ActionServlet wywołuje metodę validate().

Struts udostępnia klasę ActionErrors służącą do zarządzania błędami weryfikacji.

Konstruktor klasy ActionError wymaga przekazania łańcucha znaków (obiektu String) stanowiącego symboliczny klucz pliku zasobów. Rozwiązanie to ma na celu ułatwienie tworzenia aplikacji wielojęzycznych.

## Jak działa obiekt akcji?

Podstawowym zadaniem obiektu akcji jest przydzielanie żądań. Jest on wywoływany przez serwlet `ActionServlet`, który wywołuje metodę `execute()` obiektu akcji.

```
package com.example.web;
```

```
// Instrukcje importu niezbędne dla modelu
import com.example.model.*;
import java.util.*;
```

```
// Instrukcje importu Struts
import org.apache.struts.action.Action;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
```

```
// Instrukcje importu dla serwletu
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
```

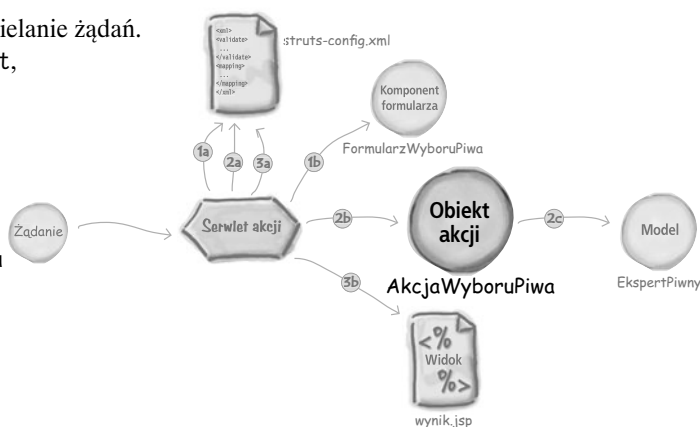
```
public class AkcjaWyborPiwa extends Action {

 public ActionForward execute(ActionMapping odwzorowanie,
 ActionForm formularz,
 HttpServletRequest zadanie,
 HttpServletResponse odpowiedz) {
```

```
 // Rzutujemy formularz na typ używany w naszej aplikacji
 FormularzWyboruPiwa mojFormularz = (FormularzWyboruPiwa) formularz;
```

```
 // Przetworzenie logiki biznesowej
 EkspertPiwny ep = new EkspertPiwny();
 List wyniki = ep.getMarki(mojFormularz.getKolor());
 // Przesłanie żądania do widoku
 // (i zapisanie danych w zasięgu żądania)
 zadanie.setAttribute("style", wyniki);
 return odwzorowanie.findForward("pokaz_wyniki");
}
```

```
}
```



Twój kontroler MUSI dziedziczyć po klasie `Action`.

Argument przesłany z serwletu `ActionServlet`, dzięki któremu możemy zwrócić odpowiedni widok.

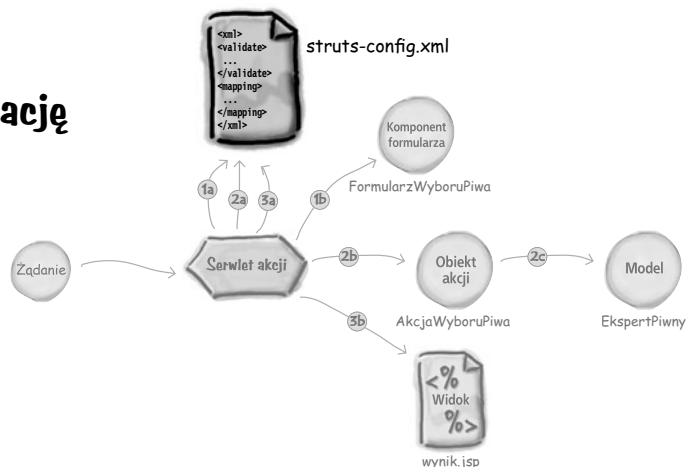
Ten obiekt zapewnia dostęp do zweryfikowanych parametrów formularza.

Przesłanie parametrów formularza do modelu.

Metoda `execute()` zwraca do serwletu obiekt typu `ActionForward`, który informuje framework Struts o konieczności przekazania obsługi żądania do kolejnego, odpowiedniego widoku. Te symboliczne „polecenia przekazania” są deklarowane w pliku `struts-config.xml`.

## struts-config.xml — łączenie wszystkich elementów w jedną aplikację

Plik *struts-config.xml* jest odpowiednikiem deskryptora wdrożenia w aplikacjach internetowych. Można mu nadać dowolną nazwę, choć zwyczajowo stosowana jest właśnie nazwa *struts-config.xml*. W tym pliku, podobnie jak w deskryptorze wdrożenia, są deklarowane i odwzorowywane wszystkie elementy aplikacji internetowej. Takie rozwiązanie pozwala na zmniejszenie stopnia powiązań pomiędzy poszczególnymi elementami aplikacji.



```
<?xml version="1.0" encoding="ISO-8859-2" ?>
<!DOCTYPE struts-config PUBLIC
 "-//Apache Software Foundation//DTD Struts Configuration 1.3//EN"
 "http://struts.apache.org/dtds/struts-config_1_3.dtd">
```

```
<struts-config>
```

```
<form-beans>
```

```
<form-bean name="formularzWyboruPiwa"
 type="com.example.web.FormularzWyboruPiwa" />
```

```
</form-beans>
```

```
<action-mappings>
```

```
<action path="/WyborPiwa"
 type="com.example.web.AkcjaWyboruPiwa"
 name="formularzWyboruPiwa" scope="request"
 validate="true" input="/formularz.jsp">
```

```
<forward name="pokaz_wyniki"
 path="/wynik.jsp" />
```

```
</action>
```

```
</action-mappings>
```

```
<message-resources parameter="ApplicationResources" />
```

```
</struts-config>
```

Element `<form-bean>` deklaruje symboliczną nazwę oraz klasę obiektu komponentu formularza

Element `<action>` odwzorowuje ścieżkę URL oraz klasę kontrolera; należy pamiętać, że w pliku konfiguracyjnym Struts NIE stosuje się rozszerzenia `.do`.

Oprócz tego element `<action>` kojarzy komponent formularza z akcją. Odwzorowanie to jest określone przy użyciu symbolicznej nazwy komponentu formularza. Struts utworzy wskazany komponent i zapisze go we wskazanym zasięgu. Jeśli jest wykonywana weryfikacja danych i metoda `validate()` zwróci informacje o wystąpieniu błędów, to atrybut `input` deklaruje Widok odpowiedzialny za wyświetlenie komunikatów o błędach; zazwyczaj jest to ten sam formularz, w którym użytkownik wpisywał informacje.

Element `<forward>` tworzy odwzorowanie pomiędzy symboliczną nazwą widoku, używaną przez obiekt akcji, oraz fizyczną ścieżką dostępu do komponentu widoku.

## Specyfikacja Struts w deskrytorze wdrożenia web.xml

Z punktu widzenia kontenera servlet akcji jest zwykłym servletem. Dlatego też należy go odpowiednio zadeklarować i upewnić się, że wszystkie żądania kierowane do aplikacji internetowej będą trafiać właśnie do niego.

```
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
 version="2.4">
```

*<!-- Definicja servletu kontrolera -->*

```
<servlet>
```

```
 <servlet-name>KontrolerFrontowy</servlet-name>
```

```
 <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
```

```
</servlet>
```

Nadawanie servletowi ActionServlet nazwy KontrolerFrontowy nie jest obowiązkowe, ale będzie Ci przypominać o funkcji, którą ten servlet pełni w ramach aplikacji.

*<!-- wskazanie pliku konfiguracyjnego Struts -->*

```
<init-param>
```

```
 <param-name>config</param-name>
```

```
 <param-value>/WEB-INF/struts-config.xml</param-value>
```

```
</init-param>
```

Parametr inicjalizacji nazwany config informuje servlet ActionServlet, gdzie można znaleźć plik konfiguracyjny frameworku Struts.

*<!-- Gwarancja załadowania servletu podczas uruchamiania aplikacji -->*

```
<load-on-startup>1</load-on-startup>
```

*<!-- Odzworowanie kontrolera Struts -->*

```
<servlet-mapping>
```

```
 <servlet-name>KontrolerFrontowy</servlet-name>
```

```
 <url-pattern>*.do</url-pattern>
```

```
</servlet-mapping>
```

*<!-- KONIEC: odzworowania kontrolera Struts -->*

Servlet ActionServlet ma skomplikowaną metodę init(), a zatem najlepiej będzie załadować go podczas uruchamiania aplikacji.

Ojej! Ten jeden servlet będzie obsługiwać WSZYSTKIE żądania kierowane do aplikacji (zakładając, że w adresach URL podawanych w żądaniu pojawi się rozszerzenie .do).

```
</web-app>
```



### Deskryptor wdrożenia Struts powinien nosić nazwę „struts-config.xml”

W PRZECIWNYM razie konieczne będzie zadeklarowanie w deskrytorze wdrożenia web.xml parametru inicjalizacji config definiującego nazwę pliku deskryptora. Jeśli jednak użyjemy domyślnej nazwy struts-config.xml, Struts będzie w stanie odszukać ten plik automatycznie, bez żadnych parametrów inicjalizacji. Niemniej jednak, nawet w przypadku stosowania nazwy standardowej, deklarowanie tego pliku w deskrytorze wdrożenia uważane jest za „dobry zwyczaj”.

# Zainstaluj Struts i po prostu go uruchom!

Instalacja frameworku Struts jest bardzo prosta.

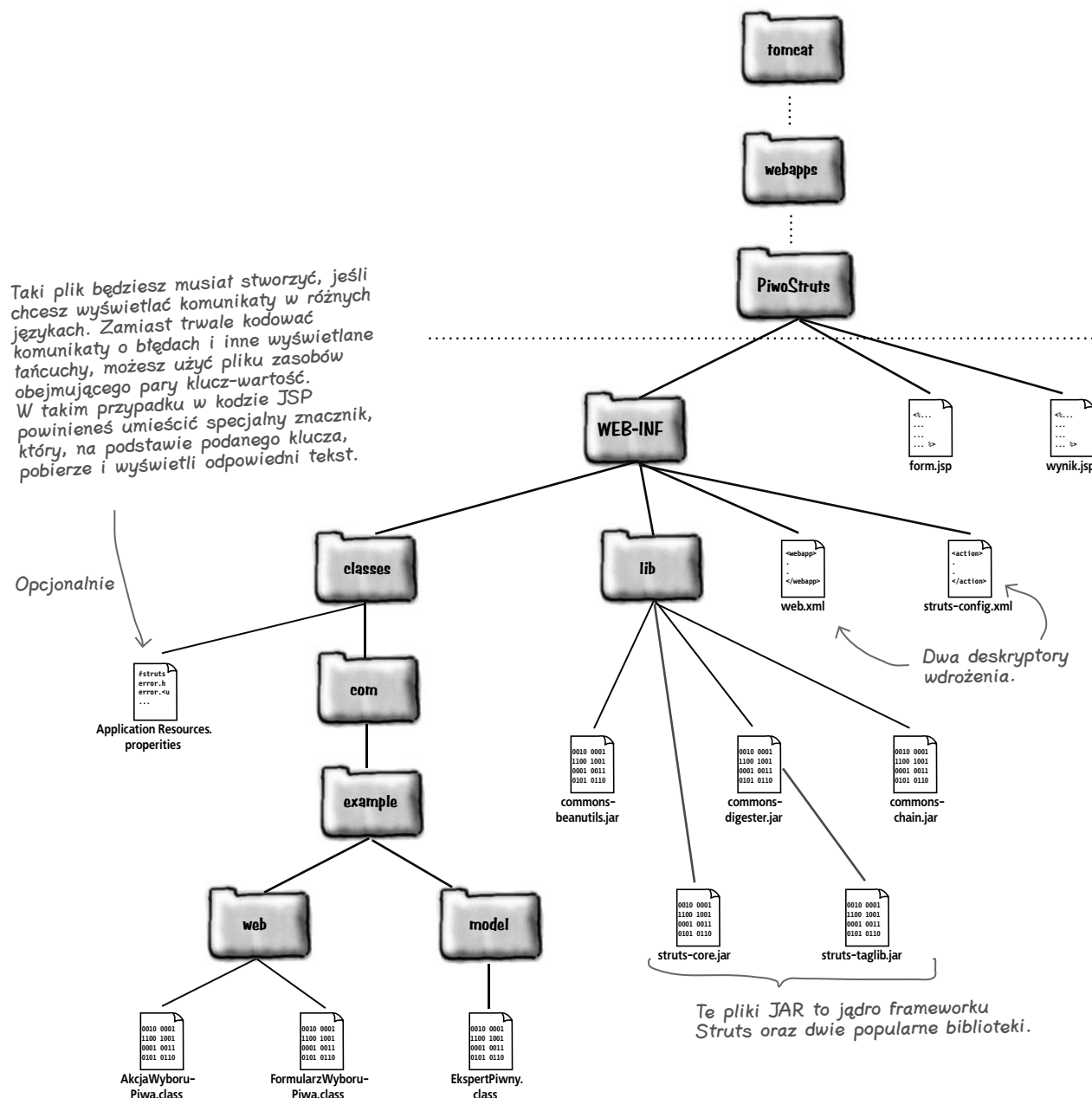
Łączy, jak i wersja Struts wymienione na tej stronie były aktualne w czasie tworzenia niniejszej książki. Nie będzie to miało dla Ciebie większego znaczenia, lecz chcemy po prostu zasygnalizować że: *nie mamy najmniejszego pojęcia, jaki będzie adres lub wersja Struts w chwili, gdy będziesz czytać niniejszą książkę, jednak staraliśmy się dostarczyć Ci najbardziej aktualne i precyzyjne informacje.*

## Sześć etapów instalacji Struts:

- ❶ W przeglądarce wyświetl stronę o adresie:  
**`http://struts.apache.org/downloads.html`**
- ❷ Na liście *General Availability* kliknij ostatnie łącze *Struts v1.3\**.
- ❸ Wybierz odpowiedni plik JAR. Najmniejszy z dostępnych plików zawiera samą bibliotekę:  
**`struts-1.3.8-lib.zip`**
- ❹ Pobierz i zapisz w katalogu tymczasowym wybrany plik zip.
- ❺ Rozpakuj pobrany plik; w efekcie powstanie następująca struktura katalogów i plików:  
**`struts-1.3.8/  
NOTICE.txt  
lib/  
struts-core-1.3.8.jar  
struts-taglib-1.3.8.jar  
commons-beanutils-1.7.0.jar  
commons-digester.jar  
commons-chain-1.1.jar`**
- ❻ Skopiuj wszystkie pięć plików JAR wymienionych w punkcie 5. do katalogu *WEB-INF/lib/* swojej aplikacji internetowej.
- ❼ Uwaga: Upewnij się, że podczas kompilacji komponentów formularzy oraz obiektów akcji w jednym z katalogów wskazanych na ścieżce klas będzie dostępna kopia pliku *struts-core-1.3.8.jar*. (Pamiętaj, że kontroler frontowy `ServletAction` jest tworzony automatycznie).

# Tworzenie środowiska wdrożeniowego

Na poniższym rysunku przedstawiliśmy strukturę katalogów, jaką musisz stworzyć, by uruchomić wersję aplikacji piwnej na bazie frameworku Struts.

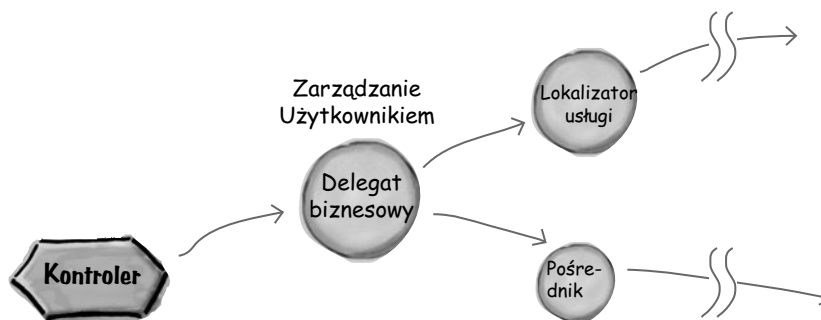


# Przegląd wzorców na potrzeby egzaminu SCWCD

W dwóch ostatnich rozdziałach przedstawiliśmy całkiem sporo wzorców projektowych. Na kilku kolejnych stronach zebraliśmy te szczegółowe informacje na ich temat, które będziesz zapewne chciał przestudiować, przygotowując się do egzaminu SCWCD.

## Business Delegate (delegat biznesowy)

**Wzorzec Business Delegate pełni funkcję swoistej tarczy chroniącej nasze kontrolery warstwy internetowej przed problemami wynikającymi ze zdalności części komponentów modelu aplikacji.**



### Cechy wzorca Business Delegate

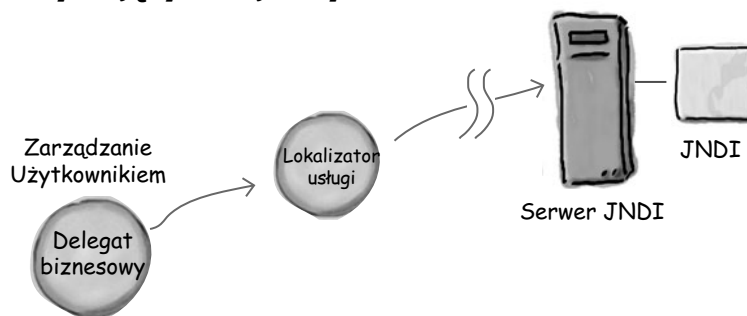
- Działa jako pośrednik, implementując interfejs usługi zdalnej.
- Inicjuje komunikację z usługą zdalną.
- Obsługuje szczegóły komunikacji oraz ewentualne wyjątki.
- Odbiera żądania przesyłane z komponentu kontrolera.
- Tłumaczy żądanie i przekazuje je do usługi biznesowej (używając w tym celu pośrednika).
- Tłumaczy odpowiedź i zwraca ją do komponentu kontrolera.
- Dzięki obsłudze operacji związanych z odszukiwaniem komponentu zdalnego oraz szczegółów wymiany informacji z tym komponentem zwiększa stopień spójności kontrolerów.

### Podstawowe zasady wzorca Business Delegate

- Wzorzec Business Delegate opiera się na następujących zasadach:
  - ukrywania złożoności,
  - kodowania według interfejsów,
  - niskiego stopnia powiązań,
  - separacji zadań.
- Wzorzec ten minimalizuje wpływ, jaki zmiany zachodzące w warstwie biznesowej wywierają na komponenty warstwy internetowej.
- Pozwala on na zmniejszenie stopnia powiązań pomiędzy poszczególnymi warstwami.
- Dodaje do aplikacji kolejną warstwę, przez co zwiększa jej złożoność.
- Metody tego wzorca powinny być możliwie ogólne, aby zmniejszyć natężenie ruchu sieciowego.

## Service Locator (lokalizator usługi)

**Wzorca Service Locator należy używać do wyszukiwania komponentów w rejestrze, aby uprościć strukturę wszystkich pozostałych komponentów (w tym delegatów biznesowych), które muszą korzystać z interfejsu JNDI (lub podobnych operacji wykorzystujących rejestry innych typów).**



### Cechy wzorca Service Locator



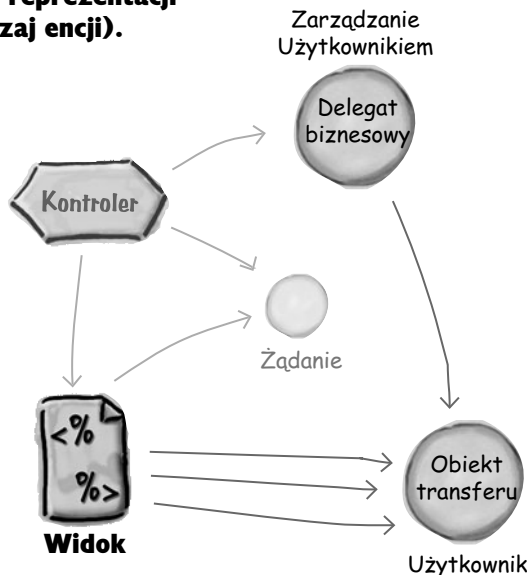
- Pobiera obiekty InitialContext.
- Przeszukuje rejestr.
- Obsługuje szczegóły komunikacji oraz wyjątki.
- Może poprawiać wydajność, przechowując pobrane wcześniej referencje w pamięci podręcznej.
- Może współpracować z wieloma różnymi rejestrami, takimi jak JNDI, RMI, UDDI, COS.

### Podstawowe zasady wzorca Service Locator

- Wzorec Service Locator bazuje na następujących zasadach:
  - ukrywania złożoności,
  - separacji zadań.
- Pozwala minimalizować wpływ ewentualnych zmian położenia komponentów zdalnych (w tym zmian ich macierzystych kontenerów) na funkcjonowanie warstwy internetowej.
- Przyczynia się do zmniejszenia stopnia powiązań pomiędzy warstwami.

## Transfer Object (obiekt transferu)

**Wzorca Transfer Object należy używać w celu minimalizacji ruchu sieciowego poprzez udostępnienie lokalnej reprezentacji komponentu zdalnego (zazwyczaj encji).**



### Cechy wzorca Transfer Object

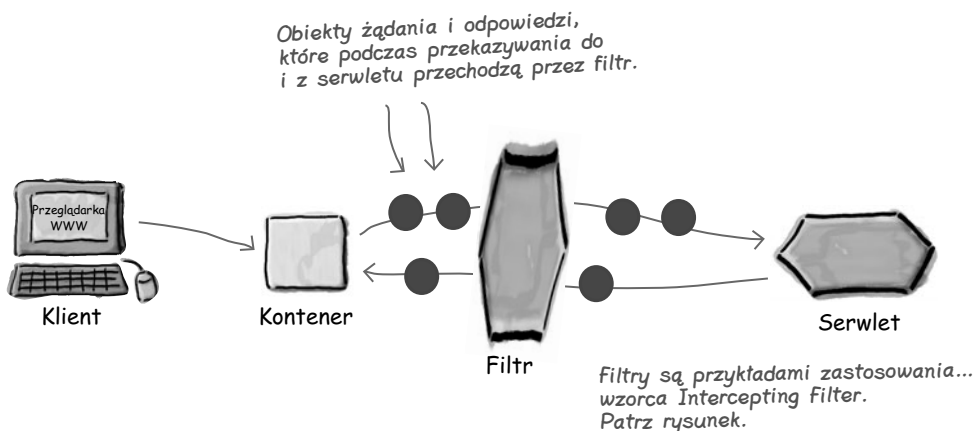
- Stanowi lokalną reprezentację zdalnego komponentu encji (czyli obiektu, który przechowuje jakiś stan bazy danych).
- Minimalizuje ruch sieciowy.
- Może być zgodny z konwencjami nazewniczymi technologii JavaBean, dzięki czemu inne komponenty z łatwością będą mogły z niego korzystać.
- Jest zaimplementowany jako obiekt, który można serializować, dzięki czemu można go przesyłać siecią.
- Zazwyczaj komponenty widoku mają do niego łatwy dostęp.

### Podstawowe zasady wzorca Transfer Object

- Wzorzec ten bazuje na zasadzie:
  - ograniczania ruchu sieciowego.
- Redukuje negatywny wpływ, jaki na wydajność warstwy internetowej mają szczegółowe odwołania do danych przechowywanych w zdalnym komponentcie.
- Przyczynia się do zmniejszenia stopnia powiązań pomiędzy warstwami.
- Wadą wzorca Transfer Object jest ryzyko udostępniania korzystającym z niego komponentom informacji nieaktualnych, ponieważ obiekt transferu w rzeczywistości stanowi tylko reprezentację informacji przechowywanych w zupełnie innym miejscu.
- Zapewnianie możliwości aktualizacji obiektów transferu z zachowaniem bezpieczeństwa przetwarzania współbieżnego zwykle jest dość trudne.

# Intercepting Filter (filtr przechwytyjący)

**Wzorca Intercepting Filter można używać do modyfikowania żądań przekazywanych do serwletów lub odpowiedzi odsyłanych do użytkownika.**



## Cechy wzorca Intercepting Filter

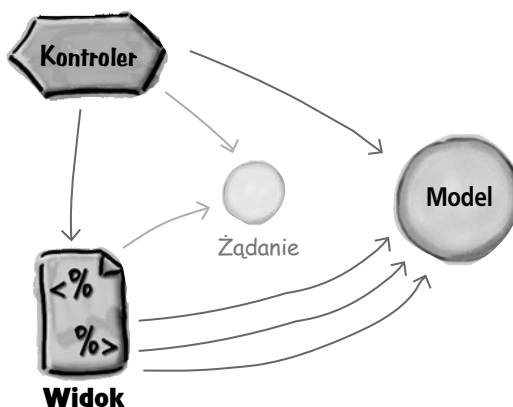
- Może przechwytywać (oraz modyfikować) żądania, zanim trafią do serwletu.
- Może przechwytywać (oraz modyfikować) odpowiedzi, zanim zostaną przesłane do klienta.
- Filtry są wdrażane w sposób deklaracyjny przy wykorzystaniu deskryptora wdrożenia.
- Filtry są modułarne i mogą być łączone w formie łańcucha i wykonywane kolejno jeden po drugim.
- Filtry posiadają swój cykl istnienia, którym zarządza kontener.
- Filtry muszą implementować metody zwrotne kontenera.

## Podstawowe zasady wzorca Intercepting Filter

- Wzorec Intercepting Filter bazuje na:
  - zwiększaniu stopnia spójności,
  - zmniejszaniu stopnia powiązań,
  - zwiększaniu kontroli deklaratywnej.
- Dzięki kontroli deklaratywnej filtry z łatwością mogą być stosowane bądź to na stałe, bądź jedynie czasowo.
- Dzięki kontroli deklaratywnej z łatwością można zmieniać kolejność, w jakiej są wykonywane poszczególne filtry w łańcuchu.

## Model, View, Controller (MVC)

**Wzorzec MVC jest stosowany do tworzenia logicznej struktury rozdzielającej komponenty tworzące aplikację na trzy podstawowe warstwy (Modelu, Widoku oraz Kontrolera). W ten sposób uzyskujemy większy stopień spójności poszczególnych komponentów i zwiększamy możliwości ich wielokrotnego wykorzystywania (w szczególności dotyczy to komponentów modelu).**



### Cechy wzorca MVC

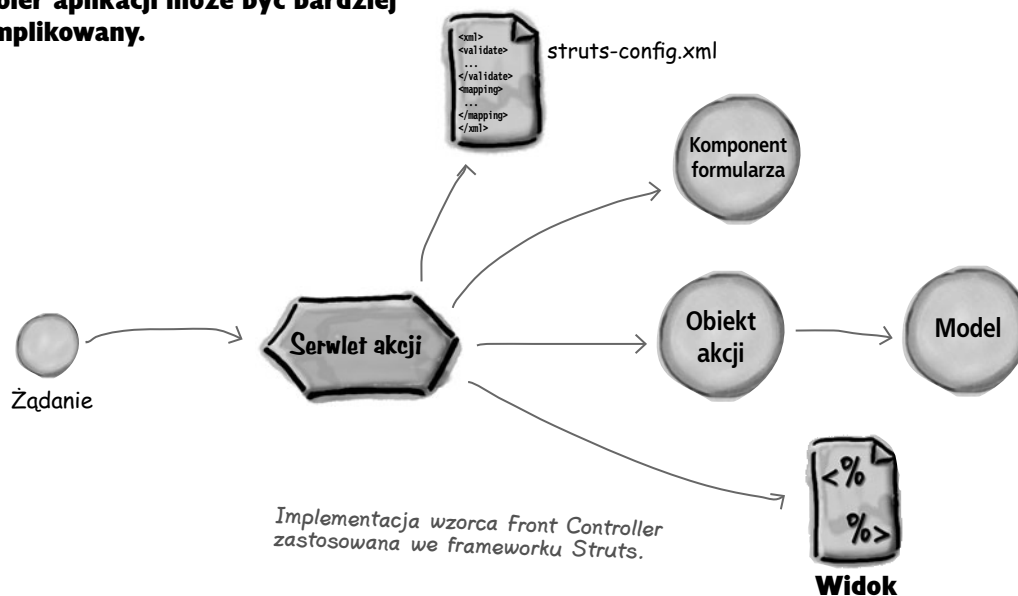
- Widoki mogą się zmieniać niezależnie od stosowanych kontrolerów i modeli.
- Szczegóły (struktury danych) komponentów modelu są niewidoczne dla komponentów widoku i kontrolera.
- Jeśli model jest zgodny ze ściśle określonym kontraktem (interfejsem), jego komponenty można wykorzystywać także w innych rodzajach aplikacji, takich jak aplikacje z graficznym interfejsem użytkownika (GUI) czy aplikacje J2ME.
- Oddzielenie kodu modelu od kontrolera znacznie ułatwia przyszłe przystosowanie aplikacji do współpracy ze zdalnymi komponentami biznesowymi.

### Podstawowe zasady wzorca MVC

- Wzorzec MVC bazuje na zasadach:
  - separacji zadań,
  - zmniejszenia stopnia powiązań.
- Wzorzec ten zwiększa stopień spójności poszczególnych komponentów.
- Zwiększa ogólną złożoność aplikacji. (To fakt, gdyż zastosowanie tego wzorca powoduje dodanie do aplikacji wielu nowych komponentów, choć stopień spójności poszczególnych komponentów jest większy).
- Jego zastosowanie minimalizuje wpływ, jaki zmiany wprowadzane w jednej warstwie wywierają na pozostałe warstwy aplikacji.

## Front Controller (kontroler frontowy)

**Wzorzec Front Controller można wykorzystać w celu umieszczenia wspólnego, często powtarzanego kodu przetwarzającego żądania w jednym komponencie. Dzięki temu kontroler aplikacji może być bardziej spójny i mniej skomplikowany.**



### Cechy wzorca Front Controller

- Gromadzi wszystkie czynności związane z początkowym przetwarzaniem żądań w jednym komponencie.
- Wzorzec Front Controller stosowany łącznie z innymi wzorcami projektowymi pozwala zmniejszyć stopień powiązań poprzez umożliwienie deklaratywnego przekazywania żądań do warstwy prezentacji.
- W porównaniu z frameworkiem Struts sam wzorzec Front Controller jest stosunkowo ubogi. Stworzenie sensownej aplikacji przy wykorzystaniu tego wzorca wymagałoby samodzielnego zaimplementowania wielu rozwiązań, których gotowe wersje są oferowane w ramach frameworku Struts.

### Podstawowe zasady wzorca Front Controller

- Wzorzec Front Controller bazuje na zasadach:
  - ukrywania złożoności,
  - separacji zadań,
  - zmniejszenie stopnia powiązań.
- Jego zastosowanie przyczynia się do zwiększenia stopnia spójności komponentów kontrolera.
- Zmniejszenie ogólnej złożoności aplikacji.
- Uproszczenie utrzymania kodu infrastruktury.



**BAR  
KAWOWY**

*Egzamin próbny*

---

**1** Na podstawie następującej listy cech:

- związany ze wzorcem Intercepting Filter (filtra przechwytyjącego),
- wspomaga separację zadań wykonywanych przez różnych programistów,
- wspomaga możliwości wielokrotnego stosowania komponentów,

wskaż wzorec projektowy, który opisują powyższe cechy.

- ☐ A. Transfer Object (obiekt transferu)
- ☐ B. Service Locator (lokalizator usługi)
- ☐ C. Front Controller (kontroler frontowy)
- ☐ D. Business Delegate (delegat biznesowy)

---

**2** Projekt tworzonej aplikacji internetowej wymaga zastosowania pewnych mechanizmów zabezpieczeń, które będą wykorzystywane podczas obsługi wszystkich odbieranych żądań. Niektóre z testów warunków będą wykonywane zawsze, niezależnie od typu żądania.

Który ze wymienionych wzorców można zastosować, by zaspokoić te wymagania projektowe?

- ☐ A. Transfer Object (obiekt transferu)
- ☐ B. Service Locator (lokalizator usługi)
- ☐ C. Composite Entity (jednostka złożona)
- ☐ D. Business Delegate (delegat biznesowy)
- ☐ E. Intercepting Filter (filtr przechwytyjący)

---

**3** Twoja firma chciałaby zwiększyć moc swojej aplikacji rozproszonej. Twoim zadaniem jest jak najprostsza integracja usług internetowych aplikacji z firmowymi obiektami dostępu do danych (DAO). Co więcej, Twoje ogólne Lokalizatory kontrolera muszą zostać wzbogacone o obsługę rejestrów UDDI stosowanych w technologii J2ME.

Który ze wymienionych wzorców można zastosować, by spełnić te wymagania projektowe?

- ☐ A. Domain Activator (aktywator domeny)
- ☐ B. Intercepting Observer (obserwator przechwytyjący)
- ☐ C. Composite Delegate (delegat złożony)
- ☐ D. Transfer Facade (fasada transferu)

- 
- 4** Poniższe stwierdzenia opisują potencjalne korzyści jakie niesie ze sobą stosowanie pewnego wzorca projektowego.

Ten wzorec ogranicza ilość transmisji sieciowych wykonywanych pomiędzy klientem a komponentem korporacyjnym i pozwala klientowi na pobranie lokalnej kopii zawartości komponentu korporacyjnego poprzez wywołanie tylko jednej metody komponentu korporacyjnego.

Do którego z poniższych wzorców odnoszą się te stwierdzenia?

- ☐ A. Transfer Object (obiekt transferu)
  - ☐ B. Intercepting Filter (filtr przechwytyjący)
  - ☐ C. Model-View-Controller (model-widok-kontroler)
  - ☐ D. Business Delegate (delegat biznesowy)
- 

- 5** Twoja firma — Modele UA — pracuje nad stworzeniem zaawansowanego komponentu do obsługi spisu magazynu, który mógłby być stosowany we wszystkich najpopularniejszych kontenerach J2EE. Twoim zadaniem jest zaprojektowanie fragmentu tego komponentu odpowiedzialnego za wyszukiwanie JNDI niezależnie od oprogramowania używanego przez klienta.

Który z poniższych wzorców projektowych może Ci pomóc w zrealizowaniu zadania?

- ☐ A. Transfer Object (obiekt transferu)
  - ☐ B. Intercepting Filter (filtr przechwytyjący)
  - ☐ C. Model-View-Controller (model-widok-kontroler)
  - ☐ D. Business Delegate (delegat biznesowy)
  - ☐ E. Service Locator (lokalizator usługi)
- 

- 6** Podczas dopracowywania swojej wielowarstwowej biznesowej aplikacji J2EE zauważyłeś, że mógłbyś poprawić jej wydajność, ograniczając liczbę zdalnych żądań wykonywanych przez aplikację i jednocześnie zwiększając ilości danych pobieranych w trakcie realizacji każdego z tych żądań.

Który z poniższych wzorców projektowych mógłbyś wykorzystać podczas wprowadzania tych zmian w aplikacji?

- ☐ A. Transfer Object (obiekt transferu)
- ☐ B. Service Locator (lokalizator usługi)
- ☐ C. Front Controller (kontroler frontowy)
- ☐ D. Intercepting Filter (filtr przechwytyjący)
- ☐ E. Model-View-Controller (model-widok-kontroler)

---

**7** Wskaż wzorec projektowy charakteryzowany przez poniższą listę cech:

- związany ze wzorcem Service Locator,
  - zmniejsza stopień powiązań,
  - może powodować dodanie do aplikacji kolejnej warstwy oraz nieznaczne zwiększenie jej złożoności.
- ☐ A. Transfer Object (obiekt transferu)
- ☐ B. Front Controller (kontroler frontowy)
- ☐ C. Business Delegate (delegat biznesowy)
- ☐ D. Intercepting Filter (filtr przechwytyjący)
- ☐ E. Model-View-Controller (model-widok-kontroler)

---

**8** Twoja aplikacja internetowa pracująca w środowisku rozproszonym wykorzystuje komponent `SessionBean` do wykonywania złożonych obliczeń, takich jak weryfikacja numerów kart kredytowych. Niemniej jednak chciałbyś odizolować komponenty swojej aplikacji od kodu używanego do odszukania komponentu `SessionBean` oraz do posługiwania się jego interfejsem. Chciałbyś oddzielić klasy aplikacji lokalnej od odszukiwania i stosowania komponentu zdalnego, którego interfejs może się zmieniać. Który ze wzorców projektowych J2EE zastosowałbyś w takiej sytuacji?

- ☐ A. Transfer Object (obiekt transferu)
- ☐ B. Service Locator (lokalizator usługi)
- ☐ C. Model-View-Controller (model-widok-kontroler)
- ☐ D. Business Delegate (delegat biznesowy)

---

**9** Wskaż wzorec projektowy charakteryzowany przez poniższą listę cech:

- związany ze wzorcem Business Delegate,
  - poprawia wydajność operacji sieciowych,
  - może poprawić wydajność działania klienta poprzez wykorzystanie pamięci podręcznej.
- ☐ A. Transfer Object (obiekt transferu)
- ☐ B. Service Locator (lokalizator usługi)
- ☐ C. Front Controller (kontroler frontowy)
- ☐ D. Intercepting Filter (filtr przechwytyjący)
- ☐ E. Model-View-Controller (model-widok-kontroler)



## Egzamin próbny — odpowiedzi

1 Na podstawie następującej listy cech:

- związany ze wzorcem Intercepting Filter (filtra przechwytyjącego),
- wspomaga separację zadań wykonywanych przez różnych programistów,
- wspomaga możliwości wielokrotnego stosowania komponentów,

wskaż wzorec projektowy, który opisują powyższe cechy.

- ☐ A. Transfer Object (obiekt transferu)
- ☐ B. Service Locator (lokalizator usługi)
- ☒ C. Front Controller (kontroler frontowy)
- ☐ D. Business Delegate (delegat biznesowy)

(J2EE. Wzorce projektowe.  
Wydanie drugie, str. 157 ).

– Ten wzorec (jak kilka innych) pomaga izolować zadania wykonywane przez programistów od zadań realizowanych przez projektantów stron WWW.

2 Projekt tworzonej aplikacji internetowej wymaga zastosowania pewnych mechanizmów zabezpieczeń, które będą wykorzystywane podczas obsługi wszystkich odbieranych żądań. Niektóre z testów warunków będą wykonywane zawsze, niezależnie od typu żądania.

Który ze wymienionych wzorców można zastosować, by zaspokoić te wymagania projektowe?

- ☐ A. Transfer Object (obiekt transferu)
- ☐ B. Service Locator (lokalizator usługi)
- ☐ C. Composite Entity (jednostka złożona)
- ☐ D. Business Delegate (delegat biznesowy)
- ☒ E. Intercepting Filter (filtr przechwytyjący)

(J2EE. Wzorce projektowe.  
Wydanie drugie, str. 127).

– Filtr przechwytyjący doskonale nadaje się do wykorzystania w sytuacjach, gdy chcemy przechwycić i zmodyfikować żądanie przed przystąpieniem do jego standardowej obsługi.

3 Twoja firma chciałaby zwiększyć moc swojej aplikacji rozproszonej. Twoim zadaniem jest jak najprostsza integracja usług internetowych aplikacji z firmowymi obiektami dostępu do danych (DAO). Co więcej, Twoje ogólne Lokalizatory kontrolera muszą zostać wzbogacone o obsługę rejestrów UDDI stosowanych w technologii J2ME.

Który ze wymienionych wzorców można zastosować, by spełnić te wymagania projektowe?

- ☐ A. Domain Activator (aktywator domeny)
- ☐ B. Intercepting Observer (obserwator przechwytyjący)
- ☒ C. Composite Delegate (delegat złożony)
- ☐ D. Transfer Facade (fasada transferu)

(Dating Design  
Patterns, rozdz. 7.).

– Zważywszy na nieregularność wymagań, wzorec Composite Delegate powinien nam zapewnić najwięcej swobody w procesie wprowadzania dalszych modyfikacji. :)

- 4 Poniższe stwierdzenia opisują potencjalne korzyści jakie niesie ze sobą stosowanie pewnego wzorca projektowego. (J2EE. Wzorce projektowe. Wydanie drugie, str. 354).

Ten wzorec ogranicza ilość transmisji sieciowych wykonywanych pomiędzy klientem a komponentem korporacyjnym i pozwala klientowi na pobranie lokalnej kopii zawartości komponentu korporacyjnego poprzez wywołanie tylko jednej metody komponentu korporacyjnego.

Do którego z poniższych wzorców odnoszą się te stwierdzenia?

- ☒ A. Transfer Object (obiekt transferu) – Największą zaletą wzorca Transfer Object jest mniejsze obciążenie sieci.
- ☐ B. Intercepting Filter (filtr przechwytyjący)
- ☐ C. Model-View-Controller (model-widok-kontroler)
- ☐ D. Business Delegate (delegat biznesowy)

- 5 Twoja firma — Modele UA — pracuje nad stworzeniem zaawansowanego komponentu do obsługi spisu magazynu, który mógłby być stosowany we wszystkich najpopularniejszych kontenerach J2EE. Twoim zadaniem jest zaprojektowanie fragmentu tego komponentu odpowiedzialnego za wyszukiwanie JNDI niezależnie od oprogramowania używanego przez klienta. (J2EE. Wzorce projektowe. Wydanie drugie, str. 266).

Który z poniższych wzorców projektowych może Ci pomóc w zrealizowaniu zadania?

- ☐ A. Transfer Object (obiekt transferu)
- ☐ B. Intercepting Filter (filtr przechwytyjący)
- ☐ C. Model-View-Controller (model-widok-kontroler)
- ☐ D. Business Delegate (delegat biznesowy)
- ☒ E. Service Locator (lokalizator usługi) – Wzorca Service Locator można używać w sytuacji, gdy chcemy odizolować nasz kod od wszelkich aspektów wykorzystania usługi wyszukiwania charakterystycznych dla konkretnego producenta. Wzorec ten pozwala na wydzielenie kodu, który będzie się zmieniał w zależności od używanej usługi.

- 6 Podczas dopracowywania swojej wielowarstwowej biznesowej aplikacji J2EE zauważyłeś, że mógłbyś poprawić jej wydajność, ograniczając liczbę zdalnych żądań wykonywanych przez aplikację i jednocześnie zwiększając ilości danych pobieranych w trakcie realizacji każdego z tych żądań. (J2EE. Wzorce projektowe. Wydanie drugie, str. 346 – 347).

Który z poniższych wzorców projektowych mógłbyś wykorzystać podczas wprowadzania tych zmian w aplikacji?

- ☒ A. Transfer Object (obiekt transferu)
- ☐ B. Service Locator (lokalizator usługi)
- ☐ C. Front Controller (kontroler frontowy)
- ☐ D. Intercepting Filter (filtr przechwytyjący)
- ☐ E. Model-View-Controller (model-widok-kontroler) – Wzorec Transfer Object można wykorzystać do połączenia wielu szczegółowych wywołań w jedno wywołanie o bardziej ogólnym charakterze. Bardzo często ograniczenie ruchu sieciowego i tak przyczynia się do zwiększenia wydajności, pomimo konieczności zapisywania danych i przesyłania większych obiektów.

7 Wskaż wzorec projektowy charakteryzowany przez poniższą listę cech:

(J2EE. Wzorce projektowe. Wydanie drugie, str. 260 – 262).

- związany ze wzorcem Service Locator,
- zmniejsza stopień powiązań,
- może powodować dodanie do aplikacji kolejnej warstwy oraz nieznaczne zwiększenie jej złożoności.

- ☐ A. Transfer Object (obiekt transferu)
- ☐ B. Front Controller (kontroler frontowy)
- ☒ C. Business Delegate (delegat biznesowy)
- ☐ D. Intercepting Filter (filtr przechwytyjący)
- ☐ E. Model-View-Controller (model-widok-kontroler)

– Choć zastosowanie tego wzorca wiąże się z dodaniem do aplikacji kolejnej warstwy, to jednak jego zalety (takie jak zmniejszenie powiązań i uproszczenie interfejsu warstwy biznesowej) sprawiają, iż i tak jest to korzystne.

8 Twoja aplikacja internetowa pracująca w środowisku rozproszonym wykorzystuje komponent SessionBean do wykonywania złożonych obliczeń, takich jak weryfikacja numerów kart kredytowych. Niemniej jednak chciałbyś odizolować komponenty swojej aplikacji od kodu używanego do odszukania komponentu SessionBean oraz do posługiwania się jego interfejsem. Chciałbyś oddzielić klasy aplikacji lokalnej od odszukiwania i stosowania komponentu zdalnego, którego interfejs może się zmieniać. Który ze wzorców projektowych J2EE zastosowałbyś w takiej sytuacji?

(J2EE. Wzorce projektowe. Wydanie drugie, str. 260).

- ☐ A. Transfer Object (obiekt transferu)
- ☐ B. Service Locator (lokalizator usługi)
- ☐ C. Model-View-Controller (model-widok-kontroler)
- ☒ D. Business Delegate (delegat biznesowy)

– Najważniejszą zaletą wzorca Business Delegate jest zmniejszenie stopnia powiązań pomiędzy warstwą prezentacji a warstwą biznesową.

9 Wskaż wzorec projektowy charakteryzowany przez poniższą listę cech:

(J2EE. Wzorce projektowe. Wydanie drugie, str. 274 – 276).

- związany ze wzorcem Business Delegate,
- poprawia wydajność operacji sieciowych,
- może poprawić wydajność działania klienta poprzez wykorzystanie pamięci podręcznej.

- ☐ A. Transfer Object (obiekt transferu)
- ☒ B. Service Locator (lokalizator usługi)
- ☐ C. Front Controller (kontroler frontowy)
- ☐ D. Intercepting Filter (filtr przechwytyjący)
- ☐ E. Model-View-Controller (model-widok-kontroler)

– Dzięki zastosowaniu tego wzorca można potążyć wywołania sieciowe konieczne do odszukania komponentu oraz utworzenia obiektu biznesowego i zapisania w nim danych.



# Dodatek A

## Końcowy egzamin próbny



**BAR  
KAWOWY**

NIE próbuj rozwiązywać zadań tego egzaminu, dopóki nie uznasz, że jesteś gotów, by zdawać prawdziwy egzamin. Jeśli spróbujesz rozwiązać je zbyt wcześnie, to podczas kolejnej próby będziesz już pamiętał niektóre pytania, co mogłoby w sztuczny sposób zawyżyć uzyskany wynik. Naprawdę chcemy, abyś zdał egzamin już za *pierwszym* razem. (Chyba że istnieje jakaś metoda przekonania Cię, byś kupował nowy egzemplarz tej książki za każdym razem, gdy będziesz próbować zdać ten egzamin).

Aby nieco zmniejszyć wagę problemu „pamiętam to pytanie”, pytania przedstawione w naszym egzaminie są nieco *trudniejsze* od tych, które można spotkać na egzaminie prawdziwym, gdyż *nie* podajemy informacji o tym, ile odpowiedzi do każdego pytania jest poprawnych. Nasze pytania i odpowiedzi są niemal identyczne pod względem stylu, stopnia trudności oraz zagadnień z pytaniami pojawiającymi się na prawdziwym egzaminie. Z drugiej strony, ponieważ nie podajemy liczby poprawnych odpowiedzi, nie możesz z góry odrzucić żadnej z nich. Zdajemy sobie sprawę, że to okrutne z naszej strony, lecz chcielibyśmy powiedzieć Ci, iż świadomość tego boli nas bardziej niż Tobie doskwiera konieczność zdawania egzaminu w taki sposób. (Ale i tak możesz być wdzięczny losowi, gdyż jeszcze kilka lat temu prawdziwe egzaminy firmy Sun były przeprowadzane właśnie w taki sposób, a niemal wszystkie pytania kończyły się zdaniem: „Należy zaznaczyć wszystkie poprawne opcje”).

Większość osób stwierdziło, że nasz egzamin próbny faktycznie *jest* nieco trudniejszy od prawdziwego egzaminu SCWCD, jednak wyniki uzyskane na obu egzaminach były bardzo podobne. Przedstawiony egzamin próbny będzie doskonałym sposobem sprawdzenia, czy faktycznie jesteś gotowy do przystąpienia do prawdziwego egzaminu, jednak wyłącznie w przypadku, gdy:

- 1) Poświęcisz na jego rozwiązanie nie więcej niż dwie godziny i 15 minut (czyli tyle, ile trwa prawdziwy egzamin).
- 2) Podczas rozwiązywania egzaminu nie będziesz zaglądał do żadnej literatury (w tym także do tej książki).
- 3) Nie będziesz kilkakrotnie próbował zdawać tego egzaminu. Rozwiązując go po raz czwarty, możesz uzyskać 98 procent poprawnych odpowiedzi, jednak wciąż możesz nie być gotowy do zdawania prawdziwego egzaminu, gdyż będziesz pamiętał nasze pytania i odpowiedzi do nich.
- 4) Wszelkie alkohole i inne substancje wpływające na działanie mózgu zaczniesz przyjmować w dużych ilościach dopiero *po* zdaniu egzaminu...



1 Programista prawidłowo skonfigurował strukturę katalogów dla swojej aplikacji internetowej w technologii Java EE. Aplikacja nosi nazwę `MojaAplikacjaInternetowa`. W których katalogach można umieścić plik `mojZnacznik.tag`, aby był dostępny dla kontenera? (Zaznacz wszystkie odpowiedzi).

- ☐ A. `MojaAplikacjaInternetowa/WEB-INF`
- ☐ B. `MojaAplikacjaInternetowa/META-INF`
- ☐ C. `MojaAplikacjaInternetowa/WEB-INF/lib`
- ☐ D. `MojaAplikacjaInternetowa/WEB-INF/tags`
- ☐ E. `MojaAplikacjaInternetowa/WEB-INF/TLDs`
- ☐ F. `MojaAplikacjaInternetowa/WEB-INF/tags/mojeZnaczniki`

2 Które z poniższych wyrażeń są prawidłowymi konstrukcjami języka EL? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. `${"1" + "2"}`
- ☐ B. `${1 plus 2}`
- ☐ C. `${1 eq 2}`
- ☐ D. `${2 div 1}`
- ☐ E. `${2 & 1}`
- ☐ F. `${"head"+"first"}`

- 3 Plik TLD opracowany z myślą o witrynie internetowej z forum dyskusyjnym poświęconym Javie zawiera następującą definicję znacznika:

```
<tag>
 <name>avatar</name>
 <tag-class>hf.AvatarTagHandler</tag-class>
 <body-content>empty</body-content>

 <attribute>
 <name>userId</name>
 <required>true</required>
 <rtexprvalue>true</rtexprvalue>
 </attribute>
 <attribute>
 <name>size</name>
 <required>false</required>
 <rtexprvalue>false</rtexprvalue>
 </attribute>
</tag>
```

Które zdania na temat klasy **AvatarTagHandler** są prawdziwe, jeśli przyjąć, że klasa ta rozszerza klasę **SimpleTagHandler** i że generuje kod HTML-a wyświetlający obraz awatara użytkownika? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. Klasa **AvatarTagHandler** powinna definiować składową **size**, dla której musi istnieć przynajmniej metoda ustawiająca.
- ☐ B. Zmienna **size** nie jest konieczna, ponieważ plik TLD mówi wyraźnie, że odpowiedni atrybut ma charakter opcjonalny.
- ☐ C. Klasa **AvatarTagHandler** musi nadpisywać metodę cyklu życia **doTag**.
- ☐ D. Klasa **AvatarTagHandler** musi nadpisywać metodę cyklu życia **doStartTag**.
- ☐ E. Klasa **AvatarTagHandler** musi przeciążać wszystkie implementowane metody cyklu życia wersjami otrzymującymi po jednym dodatkowym parametrze dla każdego atrybutu zdefiniowanego w pliku TLD. W tym przypadku będzie konieczny tylko jeden taki parametr.

- 4** Serwlet tworzy komponent i ustawia jego właściwości przed przekazaniem żądania do strony JSP.

Poniżej przedstawiono odpowiedni fragment kodu tego serwletu:

```
20. foo.User user = new foo.User();
21. user.setFirst(request.getParameter("firstName"));
22. user.setLast(request.getParameter("lastName"));
23. user.setStreet(request.getParameter("streetAddress"));
24. user.setCity(request.getParameter("city"));
25. user.setState(request.getParameter("state"));
26. user.setZipCode(request.getParameter("zipCode"));
27. request.setAttribute("user", user);
```

Który z poniższych fragmentów kodu umieszczony w kodzie strony JSP mógłby skutecznie zastąpić przedstawiony kod serwletu? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. `<jsp:useBean id="user" type="foo.User" scope="request"/>`
- ☐ B. `<jsp:useBean id="user" type="foo.User" scope="request">`  
`<jsp:setProperty name="user" property="*/>`  
`</jsp:useBean>`
- ☐ C. `<jsp:useBean id="user" class="foo.User" scope="request">`  
`<jsp:setProperty name="user" property="first" param="firstName"/>`  
`<jsp:setProperty name="user" property="last" param="lastName"/>`  
`<jsp:setProperty name="user" property="street" param="streetAddress"/>`  
`<jsp:setProperty name="user" property="city"/>`  
`<jsp:setProperty name="user" property="state"/>`  
`<jsp:setProperty name="user" property="zipCode"/>`  
`</jsp:useBean>`
- ☐ D. `<jsp:useBean id="user" class="foo.User" scope="request">`  
`<jsp:setProperty name="user" property="*/>`  
`<jsp:setProperty name="user" property="first" param="firstName"/>`  
`<jsp:setProperty name="user" property="last" param="lastName"/>`  
`<jsp:setProperty name="user" property="street" param="streetAddress"/>`  
`</jsp:useBean>`

- 5 Które zdania opisujące korzyści, ograniczenia i zastosowania obiektu delegata biznesowego i obiektu lokalizatora usługi są prawdziwe? (Zaznacz wszystkie prawidłowe odpowiedzi).
- ☐ A. Oba obiekty równie często wykonują wywołania sieciowe.
  - ☐ B. Oba obiekty równie często wywołują metody obiektu transferu.
  - ☐ C. Oba obiekty są równie często wywoływane przez obiekt kontrolera.
  - ☐ D. Z perspektywy delegata biznesowego obiekt lokalizatora pełni funkcję serwera.
  - ☐ E. Jeśli oba obiekty zaimplementujemy z wykorzystaniem pamięci podręcznej, brak aktualizacji danych będzie poważniejszym problemem w przypadku delegata biznesowego.

- 6 Które zdania o procesie tworzenia obiektów nasłuchujących sesji są prawdziwe? (Zaznacz wszystkie prawidłowe odpowiedzi).
- ☐ A. Wszystkie te obiekty są deklarowane w deskrytorze wdrożenia.
  - ☐ B. Nie wszystkie obiekty tego typu muszą być deklarowane w deskrytorze wdrożenia.
  - ☐ C. Do ich deklarowania w deskrytorze wdrożenia służy znacznik `<listener>`.
  - ☐ D. Do ich deklarowania w deskrytorze wdrożenia służy znacznik `<session-listener>`.
  - ☐ E. Do ich deklarowania w deskrytorze wdrożenia służy znacznik umieszczany w znaczniku `<web-app>`.
  - ☐ F. Do ich deklarowania w deskrytorze wdrożenia służy znacznik umieszczany w znaczniku `<servlet>`.

- 7 Niektórzy użytkownicy skarżą się na dziwne zdarzenia, które mają miejsce, kiedy na jednym komputerze dwa okna przeglądarki internetowej jednocześnie uzyskują dostęp do tej samej aplikacji internetowej. Chcesz przetestować różne przeglądarki pod kątem możliwości współdzielenia sesji przez wiele okien. Decydujesz się na wypisanie na stronie JSP identyfikatora `JSESSIONID`. Jak można osiągnąć ten cel, zakładając, że w testowanej przeglądarce włączono obsługę cookies? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. `${cookie.JSESSIONID}`
- ☐ B. `${cookie.JSESSIONID.value}`
- ☐ C. `${cookie["JSESSIONID"] ["value"]}`
- ☐ D. `${cookie.JSESSIONID["value"]}`
- ☐ E. `${cookie["JSESSIONID"].value}`
- ☐ F. `${cookieValues[0].value}`

8 Który obiekt domyślny zapewnia dostęp do atrybutów obiektu **ServletContext**?

- ☐ A. **server**
  - ☐ B. **context**
  - ☐ C. **request**
  - ☐ D. **application**
  - ☐ E. **servletContext**
- 

9 Które z poniższych metod zdefiniowano w klasie **HttpServlet**?  
(Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. **doGet**
  - ☐ B. **doTrace**
  - ☐ C. **doError**
  - ☐ D. **doConnect**
  - ☐ E. **doOptions**
- 

10 Doszedłeś do wniosku, że niektóre funkcje Twojej aplikacji internetowej będą wymagały od użytkowników przejścia przez proces rejestracji. Co więcej, Twoja aplikacja musi operować na poufnych danych użytkowników, które należy chronić.

Które zdania są prawdziwe? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. Możesz zabezpieczać przesyłane dane dopiero po tym, jak Twoja aplikacja zweryfikuje hasło użytkownika.
- ☐ B. Spośród rozmaitych metod uwierzytelniania obsługiwanych przez kontenery zgodne ze specyfikacją Java EE tylko techniki **BASIC**, **DIGEST** i **FORM** zaimplementowano przez dopasowywanie nazwy i hasła użytkownika.
- ☐ C. Niezależnie od typu stosowanego mechanizmu uwierzytelniania Javy EE, jego aktywacja następuje dopiero w reakcji na żądanie chronionego zasobu.
- ☐ D. Wszystkie metody uwierzytelniania specyfikacji Java EE zapewniają bezpieczeństwo danych bez konieczności samodzielnego implementowania jakichkolwiek dodatkowych zabezpieczeń.

- 11 Dysponujemy następującymi fragmentami kodu zaczerpniętymi z pojedynczego znacznika w ramach deskryptora wdrożenia zgodnego ze specyfikacją Java EE:

```
343. <web-resource-collection>
344. <web-resource-name>Przepisy</web-resource-name>
345. <url-pattern>/Piwo/Aktualizuj/*</url-pattern>
346. <http-method>POST</http-method>
347. </web-resource-collection>
...
367. <auth-constraint>
368. <role-name>Członek</role-name>
369. </auth-constraint>
...
385. <user-data-constraint>
386. <transport-guarantee>CONFIDENTIAL</transport-guarantee>
387. </user-data-constraint>
```

Które zdania są prawdziwe? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. Deskryptor wdrożenia zgodny ze specyfikacją Java EE może zawierać pojedynczy znacznik, w którym mogą się znajdować wszystkie te znaczniki.
- ☐ B. Przedstawiony powyżej pojedynczy znacznik może zawierać więcej znaczników **<auth-constraint>**.
- ☐ C. Przedstawiony powyżej pojedynczy znacznik może zawierać więcej znaczników **<user-data-constraint>**.
- ☐ D. Pojedynczy znacznik **<web-resource-collection>** może zawierać więcej znaczników **<url-pattern>**.
- ☐ E. Inne znaczniki tego samego typu co powyższy (niewidoczny) znacznik mogą zawierać identyczny znacznik **<url-pattern>**.
- ☐ F. Przedstawiony powyżej znacznik deklaruje zasady uwierzytelniania, autoryzacji i integralności danych na potrzeby odpowiedniej aplikacji internetowej.

- 12** Pracujesz nad dokumentem JSP, który ma generować dynamiczny obraz SVG reprezentowany przez strukturę dokumentu w formacie XML. Twój dokument JSP musi deklarować nagłówek odpowiedzi HTTP '**Content-Type**' jako '**image/svg+xml**', aby przeglądarka internetowa właściwie wyświetliła odpowiedź w formie obrazu SVG.

Który fragment kodu strony JSP deklaruje, że dany dokument JSP jest odpowiedzią SVG?

- ☐ A. `<%@ page contentType='image/svg+xml' %>`
- ☐ B. `<jsp:page contentType='image/svg+xml' />`
- ☐ C. `<jsp:directive.page contentType='image/svg+xml' />`
- ☐ D. `<jsp:page.contentType>image/svg+xml</jsp:page.contentType>`

- 13** Dysponujemy stroną JSP z następującym wierszem:

```
<!-- out.print("Witaj świecie"); -->
```

Co znajdzie się w wynikowym kodzie HTML?

- ☐ A. **Witaj świecie**
- ☐ B. `out.print("Witaj świecie");`
- ☐ C. `<!-- Witaj świecie -->`
- ☐ D. Przedstawiony wiersz nie wygeneruje żadnych danych wyjściowych.

- 14** Które z poniższych zdań o obsłudze sesji protokołu HTTP są prawdziwe?  
(Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. Kontenery zgodne ze specyfikacją Java EE muszą obsługiwać ciasteczka HTTP.
- ☐ B. Kontenery zgodne ze specyfikacją Java EE muszą obsługiwać przepisywanie adresów URL.
- ☐ C. Kontenery zgodne ze specyfikacją Java EE muszą obsługiwać protokół Secure Sockets Layer (SSL).
- ☐ D. Kontenery zgodne ze specyfikacją Java EE muszą obsługiwać sesje HTTP, nawet jeśli klient nie obsługuje ciasteczek.
- ☐ E. Kontenery zgodne ze specyfikacją Java EE muszą rozpoznawać sygnał końca HTTP wskazujący na wyczerpanie aktywności sesji klienta.

- 15 Twoja firma zakupiła licencję umożliwiającą stosowanie w budowanych aplikacjach specjalnej biblioteki języka JavaScript stworzonej z myślą o konstruowaniu złożonych menu. Twój zespół długo nie mógł opanować sztuki korzystania z tej biblioteki, a wskutek popełnionych błędów wszyscy użytkownicy mieli dostęp do elementów menu, które powinny być widoczne tylko dla uprawnionych, wcześniej uwierzytelnionych ról. Biblioteka znaczników niestandardowych z prostymi obiektami obsługi powinna uchronić programistów przed błędami składniowymi w kodzie języka JavaScript i jednocześnie zapewnić niezbędne mechanizmy zabezpieczeń.

Po spotkaniu poświęconym rozwojowi projektu Twój szef opracował dokument, zgodnie z którym przykładowa deklaracja menu powinna mieć następującą postać:

```
<menu:main>
 <menu:headItem text="Moje konto" url="/mojeKonto.do"/>
 <menu:headItem text="Transakcje">
 <menu:subItem text="Przychodzące" url="/przychodzaceTr.do"/>
 <menu:subItem text="Wychodzące" url="/wychadzaceTr.do"/>
 <menu:subItem text="Oczekujące" url="/oczekujaceTr.do"
 requireRole="ksiegowy"/>
 </menu:headItem>
 <menu:headItem text="Administracja" url="/administracja.do"
 requireRole="administrator"/>
</menu:main>
```

Chciałbyś, aby pełna odpowiedzialność za generowanie kodu wynikowego spadła na klasę obsługującą zewnętrzny znacznik `<menu:main>`, ponieważ zakładasz, że centralizacja logiki wyświetlania menu ułatwi jej przyszłe utrzymanie. Klasa obsługi zewnętrznego znacznika będzie oczywiście wymagała dostępu do znaczników wewnętrznych. Które z zaproponowanych poniżej rozwiązań wydaje Ci się najlepsze?

- ☐ A. Każdy znacznik wewnętrzny powinien się rejestrować w swoim bezpośrednim znaczniku macierzystym. Znacznik macierzysty może składać informacje o swoich bezpośrednich potomkach w uporządkowanej kolekcji.
- ☐ B. Każdy znacznik wewnętrzny powinien się rejestrować w klasie obsługi znacznika zewnętrznego, która może składać informacje o wszystkich (pośrednich i bezpośrednich) znacznikach potomnych w jednej strukturze typu **HashSet**.
- ☐ C. W przeciwieństwie do znaczników klasycznych, klasa **SimpleTagSupport** definiuje metody **findDescendentWithClass()** i **getChildren()**, które zapewniają głównemu znacznikowi zewnętrznemu pełen dostęp do jego potomków bez konieczności samodzielnego tworzenia dodatkowych mechanizmów.
- ☐ D. Każdy znacznik wewnętrzny powinien zapisać sam siebie w formie atrybutu zasięgu strony, gdzie wartość **text** będzie pełniła funkcję klucza tego atrybutu.

- 16 Która faza cyklu życia strony JSP może spowodować zwrócenie kodu błędu 500 protokołu HTTP w odpowiedzi na żądanie danej strony? (Zaznacz wszystkie prawidłowe odpowiedzi).
- ☐ A. Kompilacja strony JSP.
  - ☐ B. Wykonanie metody obsługi żądania.
  - ☐ C. Wykonanie metody niszczenia strony.
  - ☐ D. Wykonanie metody inicjalizującej.
- 
- 17 Jeśli przyjmimy, że **session** jest referencją do prawidłowego obiektu **HttpSession**, a **"mojAtrybut"** jest nazwą obiektu skojarzonego z tą referencją, którego wywołania należałoby użyć do usunięcia skojarzenia tych obiektów? (Zaznacz wszystkie prawidłowe odpowiedzi).
- ☐ A. `session.unbind();`
  - ☐ B. `session.invalidate();`
  - ☐ C. `session.unbind("mojAtrybut");`
  - ☐ D. `session.remove("mojAtrybut");`
  - ☐ E. `session.invalidate("mojAtrybut");`
  - ☐ F. `session.removeAttribute("mojAtrybut");`
  - ☐ G. `session.unbindAttribute("mojAtrybut");`
- 
- 18 Jeśli **req** jest referencją do obiektu **HttpServletRequest** i jeśli bieżąca sesja nie istnieje, które zdania o metodzie **req.getSession()** są prawdziwe? (Zaznacz wszystkie prawidłowe odpowiedzi).
- ☐ A. Wywołanie metody **req.getSession()** zwróci wartość **null**.
  - ☐ B. Wywołanie metody **req.getSession(true)** zwróci wartość **null**.
  - ☐ C. Wywołanie metody **req.getSession(false)** zwróci wartość **null**.
  - ☐ D. Wywołanie metody **req.getSession()** zwróci nową sesję.
  - ☐ E. Wywołanie metody **req.getSession(true)** zwróci nową sesję.
  - ☐ F. Wywołanie metody **req.getSession(false)** zwróci nową sesję.

- 19 Dotychczasowy kod aplikacji obejmuje między innymi klasyczną klasę obsługi znacznika. Autor tej klasy stworzył mechanizm przetwarzania ciała znacznika sto razy, aby za jej pomocą można było testować pozostałe znaczniki generujące losową zawartość.

Mamy następujący kod:

```

06. public class HundredTimesTag extends TagSupport {
07. private int iterationCount;
08. public int doTag() throws JspException {
09. iterationCount = 0;
10. return EVAL_BODY_INCLUDE;
11. }
12.
13. public int doAfterBody() throws JspException {
14. if (iterationCount < 100) {
15. iterationCount++;
16. return EVAL_BODY_AGAIN;
17. } else {
18. return SKIP_BODY;
19. }
20. }
21. }

```

Jaki błąd popełniono w tym kodzie?

- ☐ A. Klasy obsługi znaczników nie gwarantują bezpieczeństwa przetwarzania wielowątkowego, zatem zmienna **iterationCount** może zawierać nieprawidłową wartość, jeśli wielu użytkowników jednocześnie zażąda danej strony.
- ☐ B. Metoda **doAfterBody** nigdy nie zostanie wywołana, ponieważ nie jest częścią cyklu życia klasy obsługi znacznika. Włączenie tej metody do cyklu życia wymagałoby od programisty rozszerzenia klasy **IterationTagSupport**.
- ☐ C. Metodę **doTag** należałoby zastąpić metodą **doStartTag**. W powyższym kodzie jest wywoływana domyślna wersja metody **doStartTag** klasy **TagSupport**, która zwraca wartość **SKIP\_BODY**, co z kolei powoduje, że metoda **doAfterBody** nigdy nie jest wywoływana.
- ☐ D. Kiedy metoda **doAfterBody** zwraca wartość **EVAL\_BODY\_AGAIN**, następuje ponowne wywołanie metody **doTag**. Metoda **doTag** przywraca wartość **0** zmiennej **iterationCount**, zatem cała pętla jest nieskończona, co ostatecznie prowadzi do wygenerowania wyjątku **java.lang.OutOfMemoryError**.

**20** Mamy dany następujący fragment deskryptora wdrożenia aplikacji internetowej:

```
72. <session-config>
73. <session-timeout>10</session-timeout>
74. </session-config>
```

Przyjmijmy, że **session** jest referencją do prawidłowego obiektu **HttpSession**. Nasz serwlet zawiera następujący wiersz:

```
30. session.setMaxInactiveInterval(120);
```

Które zdanie prawidłowo opisuje sytuację po wykonaniu wiersza 30.? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. Przedstawiony fragment deskryptora wdrożenia jest błędny.
- ☐ B. Wywołanie metody **setMaxInactiveInterval** zmodyfikuje wartość zadeklarowaną w znaczniku **<session-timeout>**.
- ☐ C. Na podstawie przedstawionych fragmentów nie można określić limitu czasowego sesji.
- ☐ D. Jeśli w ramach danej sesji kontener nie otrzyma od klienta żadnego żądania przez 2 godziny, sesja zostanie unieważniona.
- ☐ E. Jeśli w ramach danej sesji kontener nie otrzyma od klienta żadnego żądania przez 2 minuty, sesja zostanie unieważniona.
- ☐ F. Jeśli w ramach danej sesji kontener nie otrzyma od klienta żadnego żądania przez 10 sekund, sesja zostanie unieważniona.
- ☐ G. Jeśli w ramach danej sesji kontener nie otrzyma od klienta żadnego żądania przez 10 minut, sesja zostanie unieważniona.

**21** Utworzyłeś prawidłową strukturę katalogów i poprawny plik **WAR** dla swojej aplikacji internetowej napisanej w technologii Java EE. Jeśli wiemy, że:

- plik **WAR** nazwano **PoprawnaAplicacja.war**;
- **WARdir** reprezentuje katalog, który musi istnieć w każdym pliku **WAR**;
- **APPdir** reprezentuje katalog, który musi istnieć w każdej aplikacji internetowej,

które zdanie jest prawdziwe?

- ☐ A. Określenie rzeczywistej nazwy katalogu **WARdir** NIE jest możliwe.
- ☐ B. Określenie nazwy aplikacji NIE jest możliwe.
- ☐ C. W opisywanej strukturze katalogów katalog **APPdir** będzie się znajdował wewnątrz katalogu **WARdir**.
- ☐ D. W opisywanej strukturze katalogów deskryptor wdrożenia będzie się znajdował w tym samym katalogu co katalog **WARdir**.
- ☐ E. Umieszczenie aplikacji w pliku **WAR** stwarza kontenerowi możliwość przeprowadzenia w czasie wykonywania dodatkowych testów, które w przeciwnym razie nie byłyby możliwe.

**22** Które zdanie o metodach **GET** i **POST** protokołu HTTP jest prawdziwe? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. Tylko metoda **GET** jest idempotentna.
- ☐ B. Obie metody wymagają zadeklarowania wprost w znaczniku **form** języka HTML.
- ☐ C. Tylko metoda **POST** może obsługiwać wiele parametrów w jednym żądaniu.
- ☐ D. Obie metody obsługują żądania z pojedynczymi parametrami reprezentującymi wiele wartości.
- ☐ E. Tylko żądania **POST** powinny być obsługiwane przez nadpisaną metodę **service()** serwletu.

**23** Nasz serwlet zawiera następujący wiersz:

```
82. String s = getServletConfig().getInitParameter("mojParametr");
```

Który fragment deskryptora wdrożenia spowoduje przypisanie zmiennej **s** wartości **"mojaWartosc"**?

- ☐ A. 

```
<init-param>
 <param>mojParametr</param>
 <value>mojaWartosc</value>
</init-param>
```
- ☐ B. 

```
<init-param>
 <name>mojParametr</name>
 <value>mojaWartosc</value>
</init-param>
```
- ☐ C. 

```
<init-param>
 <param-name>mojParametr</param-name>
 <param-value>mojaWartosc</param-value>
</init-param>
```
- ☐ D. 

```
<servlet-param>
 <name>mojParametr</name>
 <value>mojaWartosc</value>
</servlet-param>
```
- ☐ E. 

```
<servlet-param>
 <param-name>mojParametr</param-name>
 <param-value>mojaWartosc</param-value>
</servlet-param>
```

**24** Wiemy, że jakiś łańcuch jest składowany w jednym z zasięgów w formie atrybutu nazwanego **numerKonta**. Który z poniższych skryptletów wyświetli wartość tego atrybutu?

- ☐ A. `<%= pageContext.findAttribute("numerKonta") %>`
- ☐ B. `<%= out.print("${numerKonta}") %>`
- ☐ C. `<% Object nrKonta = pageContext.getAttribute("numerKonta");  
if(nrKonta == null) {  
nrKonta = request.getAttribute("numerKonta");  
}  
if(nrKonta == null) {  
nrKonta = session.getAttribute("numerKonta");  
}  
if(nrKonta == null) {  
nrKonta = servletContext.getAttribute("numerKonta");  
}  
out.print(nrKonta);  
%>`
- ☐ D. `<% requestDispatcher.include("numerKonta"); %>`

**25** Odziedziczyłeś stosowaną od dawna aplikację internetową złożoną ze stron JSP zawierających mnóstwo kodu skryptowego. Szef żąda od Ciebie takiej przebudowy każdej z tych stron, aby nie zawierała żadnego kodu skryptowego. Masz mu zagwarantować, że w bazie kodowej JSP nie znajdzie się ani jeden skryptlet — w ten sposób będzie można zrealizować w kontenerze „politykę bezskryptową”.

Który element konfiguracyjny użyty w pliku **web.xml** pozwoli osiągnąć ten cel?

- ☐ A. `<jsp-property-group>  
    <url-pattern> *.jsp </url-pattern>  
    <permit-scripting> false </permit-scripting>  
</jsp-property-group>`
- ☐ B. `<jsp-config>  
    <url-pattern> *.jsp </url-pattern>  
    <permit-scripting> false </permit-scripting>  
</jsp-config>`
- ☐ C. `<jsp-property-group>  
    <url-pattern> *.jsp </url-pattern>  
    <scripting-invalid> true </scripting-invalid>  
</jsp-property-group>`
- ☐ D. `<jsp-config>  
    <url-pattern> *.jsp </url-pattern>  
    <scripting-invalid> true </scripting-invalid>  
</jsp-config>`

26 Dysponujesz następującym fragmentem kodu:

```
01. <%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
02.
03. <%
04. java.util.List books = new java.util.ArrayList();
05. // tutaj wstaw brakujący wiersz
06. request.setAttribute("myFavoriteBooks", books);
07. %>
08.
09. <c:choose>
10. <c:when test="${not empty myFavoriteBooks}">
11. Oto moje ulubione książki:
12. <c:forEach var="book" items="${myFavoriteBooks}">
13.
 * ${book}
14. </c:forEach>
15. </c:when>
16. <c:otherwise>
17. Nie wybrałem żadnej ulubionej książki.
18. </c:otherwise>
19. </c:choose>
```

Który z poniższych wierszy kodu wstawiony w wierszu 5. przedstawionego listingu spowoduje wyświetlenie tekstu wewnątrz znacznika **c:otherwise**? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. `books.add("");`
- ☐ B. `books.add(null);`
- ☐ C. `books.clear();`
- ☐ D. `books.add("Head First");`
- ☐ E. `books = null;`

---

**27** Pracujesz nad aplikacją zarządzającą katalogami list biznesowych.

Masz dany następujący kod:

```
29. <c:forEach var="phoneNumber" items='${company.contactInfo.phoneNumbers}'>
30. <c:if test='${verify:isTollFree(phoneNumber)}'>
31.
32. </c:if>
33. ${phoneNumber}

34. </c:forEach>
```

Przedstawiony fragment kodu poprzedza specjalną ikoną numery telefonów, z którymi można się łączyć za darmo. Które zdanie o użytej powyżej funkcji języka EL na pewno jest prawdziwe?

- ☐ A. Użyta funkcja języka EL musi zostać zadeklarowana jako składowa publiczna i statyczna.
- ☐ B. Użyta funkcja języka EL nie może zwracać żadnej wartości (musi zwracać typ **void**).
- ☐ C. W elemencie **<uri>** deklaracji TLD tej funkcji EL należy użyć wartości **Verify**.
- ☐ D. Klasa implementująca tę funkcję języka EL musi się nazywać **Verify**.
- ☐ E. Jeśli **phoneNumber** jest łańcuchem, element **<function-signature>** odpowiedniej deklaracji TLD powinien zawierać sygnaturę **isTollFree(String)**.

---

**28** Które z wymienionych metod interfejsu **HttpServletRequest** zapewniają dostęp do ciała żądania? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. **getReader()**
- ☐ B. **getStream()**
- ☐ C. **getInputReader()**
- ☐ D. **getInputStream()**
- ☐ E. **getServletReader()**
- ☐ F. **getServletStream()**

- 29 Dysponujesz aplikacją internetową stworzoną w technologii Java EE. Do Twojej aplikacji trafia następujące żądanie:

**http://www.wickedlysmart.com/MojaAplikacja/mojKatalog/ZrobCos**

Przytoczone żądanie zostanie obsłużone przez pewien serwlet. Które zdania są prawdziwe? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. Deskryptor wdrożenia musi zawierać instrukcje opisujące, jak obsłużyć żądanie w tej formie.
- ☐ B. Powyższe żądanie może zostać prawidłowo obsłużone mimo braku odpowiednich instrukcji w deskrypcji wdrożenia.
- ☐ C. Serwlet obsługujący to żądanie musi się nazywać **ZrobCos.class**.
- ☐ D. Na podstawie podanych informacji określenie nazwy tego serwletu jest niemożliwe.
- ☐ E. Aplikacja musi zawierać katalog nazwany **mojKatalog**.
- ☐ F. Na podstawie podanych informacji określenie nazwy katalogu, w którym umieszczono dany serwlet, jest niemożliwe.

- 30 Twoja aplikacja internetowa obejmuje prawidłowy deskryptor wdrożenia, w którym zdefiniowano tylko dwie role zabezpieczeń: **uczen** i **mistrz**. Wspomniany deskryptor zawiera też dwa ograniczenia zabezpieczeń odnoszące się do tego samego zasobu. Pierwsze z tych ograniczeń ma następującą postać:

```
234. <auth-constraint>
235. <role-name>uczen</role-name>
236. </auth-constraint>
```

Drugie ograniczenie zdefiniowano w następujący sposób:

```
251. <auth-constraint/>
```

Które zdania są prawdziwe? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. Deskryptor wdrożenia w tej formie deklaruje, że chroniony zasób jest dostępny dla obu ról.
- ☐ B. Deskryptor wdrożenia w tej formie deklaruje, że chroniony zasób jest dostępny tylko dla użytkowników przypisanych do roli **mistrz**.
- ☐ C. Deskryptor wdrożenia w tej formie deklaruje, że chroniony zasób jest dostępny tylko dla użytkowników przypisanych do roli **uczen**.
- ☐ D. Gdybyśmy usunęli drugi znacznik **<auth-constraint>**, chroniony zasób byłby dostępny dla obu ról.
- ☐ E. Gdybyśmy usunęli drugi znacznik **<auth-constraint>**, chroniony zasób byłby dostępny tylko dla użytkowników przypisanych do roli **mistrz**.
- ☐ F. Gdybyśmy usunęli drugi znacznik **<auth-constraint>**, chroniony zasób byłby dostępny tylko dla użytkowników przypisanych do roli **uczen**.

**31** Który z poniższych znaczników niestandardowych na pewno nie zadziała?  
(Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. `<moje:ramka>`  
`<moje:zdjecia album="${wybranyAlbum}">`  
`</moje:ramka>`  
`</moje:zdjecia>`
  - ☐ B. `<moje:ramka>`  
`<moje:zdjecia album="${wybranyAlbum}"/>`  
`</moje:ramka>`
  - ☐ C. `<moje:ramka>`  
`${wybranyAlbum.tytul}`  
`<moje:zdjecia>${wybranyAlbum}</moje:zdjecia>`  
`</moje:ramka>`
  - ☐ D. `<moje:zdjecia includeBorder="${userPreference.ramka}"`  
`album="${wybranyAlbum}" />`
- 

**32** Twoja  $n$ -warstwowa aplikacja internetowa wykorzystuje najbardziej popularne wzorce projektowe technologii Java EE w operacjach dostępu do zdalnych rejestrów. Jakie są korzyści stosowania tych wzorców? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. Większa spójność.
  - ☐ B. Wyższa wydajność.
  - ☐ C. Łatwość utrzymania.
  - ☐ D. Mniejsze obciążenie sieci.
  - ☐ E. Szersze możliwości w zakresie interaktywności przeglądarki.
- 

**33** Które zdania o cyklu życia serwletu są prawdziwe? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. NIE powinien pisać konstruktora dla serwletu.
- ☐ B. NIE powinien nadpisywać metody `init()` serwletu.
- ☐ C. NIE powinien nadpisywać metody `doGet()` serwletu.
- ☐ D. NIE powinien nadpisywać metody `doPost()` serwletu.
- ☐ E. NIE powinien nadpisywać metody `service()` serwletu.
- ☐ F. NIE powinien nadpisywać metody `destroy()` serwletu.

**34** Mamy dany następujący fragment struktury katalogów aplikacji Javy EE w ramach pliku `.war`:

**MojaAplikacja**

```

|-- META-INF
| |-- MANIFEST.MF
| |-- web.xml
|
|-- WEB-INF
| |-- index.html
| |-- TLDs
|
| |-- Naglowek.tag

```

Jaka zmiana (zmiany) jest konieczna, aby przedstawiona struktura była prawidłowa i aby zasoby tej aplikacji były dostępne? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. Nie potrzeba żadnych zmian.
- ☐ B. Należy przenieść w inne miejsce plik **web.xml**.
- ☐ C. Należy przenieść w inne miejsce plik **index.html**.
- ☐ D. Należy przenieść w inne miejsce plik **Naglowek.tag**.
- ☐ E. Należy przenieść w inne miejsce plik **MANIFEST.MF**.
- ☐ F. Należy przenieść w inne miejsce katalog **WEB-INF**.
- ☐ G. Należy przenieść w inne miejsce katalog **META-INF**.

**35** Zastanawiasz się nad implementacją pewnej odmiany wzorca MVC w swojej  $n$ -warstwowej aplikacji Javy EE. Które zdania są prawdziwe? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. Tego rodzaju projekty często wykorzystują obiekty delegatów biznesowych.
- ☐ B. Składowanie danych pobieranych ze zdalnego komponentu często pozwala ograniczyć obciążenie sieci.
- ☐ C. Postawiony cel projektowy upraszcza komunikację z heterogenicznymi rejestrami zasobów.
- ☐ D. Rozwiązania na bazie wzorca MVC mają co prawda wiele zalet, ale często komplikują projekt aplikacji.
- ☐ E. Warto rozważyć realizację tego celu projektowego z wykorzystaniem wzorca Front Controller (kontrolera frontowego) i frameworku Struts.
- ☐ F. Gotowy projekt powinien w przyszłości ułatwić przebudowę klas odpowiedzialnych za obsługę żądań i odpowiedzi.

**36** Masz daną stronę JSP z następującym wierszem:

```
<% List mojaLista = new ArrayList(); %>
```

Który z poniższych fragmentów kodu JSP można wykorzystać do zaimportowania tych typów danych? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. `<%! import java.util.*; %>`
- ☐ B. `<%@ import java.util.List java.util.ArrayList %>`
- ☐ C. `<%@ page import='java.util.List,java.util.ArrayList' %>`
- ☐ D. `<%! import java.util.List; import java.util.ArrayList; %>`
- ☐ E. `<%@ page import='java.util.List' %> <%@ page import='java.util.ArrayList' %>`

**37** Otrzymałeś zadanie dodania kilku mechanizmów zabezpieczeń do używanej w Twojej firmie aplikacji internetowej Javy EE. W szczególności musisz opracować szereg klas użytkowników, by na tej podstawie określać, które grupy mają dostęp do poszczególnych stron aplikacji. Kontrola dostępu do chronionych stron wymaga potwierdzenia tożsamości użytkowników.

Które zdania są prawdziwe? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. Jeśli musisz sprawdzać, czy użytkownicy rzeczywiście są tymi, za których się podają, powinieneś odpowiednie wymaganie zadeklarować w deskrytorze wdrożenia aplikacji.
- ☐ B. Do sprawdzania, czy użytkownicy są tymi, za których się podają, należy wykorzystać mechanizmy autoryzacji Javy EE.
- ☐ C. Aby uprościć proces weryfikacji, czy użytkownicy są tymi, za których się podają, można użyć znaczników `<login-config>` deskryptora wdrożenia.
- ☐ D. Aby uprościć proces weryfikacji, czy użytkownicy są tymi, za których się podają, można użyć znaczników `<user-data-constraint>` deskryptora wdrożenia.
- ☐ E. W zależności od stosowanego rozwiązania określanie, czy użytkownicy są tymi, za których się podają, może wymagać dołączenia domeny (`realm`).

**38** PoprawnaAplikacja to aplikacja internetowa Javy EE z prawidłową strukturą katalogów. PoprawnaAplikacja obejmuje między innymi pliki graficzne *.gif* składowane w trzech katalogach tej struktury:

- **PoprawnaAplikacja/imageDir/**
- **PoprawnaAplikacja/META-INF/**
- **PoprawnaAplikacja/WEB-INF/**

W którym z wymienionych poniżej katalogów należy umieścić pliki *.gif*, aby były bezpośrednio dostępne dla klientów?

- ☐ A. Tylko w katalogu **PoprawnaAplikacja/META-INF/**.
- ☐ B. Tylko w katalogu **PoprawnaAplikacja/imageDir/**.
- ☐ C. We wszystkich wymienionych katalogach.
- ☐ D. Tylko w katalogach **PoprawnaAplikacja/imageDir/** i **PoprawnaAplikacja/WEB-INF/**.
- ☐ E. Tylko w katalogach **PoprawnaAplikacja/imageDir/** i **PoprawnaAplikacja/META-INF/**.

**39** Przyjmijmy, że **req** jest referencją do obiektu **HttpServletRequest** oraz że dysponujemy kodem w postaci:

```
13. String[] s = req.getCookies();
14. Cookie[] c = req.getCookies();
15. req.setAttribute("mojAtrybut1", "42");
16. req.setAttribute("mojAtrybut2", 42);
17. String[] s2 = req.getAttributeNames();
18. String[] s3 = req.getParameterValues("attr");
```

Które wiersze tego kodu zostaną odrzucone przez kompilator? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. Wiersz 13.
- ☐ B. Wiersz 14.
- ☐ C. Wiersz 15.
- ☐ D. Wiersz 16.
- ☐ E. Wiersz 17.
- ☐ F. Wiersz 18.

**40** Plik znacznika nazwany **Products.tag** wyświetla listę produktów.

Dysponujemy następującym fragmentem tego pliku:

1. `<%@ attribute name="header" required="false" rtexprvalue="false" %>`
2. `<%@ attribute name="products" required="true" rtexprvalue="true" %>`
3. `<%@ tag body-content="tagdependent" %>`

Które z poniższych zastosowań tego pliku znacznika są prawidłowe? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. `<display:Products header="Koszyk z zakupami" products="${shoppingCart}"/>`
- ☐ B. `<display:Products header="Lista życzeń" products="${wishList}" body-content="${body}"/>`
- ☐ C. `<display:Products header="Podobne produkty" products="${similarProducts}"  
Klienci, którzy kupili ten produkt, kupili też:  
</display:Products>`
- ☐ D. `<display:Products header='<%= request.getParameter("listType") %>' />`

**41** Bierzesz udział w przedsięwzięciu polegającym na eliminowaniu skryptletów z kodu JSP przestarzałej aplikacji dużego banku. Odkrywasz w modyfikowanym kodzie następujące wiersze:

```
<% if((com.yourcompany.Account)request.getAttribute("account")).
isPersonalChecking()){ %>
```

Weryfikacja zgodności z Twoim stylem życia.

```
<% } %>
```

Którym znacznikiem biblioteki JSTL można zastąpić tę konstrukcję? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. `<c:if test='${account.personalChecking}'> Weryfikacja zgodności  
z Twoim stylem życia.</c:if>`
- ☐ B. `<c:if test='${account["personalChecking"]}'> Weryfikacja zgodności  
z Twoim stylem życia.</c:if>`
- ☐ C. `<c:if test='${account['personalChecking']}'> Weryfikacja zgodności  
z Twoim stylem życia.</c:if>`
- ☐ D. `<c:if test='${account.isPersonalChecking}'> Weryfikacja zgodności  
z Twoim stylem życia.</c:if>`

**42** Mamy następujące typy zdarzeń:

- `HttpSessionEvent`
- `HttpSessionBindingEvent`
- `HttpSessionAttributeEvent`

Dopasuj powyższe typy zdarzeń do odpowiednich interfejsów nasłuchujących. (Uwaga: pojedynczy typ zdarzeń można dopasować do więcej niż jednego interfejsu).

<code>HttpSessionAttributeListener</code>	.....
<code>HttpSessionListener</code>	.....
<code>HttpSessionActivationListener</code>	.....
<code>HttpSessionBindingListener</code>	.....

**43** Które zdanie o cyklu życia serwletu jest prawdziwe? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. Metoda **`service()`** jest wywoływana przez kontener jako pierwsza po otrzymaniu nowego żądania.
- ☐ B. Metoda **`service()`** jest wywoływana albo przez metodę **`doPost()`**, albo przez metodę **`doGet()`** już po przetworzeniu żądania przez te metody.
- ☐ C. Każde wywołanie metody **`doPost()`** jest realizowane w odrębnym wątku.
- ☐ D. Metoda **`destroy()`** jest każdorazowo wywoływana po wykonaniu metody **`doGet()`**.
- ☐ E. Kontener tworzy odrębny wątek dla każdego żądania klienta.

**44** Kiedy następuje tłumaczenie kodu strony JSP? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. Kiedy programista kompiluje kod umieszczony w folderze **`src`**.
- ☐ B. W chwili uruchamiania aplikacji.
- ☐ C. Po otrzymaniu pierwszego żądania danej strony JSP od użytkownika.
- ☐ D. Po wywołaniu metody **`jspDestroy()`**.

---

**45** Mamy dany następujący fragment prawidłowej metody **doGet()**:

```
12. OutputStream os = response.getOutputStream();
13. byte[] ba = {1,2,3};
14. os.write(ba);
15. RequestDispatcher rd = request.getRequestDispatcher("moja.jsp");
16. rd.forward(request, response);
```

Jeśli przyjmiemy, że plik **moja.jsp** dodaje bajty **4, 5 i 6** do odpowiedzi, jaki będzie wynik?

- ☐ A. 123
- ☐ B. 456
- ☐ C. 123456
- ☐ D. 456123
- ☐ E. Zostanie wygenerowany wyjątek.

---

**46** Programista musi tak zaktualizować parametry inicjalizacji żywego, działającego serwletu, aby dana aplikacja internetowa natychmiast zaczęła uwzględniać nowe parametry.

Które zdanie prawidłowo opisuje niezbędne (choć niekoniecznie wystarczające) działania? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. Dla każdego parametru należy zmodyfikować znacznik deskryptora wdrożenia opisujący nazwę serwletu, nazwę samego parametru oraz jego nową wartość.
- ☐ B. Konstruktor serwletu musi odczytać zaktualizowane parametry (zadeklarowane w deskrypcji wdrożenia) za pośrednictwem obiektu **ServletConfig**.
- ☐ C. Kontener musi zniszczyć i ponownie zainicjalizować dany serwlet.
- ☐ D. Dla każdego parametru deskryptor wdrożenia musi zawierać osobny znacznik **<init-param>**.

---

**47** Które typy można stosować łącznie z metodami interfejsu **HttpServletResponse** do kierowania danych wyjściowych do odpowiedniego strumienia? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. **java.io.PrintStream**
- ☐ B. **java.io.PrintWriter**
- ☐ C. **javax.servlet.OutputStream**
- ☐ D. **java.io.FileOutputStream**
- ☐ E. **javax.servlet.ServletOutputStream**
- ☐ F. **java.io.ByteArrayOutputStream**

**48** Twoja aplikacja internetowa obejmuje prawidłowy deskryptor wdrożenia z pojedynczym znacznikiem **<security-constraint>**. Wspomniany znacznik zawiera:

- pojedynczy wzorzec adresów URL z katalogiem **katalog1**;
- pojedynczą metodę protokołu HTTP z wartością **POST**;
- pojedynczą nazwę roli wskazującą rolę **GOSC**.

Jeśli przyjąć, że wszystkie zasoby Twojej aplikacji znajdują się w katalogach **katalog1** i **katalog2**, i jeśli inną prawidłową rolą jest **CZLONEK**, które zdania są prawdziwe? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. Użytkownicy przypisani do roli **GOSC** nie mogą kierować żądań **GET** do zasobów w katalogu **katalog1**.
- ☐ B. Użytkownicy przypisani do roli **GOSC** mogą kierować żądania **GET** do zasobów w obu katalogach.
- ☐ C. Użytkownicy przypisani do roli **GOSC** mogą kierować żądania **POST** tylko do zasobów w katalogu **katalog2**.
- ☐ D. Użytkownicy przypisani do roli **CZLONEK** mogą kierować żądania **GET** do zasobów w obu katalogach.
- ☐ E. Użytkownicy przypisani do roli **GOSC** mogą kierować żądania **POST** do zasobów w obu katalogach.
- ☐ F. Użytkownicy przypisani do roli **CZLONEK** mogą kierować tylko żądania **POST** do zasobów w katalogu **katalog1**.

**49** Dysponujesz następującym kodem:

1. `<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>`
2. `<%@ taglib prefix="tables" uri="http://www.javaranch.com/tables" %>`
3. `<%@ taglib prefix="jsp" tagdir="/WEB-INF/tags" %>`
4. `<%@ taglib uri="UtilityFunctions" prefix="util" %>`

Które elementy powyższych dyrektyw **taglib** uniemożliwią właściwe funkcjonowanie tej strony JSP?

- ☐ A. Wiersz 4. zawiera błąd, ponieważ atrybut **prefix** musi się znajdować przed atrybutem **uri**.
- ☐ B. Wiersz 3. zawiera błąd, ponieważ zabrakło atrybutu **uri**.
- ☐ C. Wiersz 4. zawiera błąd, ponieważ wartość atrybutu **uri** musi się rozpoczynać od sekwencji **http://**.
- ☐ D. Wiersz 3. zawiera błąd, ponieważ przedrostek **jsp** jest zarezerwowany dla akcji standardowych.

- 50** Przyjmijmy, że **resp** jest referencją do prawidłowego obiektu **HttpServletResponse** zawierającego między innymi następujące nagłówki:

**Content-Type: text/html**

**MojNaglowek: mojedane**

Kod serwletu zawiera następujące wywołania:

```
25. resp.addHeader("MojNaglowek", "mojedane2");
26. resp.setHeader("MojNaglowek", "mojedane3");
27. resp.addHeader("MojNaglowek", "mojedane");
```

Jakie dane ostatecznie znajdują się w nagłówku **MojNaglowek**?

- ☐ A. **mojedane**
- ☐ B. **mojedane3**
- ☐ C. **mojedane3,mojedane**
- ☐ D. **mojedane3,mojedane2**
- ☐ E. **mojedane,mojedane2,mojedane3**
- ☐ F. **mojedane,mojedane2,mojedane3,mojedane**

- 51** Dysponujesz następującym fragmentem pliku *web.xml* starej, odziedziczonej aplikacji:

```
<jsp-config>
 <taglib>
 <taglib-uri>prettyTables</taglib-uri>
 <taglib-location>/WEB-INF/tlds/prettyTables.tld</taglib-location>
 </taglib>
</jsp-config>
```

Przyjmij, że serwer wykonujący Twój kod jest teraz zgodny ze specyfikacją Java 1.4 EE lub nowszą. Co możesz zrobić, aby usunąć powyższy znacznik **<jsp-config>** i jednocześnie zachować możliwość wykonywania tego kodu?

- ☐ A. Można tak zmienić atrybut **uri** dyrektywy **taglib** w kodzie stron JSP, aby zawierał wartość **"\*"**, która zostanie automatycznie odwzorowana przez kontener.
- ☐ B. W pliku TLD należy umieścić znacznik **<uri>prettyTables</uri>**.
- ☐ C. Należy usunąć z kodu JSP dyrektywy **taglib**, które wykorzystywały to odwzorowanie. Kontener automatycznie obsłuży tę zmianę.
- ☐ D. To niemożliwe. Element **<jsp-config>** jest niezbędny, aby kontener mógł odwzorowywać ten TLD na identyfikator **uri** wykorzystywany w kodzie stron JSP.

**52** Strona prezentująca zawartość koszyka z zakupami powinna wyświetlać komunikat *Twój koszyk jest pusty* w sytuacji, gdy użytkownik nie doda do koszyka żadnych produktów. Który z przedstawionych poniżej fragmentów kodu prawidłowo realizuje to zadanie, zakładając, że atrybut **cart** reprezentuje listę produktów? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. 

```
<c:if test='${empty cart}'>
 Twój koszyk jest pusty.
</c:if>
<c:forEach var="itemInCart" items="${cart}">
 <shop:displayItem item="${itemInCart}"/>
</c:forEach>
```
- ☐ B. 

```
<c:forEach var="itemInCart" items="${cart}">
 <c:choose>
 <c:when test='${empty itemInCart}'>
 Twój koszyk jest pusty.
 </c:when>
 <c:otherwise>
 <shop:displayItem item="${itemInCart}"/>
 </c:otherwise>
 </c:choose>
</c:forEach>
```
- ☐ C. 

```
<c:choose>
 <c:when test='${empty cart}'>
 Twój koszyk jest pusty.
 </c:when>
 <c:when test='${not empty cart}'>
 <c:forEach var="itemInCart" items="${cart}">
 <shop:displayItem item="${itemInCart}"/>
 </c:forEach>
 </c:when>
</c:choose>
```
- ☐ D. 

```
<c:choose>
 <c:when test='${empty cart}'>
 Twój koszyk jest pusty.
 </c:when>
 <c:otherwise>
 <c:forEach var="itemInCart" items="${cart}">
 <shop:displayItem item="${itemInCart}"/>
 </c:forEach>
 </c:otherwise>
</c:choose>
```

**59** Przyjmijmy, że nasz serwlet zawiera następujący kod i że **mojaZmienna** jest referencją albo do obiektu **HttpSession**, albo do obiektu **ServletContext**.

```
15. mojaZmienna.setAttribute("mojaNazwa", "mojaWartosc");
16. String s = (String) mojaZmienna.getAttribute("mojaNazwa");
17. // dalszy kod
```

Które zdanie opisujące sytuację po wykonaniu wiersza 16. jest prawdziwe? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. Nie można jednoznacznie określić wartości zmiennej **s**.
- ☐ B. Jeśli **mojaZmienna** jest referencją do obiektu **HttpSession**, próba kompilacji zakończy się niepowodzeniem.
- ☐ C. Jeśli **mojaZmienna** jest referencją do obiektu **ServletContext**, próba kompilacji zakończy się niepowodzeniem.
- ☐ D. Jeśli **mojaZmienna** jest referencją do obiektu **HttpSession**, zmienna **s** na pewno będzie zawierała łańcuch **"mojaWartość"**.
- ☐ E. Jeśli **mojaZmienna** jest referencją do obiektu **ServletContext**, zmienna **s** na pewno będzie zawierała łańcuch **"mojaWartość"**.

**54** Dysponujemy następującym fragmentem deskryptora wdrożenia aplikacji internetowej Javy EE:

```
62. <error-page>
63. <exception-type>IOException</exception-type>
64. <location>/mainError.jsp</location>
65. </error-page>
66. <error-page>
67. <error-code>404</error-code>
68. <location>/notFound.jsp</location>
69. </error-page>
```

Które zdanie jest prawdziwe?

- ☐ A. Deskryptor wdrożenia w tej formie jest nieprawidłowy.
- ☐ B. Jeśli aplikacja wygeneruje wyjątek **IOException**, nie zostanie zwrócona żadna strona.
- ☐ C. Jeśli aplikacja wygeneruje wyjątek **IOException**, zostanie zwrócona strona **notFound.jsp**.
- ☐ D. Jeśli aplikacja wygeneruje wyjątek **IOException**, zostanie zwrócona strona **mainError.jsp**.

**55** Dysponujemy następującym kodem JSP:

1. `<%! String GREETING = "Witaj na mojej stronie"; %>`
2. `<% request.setAttribute("greeting", GREETING); %>`
3. `Pozdrowienie: ${greeting}`
4. `Jeszcze raz: <%= request.getAttribute("greeting") %>`

Podjęto próbę konwersji tej strony na następujący dokument JSP:

01. `<jsp:declaration>`
02. `String GREETING = "Witaj na mojej stronie";`
03. `</jsp:declaration>`
04. `<jsp:scriptlet>`
05. `request.setAttribute("greeting", GREETING);`
06. `</jsp:scriptlet>`
07. `Pozdrowienie: ${greeting}`
08. `Jeszcze raz: <jsp:expression>`
09. `request.getAttribute("greeting");`
10. `</jsp:expression>`

Jaki błąd popełniono w nowym dokumencie JSP? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. Nie zadeklarowano elementu `<jsp:root>`.
- ☐ B. Tekst szablonu należałoby opakować w znaczniku `<jsp:text>`.
- ☐ C. W dokumentach JSP nie można stosować wyrażeń języka EL.
- ☐ D. W zawartości znacznika `<jsp:expression>` nie należy stosować średników.

**56** Która z poniższych składowych aplikacji internetowej ma NAJMNIEJSZE szanse na otrzymanie wywołania za pośrednictwem sieci?

- ☐ A. serwer JNDI
- ☐ B. obiekt transferu
- ☐ C. lokalizator usługi
- ☐ D. kontroler frontowy
- ☐ E. filtr przechwytyjący

**57** Dysponujemy następującym fragmentem kodu:

```
10. ${questionNumber}: ${question}
11. <c:forEach var="answer" items="${answer}">
...
16. </c:forEach>
```

Atrybut **question** jest łańcuchem, który może zawierać znaczniki XML-a wymagające wyświetlenia w oknie przeglądarki w formie zwykłego tekstu. W powyższym fragmencie brakuje rozwiązań wymuszających na przeglądarce wyświetlanie znaczników XML-a. Co należałoby zmienić, aby te znaczniki były wyświetlane? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. Zastąpić `${question}` znacznikiem `<c:out value="${question}"/>`.
- ☐ B. Zastąpić `${question}` znacznikiem `<c:out>${question}</c:out>`.
- ☐ C. Zastąpić `${question}` znacznikiem `<c:out escapeXML="true" value="${question}"/>`.
- ☐ D. Zastąpić `${question}` konstrukcją `<%= ${question} %>`.

---

**58** Twoja aplikacja internetowa Javy EE cieszy się dużą popularnością, zatem decydujesz się dodać drugi serwer, aby sprawniej obsługiwać rosnącą liczbę żądań. Które zdania o migracji sesji pomiędzy serwerami są prawdziwe? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. Taka migracja w ramach jednej sesji jest niemożliwa.
- ☐ B. Wraz z migrującą sesją jest przenoszony obiekt **HttpSession**.
- ☐ C. Wraz z migrującą sesją jest przenoszony obiekt **ServletContext**.
- ☐ D. Wraz z migrującą sesją jest przenoszony obiekt **HttpServletRequest**.
- ☐ E. Warunkiem migracji pomiędzy serwerami obiektu dodawanego do sesji za pośrednictwem metody **HttpSession.setAttribute** jest implementowanie przez ten obiekt interfejsu **Serializable**.
- ☐ F. Jeśli dodamy do sesji obiekt za pośrednictwem metody **HttpSession.setAttribute**, jeśli klasa tego obiektu implementuje metody **Serializable.readObject** i **Serializable.writeObject** oraz jeśli sesja podlega migracji, kontener wywoła wspomniane metody **readObject** i **writeObject**.
- ☐ G. Jeśli atrybut sesji implementuje interfejs **HttpSessionActivationListener**, wymagania kontenera ograniczają się do konieczności informowania obiektów nasłuchujących o aktywacji sesji na nowym serwerze.

**59** Deskryptor wdrożenia aplikacji internetowej Javy EE deklaruje kilka filtrów, których adresy URL pasują do bieżącego żądania. Deskryptor deklaruje też kilka filtrów, których znaczniki **<servlet-name>** są dopasowywane do tego samego żądania.

Które zdania o regułach wywoływania przez kontener filtrów dla tego żądania są prawdziwe? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. Zostaną wywołane tylko filtry z pasującymi znacznikami **<servlet-name>**.
- ☐ B. Spośród wszystkich filtrów z pasującymi adresami URL zostanie wywołany tylko pierwszy filtr.
- ☐ C. Spośród wszystkich filtrów z pasującymi znacznikami **<servlet-name>** zostanie wywołany tylko pierwszy filtr.
- ☐ D. Filtry z pasującymi znacznikami **<servlet-name>** zostaną wywołane przed filtrami z pasującymi adresami URL.
- ☐ E. Zostaną wywołane wszystkie filtry z pasującymi adresami URL, ale kolejność tych wywołań jest niezdefiniowana.
- ☐ F. Zostaną wywołane wszystkie filtry z pasującymi adresami URL w kolejności zgodnej z porządkiem ich deklaracji w deskrytorze wdrożenia.

**60** Które zdania o parametrach inicjalizacji serwletu i parametrach inicjalizacji kontekstu są prawdziwe? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. Znaczniki deskryptora wdrożenia deklarujące oba typy parametrów zawierają znaczniki **<param-name>** i **<param-value>**.
- ☐ B. Znaczniki deskryptora wdrożenia deklarujące oba typy parametrów są umieszczane bezpośrednio pod znacznikiem **<web-app>**.
- ☐ C. Metody zwracające wartości parametrów inicjalizacji obu typów nazwano **getInitParameter**.
- ☐ D. Dostęp do parametrów obu typów można uzyskiwać bezpośrednio z poziomu kodu JSP.
- ☐ E. Tylko zmiany parametrów inicjalizacji kontekstu (wprowadzane w deskrytorze wdrożenia) są uwzględniane bez konieczności ponownego wdrożenia aplikacji internetowej.

**61** Programista JSP chce dołączyć zawartość pliku **copyright.jsp** do wszystkich właściwych stron JSP swojej aplikacji.

Które mechanizmy umożliwiają takie dołączenie? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. **<jsp:directive.include file="copyright.jsp" />**
- ☐ B. **<%@ include file="copyright.jsp" %>**
- ☐ C. **<%@ page include="copyright.jsp" %>**
- ☐ D. **<jsp:include page="copyright.jsp" />**
- ☐ E. **<jsp:insert file="copyright.jsp" />**

- 62** Pracujesz nad aplikacją zarządzającą kontami klientów operatora telefonii stacjonarnej, telewizji kablowej i internetu. Znaczna część stron tej aplikacji zawiera funkcjonalność wyszukiwania. Pole wyszukiwanej frazy powinno co prawda wyglądać tak samo na wszystkich stronach, jednak niektóre strony powinny ograniczać zakres przeszukiwania do kont telefonicznych, telewizji kablowej lub internetowych.

Dysponujesz odrębną stroną JSP nazwaną `Search.jsp`:

1. `<form action="/search.go">`
2.     **Znajdź konto** `${param.accountType}`:
3.     `<input type="text" name="searchText"/>`
4.     `<input type="hidden" name="accountType" value="${param.accountType}"/>`
5.     `<input type="submit" value="Szukaj"/>`
6. `</form>`

Którego z poniższych znaczników należałoby użyć w kodzie strony JSP oferującej możliwość wyszukiwania kont telewizji kablowej?

- ☐ A. `<jsp:include page="Search.jsp" accountType="telewizji kablowej"/>`
- ☐ B. `<jsp:include page="Search.jsp">`  
    `<jsp:param name="accountType" value="telewizji kablowej"/>`  
    `</jsp:include>`
- ☐ C. `<jsp:include file="Search.jsp" accountType="telewizji kablowej"/>`
- ☐ D. `<jsp:include file="Search.jsp">`  
    `<jsp:attribute name="accountType" value="telewizji kablowej"/>`  
    `</jsp:include>`

- 
- 63** W czasie testów rozmaitych znaczników i skryptletów programista zdecydował się utworzyć następujący kod JSP:

1. `<% request.setAttribute("name", "świecie"); %>`
2. `<!-- Test -->`
3. `<c:out value='Witaj ${name}'/>`

Ku jego zdziwieniu, w odpowiedzi na żądanie tej strony przeglądarka niczego nie wyświetla. Co programista znajdzie w kodzie źródłowym zwróconej strony HTML?

- ☐ A. `<!-- Test -->`
- ☐ B. `<!-- Test -->`  
    `<c:out value='Witaj ${name}'/>`
- ☐ C. `<!-- Test -->`  
    `<c:out value='Witaj świecie'/>`
- ☐ D. Nie zostaną zwrócone żadne dane wynikowe.

**64** Aplikacja usług randkowych zadaje swoim użytkownikom serie pytań. Przyjmijmy, że istnieje już atrybut zasięgu sesji typu **HashMap** nazwany **compatibilityProfile**, w którym składujemy identyfikatory poszczególnych pytań wraz z udzielonymi odpowiedziami.

Dysponujemy następującym kodem:

```
22. <% ((java.util.HashMap)request.getSession().getAttribute(
 "compatibilityProfile")).put(
23. request.getParameter("questionIdSubmitted"),
24. request.getParameter("answerSubmitted"));
25. %>
```

Jak należałoby zastąpić tę konstrukcję bez stosowania skryptletów? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. `<c:map target="${compatibilityProfile}"`  
`key="${param.questionIdSubmitted}"`  
`value="${param.answerSubmitted}"/>`
- ☐ B. `<jsp:useBean id="compatibilityProfile" class="java.util.HashMap"`  
`scope="session">`  
`<jsp:setProperty name="compatibilityProfile"`  
`property="${param.questionIdSubmitted}"`  
`value="${param.answerSubmitted}"/>`  
`</jsp:useBean>`
- ☐ C. `${compatibilityProfile[param.questionIdSubmitted]} = param.answerSubmitted}`
- ☐ D. `<c:set target="${compatibilityProfile}"`  
`property="${param.questionIdSubmitted}"`  
`value="${param.answerSubmitted}"/>`

- 65 Programista pracuje nad filtrem dla swojej aplikacji internetowej Javy EE. Dysponujemy następującym fragmentem kodu:

```
7. public class MojFiltr implements Filter {
8. public void init(FilterConfig config) throws FilterException { }
9.
10. public void doFilter(HttpServletRequest request,
11. HttpServletResponse response,
12. FilterChain chain)
13. throws IOException, ServletException { }
14.
15. }
```

Która z zaproponowanych poniżej zmian jest niezbędna do utworzenia prawidłowego filtra?  
(Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. Żadne zmiany nie są konieczne.
- ☐ B. Należy dodać metodę **destroy()**.
- ☐ C. Należy zmienić ciało metody **doFilter()**.
- ☐ D. Należy zmienić sygnaturę metody **init()**.
- ☐ E. Należy zmienić argumenty metody **doFilter()**.
- ☐ F. Należy zmienić listę wyjątków generowanych przez metodę **doFilter()**.

- 66 Twoja firma chce dodać do istniejącej aplikacji stronę powitalną nazwaną **SplashAd.jsp**, która będzie reklamowała użytkownikom odwiedzającym daną witrynę oferty innych przedsiębiorstw. Na stronie powitalnej użytkownicy będą mieli do dyspozycji pole wyboru *Nie wyświetlaj więcej tej oferty* oraz przycisk *Przejdź do mojego konta*. Jeśli użytkownik wyśle ten formularz z zaznaczonym polem wyboru, docelowy serwet ustawi cookie (znacznik kontekstu klienta) nazwane **skipSplashAd**, po czym zwróci sterowanie do głównej strony JSP.

Główna strona JSP będzie odpowiedzialna za skierowanie danego żądania na stronę powitalną. Jaki fragment kodu należy dodać na początek strony głównej, aby odsyłała strony powitalne tym użytkownikom, którzy jeszcze nie zaznaczyli odpowiedniego pola wyboru?

- ☐ A. `<c:if test="${empty cookie.skipSplashAd and pageContext.session.new}">  
 <jsp:forward page="SplashAd.jsp"/>  
</c:if>`
- ☐ B. `<jsp:forward page="SplashAd.jsp" flush="${empty cookie.  
skipSplashAd}"/>`
- ☐ C. `<jsp:redirect page="SplashAd.jsp"/>`
- ☐ D. `<jsp:redirect file="SplashAd.jsp"/>`
- ☐ E. `<% if(cookie.get("skipSplashAd") == null && session.isNew()){ %>  
 <jsp:forward page="SplashAd.jsp"/>  
 <% } %>`

**67** Programista chce zaimplementować interfejs `ServletContextListener`. Dysponujemy następującym fragmentem deskryptora wdrożenia:

```

101. <!-- tutaj wstaw znacznik1 -->
102. <param-name>mojParametr</param-name>
103. <param-value>mojaWartosc</param-value>
104. <!-- tutaj zamknij znacznik1 -->
105. <listener>
106. <!-- tutaj wstaw znacznik2 -->
107. com.wickedlysmart.MojaKlasaNasluchujaca
108. <!-- tutaj zamknij znacznik2 -->
109. </listener>

```

Poniżej przedstawiono pseudokod samej klasy nasłuchującej:

```

5. // tutaj powinna się znaleźć deklaracja pakietu i instrukcje importu
6. public class MojaKlasaNasluchujaca implements ServletContextListener {
7. // tutaj wstaw metodę metoda1
8. // tutaj zakończ odpowiednią metodę
9. }

```

Które zdania są prawdziwe? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. Fragment deskryptora wdrożenia w tej formie nie może być prawidłowy.
- ☐ B. Znacznikiem `znacznik1` powinien być `<context-param>`.
- ☐ C. Znacznikiem `znacznik1` powinien być `<servlet-param>`.
- ☐ D. Znacznikiem `znacznik2` powinien być `<listener-class>`.
- ☐ E. Znacznikiem `znacznik2` powinien być `<servlet-context-class>`.
- ☐ F. Metodą `metoda1` powinna być `initializeListener`.
- ☐ G. Metodą `metoda1` powinna być `contextInitialized`.

- 68** Witryna internetowa *wickedlysmart.com* udostępnia prawidłowo wdrożoną aplikację internetową Javy EE. Deskryptor wdrożenia tej aplikacji zawiera następujący fragment:

```
<welcome-file-list>
 <welcome-file>welcome.html</welcome-file>
 <welcome-file>howdy.html</welcome-file>
 <welcome-file>index.html</welcome-file>
</welcome-file-list>
```

Poniżej przedstawiono fragment struktury katalogów tej aplikacji:

**MojaAplikacjaInternetowa**

```
|
|-- index.html
|
|-- welcome
| |-- welcome.html
|
|-- foobar
| | howdy.html
```

Załóżmy, że do opisywanej aplikacji trafiają dwa następujące żądania:

**<http://www.wickedlysmart.com/MojaAplikacjaInternetowa/foobar>**

**<http://www.wickedlysmart.com/MojaAplikacjaInternetowa>**

Która z wymienionych sekwencji odpowiedzi zostanie zwrócona?

- ☐ A. **howdy.html** i kod **404**
- ☐ B. **index.html** i kod **404**
- ☐ C. **welcome.html** i kod **404**
- ☐ D. **howdy.html** i **index.html**
- ☐ E. **index.html** i **index.html**
- ☐ F. **howdy.html** i **welcome.html**
- ☐ G. **welcome.html** i **index.html**

**69** Twoja aplikacja internetowa obejmuje prawidłowy deskryptor wdrożenia z pojedynczym znacznikiem **<security-constraint>**. W ciele tego znacznika zadeklarowano:

– pojedynczą metodę **GET** protokołu HTTP.

Wszystkie zasoby Twojej aplikacji umieszczono w katalogach **katalog1** i **katalog2**, a w deskrytorze wdrożenia zdefiniowano tylko dwie role: **NOWICJUSZ** oraz **EKSPERT**.

Które zdania prawidłowo opisują adresy URL i znaczniki ról niezbędne do ograniczenia dostępu do zasobów w katalogu **katalog2** użytkowników przypisanych do roli **NOWICJUSZ**? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. Pojedynczy znacznik adresu URL powinien deklarować katalog **katalog1**, a pojedynczy znacznik roli powinien deklarować rolę **EKSPERT**.
- ☐ B. Pojedynczy znacznik adresu URL powinien deklarować katalog **katalog2**, a pojedynczy znacznik roli powinien deklarować rolę **EKSPERT**.
- ☐ C. Pojedynczy znacznik adresu URL powinien deklarować katalog **katalog1**, a pojedynczy znacznik roli powinien deklarować rolę **NOWICJUSZ**.
- ☐ D. Pojedynczy znacznik adresu URL powinien deklarować katalog **katalog2**, a pojedynczy znacznik roli powinien deklarować rolę **NOWICJUSZ**.
- ☐ E. Jeden znacznik adresu URL powinien deklarować wartość **ANY**, a jego znacznik roli powinien deklarować rolę **EKSPERT**. Drugi znacznik adresu URL powinien deklarować katalog **katalog2**, a jego znacznik roli powinien deklarować rolę **NOWICJUSZ**.
- ☐ F. Jeden znacznik adresu URL powinien deklarować oba katalogi, a jego znacznik roli powinien deklarować rolę **EKSPERT**. Drugi znacznik adresu URL powinien deklarować katalog **katalog1**, a jego znacznik roli powinien deklarować rolę **NOWICJUSZ**.



BAR  
KAWOWY

## Końcowy egzamin próbny. Odpowiedzi

- 1 Programista prawidłowo skonfigurował strukturę katalogów dla swojej aplikacji internetowej w technologii Java EE. Aplikacja nosi nazwę MojaAplikacjaInternetowa. W których katalogach można umieścić plik **mojZnacznik.tag**, aby był dostępny dla kontenera? (Zaznacz wszystkie odpowiedzi). jsp 8., hf 608.
- ☐ A. MojaAplikacjaInternetowa/WEB-INF
  - ☐ B. MojaAplikacjaInternetowa/META-INF
  - ☐ C. MojaAplikacjaInternetowa/WEB-INF/lib
  - ☒ D. MojaAplikacjaInternetowa/WEB-INF/tags -- Odpowiedzi D i F: pliki znaczników MUSZĄ się znajdować w katalogu tags lub jego podkatalogu
  - ☐ E. MojaAplikacjaInternetowa/WEB-INF/TLDs
  - ☒ F. MojaAplikacjaInternetowa/WEB-INF/tags/mojeZnaczniki
- 
- 2 Które z poniższych wyrażeń są prawidłowymi konstrukcjami języka EL? Specyfikacja JSP 2.0, punkt 2.3.5, hf 396.  
(Zaznacz wszystkie prawidłowe odpowiedzi).
- ☒ A. `${"1" + "2"}` – Odpowiedź A: zarówno tańcuch "1", jak i tańcuch "2" można przekonwertować na wartości typu Long, zatem otrzymamy wynik równy 3.
  - ☐ B. `${1 plus 2}` – Odpowiedź B: plus nie jest operatorem języka EL.
  - ☒ C. `${1 eq 2}` – Odpowiedź C jest prawidłowa — na wyjściu otrzymamy wartość false.
  - ☒ D. `${2 div 1}` – Odpowiedź D jest prawidłowa — na wyjściu otrzymamy wartość 2,0.
  - ☐ E. `${2 & 1}` – Odpowiedź E jest błędna — & nie jest prawidłowym operatorem języka EL (w przeciwieństwie do operatorów && oraz and).
  - ☐ F. `${"head"+"first"}` – Odpowiedź F: nie można konkatelować tańcuchów za pomocą operatora +. Język EL nie przekształca użytych wartości tańcuchowych w wartość typu Double.

- 3 Plik TLD opracowany z myślą o witrynie internetowej z forum dyskusyjnym poświęconym Javie zawiera następującą definicję znacznika:

*Specyfikacja JSP 2.0, podpunkt 7.4.1.1, hf 476 – 480.*

```
<tag>
 <name>avatar</name>
 <tag-class>hf.AvatarTagHandler</tag-class>
 <body-content>empty</body-content>

 <attribute>
 <name>userId</name>
 <required>true</required>
 <rtexprvalue>true</rtexprvalue>
 </attribute>
 <attribute>
 <name>size</name>
 <required>false</required>
 <rtexprvalue>false</rtexprvalue>
 </attribute>
</tag>
```

Które zdania na temat klasy **AvatarTagHandler** są prawdziwe, jeśli przyjąć, że klasa ta rozszerza klasę **SimpleTagHandler** i że generuje kod HTML-a wyświetlający obraz awatara użytkownika? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☒ A. Klasa **AvatarTagHandler** powinna definiować składową **size**, dla której musi istnieć przynajmniej metoda ustawiająca.
- ☐ B. Zmienna **size** nie jest konieczna, ponieważ plik TLD mówi wyraźnie, że odpowiedni atrybut ma charakter opcjonalny.
- ☒ C. Klasa **AvatarTagHandler** musi nadpisywać metodę cyklu życia **doTag**.
- ☐ D. Klasa **AvatarTagHandler** musi nadpisywać metodę cyklu życia **doStartTag**.
- ☐ E. Klasa **AvatarTagHandler** musi przeciążać wszystkie implementowane metody cyklu życia wersjami otrzymującymi po jednym dodatkowym parametrze dla każdego atrybutu zdefiniowanego w pliku TLD. W tym przypadku będzie konieczny tylko jeden taki parametr.

– Odpowiedź A: klasa obsługująca znacznik powinna składać rozmiar, mimo że sam znacznik nie zawsze tego wymaga.

– Odpowiedź C nie rozwiązuje problemu, chyba że w nadpisanej metodzie **doTag** zaimplementujemy odpowiednie zachowania. Domyślna implementacja tej metody w klasie **SimpleTagSupport** co prawda istnieje, ale nie podejmuje żadnych działań.

– Odpowiedź D: metodę **doStartTag** stosuje się w klasycznych klasach obsługi znaczników.

– Odpowiedź E: klasyczne klasy obsługi znaczników zawierają tylko jedną metodę cyklu życia, a przeciążone wersje tej metody nie będą rozpoznawane przez kontener.

- 4** Serwlet tworzy komponent i ustawia jego właściwości przed przekazaniem żądania do strony JSP.

Specyfikacja JSP 2.0, podrozdziały 5.0 – 5.1, hf 350 – 363.

Poniżej przedstawiono odpowiedni fragment kodu tego serwletu:

```
20. foo.User user = new foo.User();
21. user.setFirst(request.getParameter("firstName"));
22. user.setLast(request.getParameter("lastName"));
23. user.setStreet(request.getParameter("streetAddress"));
24. user.setCity(request.getParameter("city"));
25. user.setState(request.getParameter("state"));
26. user.setZipCode(request.getParameter("zipCode"));
27. request.setAttribute("user", user);
```

Który z poniższych fragmentów kodu umieszczony w kodzie strony JSP mógłby skutecznie zastąpić przedstawiony kod serwletu? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. `<jsp:useBean id="user" type="foo.User" scope="request"/>`
- ☐ B. `<jsp:useBean id="user" type="foo.User" scope="request">`  
`<jsp:setProperty name="user" property="*" />`  
`</jsp:useBean>`
- ☒ C. `<jsp:useBean id="user" class="foo.User" scope="request">`  
`<jsp:setProperty name="user" property="first" param="firstName" />`  
`<jsp:setProperty name="user" property="last" param="lastName" />`  
`<jsp:setProperty name="user" property="street" param="streetAddress" />`  
`<jsp:setProperty name="user" property="city" />`  
`<jsp:setProperty name="user" property="state" />`  
`<jsp:setProperty name="user" property="zipCode" />`  
`</jsp:useBean>`
- ☒ D. `<jsp:useBean id="user" class="foo.User" scope="request">`  
`<jsp:setProperty name="user" property="*" />`  
`<jsp:setProperty name="user" property="first" param="firstName" />`  
`<jsp:setProperty name="user" property="last" param="lastName" />`  
`<jsp:setProperty name="user" property="street" param="streetAddress" />`  
`</jsp:useBean>`

– W odpowiedziach A i B wykorzystano atrybut `type`, który wymaga uprzedniego zapisania danego komponentu w jakimś zasięgu. Nawet użycie atrybutu `class` nie wystarczyłoby do wypełnienia wszystkich właściwości tego komponentu.

– Odpowiedzi C i D: poszczególne znaczniki `<jsp:setProperty>` są niezbędne do prawidłowego odwzorowania parametrów na właściwości komponentu w sytuacji, gdy ich nazwy różnią się od siebie. Pasujące nazwy parametrów można automatycznie przekazywać do komponentu za pomocą konstrukcji `property="*" />`.

- 5 Które zdania opisujące korzyści, ograniczenia i zastosowania obiektu delegata biznesowego i obiektu lokalizatora usługi są prawdziwe? (Zaznacz wszystkie prawidłowe odpowiedzi). *core j2ee 302, 315, hf 760 – 761.*
- ☐ A. Oba obiekty równie często wykonują wywołania sieciowe. *– Odpowiedź A: obiekt delegata biznesowego z reguły prosi inny obiekt o wykonanie wywołania sieciowego.*
  - ☐ B. Oba obiekty równie często wywołują metody obiektu transferu. *– Odpowiedź B: obiekt lokalizatora usługi z reguły nie korzysta z pomocy obiektu transferu.*
  - ☐ C. Oba obiekty są równie często wywoływane przez obiekt kontrolera. *– Odpowiedź C: kontroler z reguły kieruje żądania do obiektu delegata biznesowego, a w razie konieczności sam delegat biznesowy kieruje żądania do lokalizatora usługi.*
  - ☒ D. Z perspektywy delegata biznesowego obiekt lokalizatora pełni funkcję serwera.
  - ☒ E. Jeśli oba obiekty zaimplementujemy z wykorzystaniem pamięci podręcznej, brak aktualizacji danych będzie poważniejszym problemem w przypadku delegata biznesowego.
- 6 Które zdania o procesie tworzenia obiektów nasłuchujących sesji są prawdziwe? (Zaznacz wszystkie prawidłowe odpowiedzi). *Specyfikacja serwetów, dodatek B; hf 256 – 263.*
- ☐ A. Wszystkie te obiekty są deklarowane w deskrytorze wdrożenia. *– Odpowiedź A: obiektu HttpSessionBindingListener nie deklaruje się w deskrytorze wdrożenia.*
  - ☒ B. Nie wszystkie obiekty tego typu muszą być deklarowane w deskrytorze wdrożenia.
  - ☒ C. Do ich deklarowania w deskrytorze wdrożenia służy znacznik **<listener>**. *– Odpowiedź C: mamy nadzieję, że potrafisz tę odpowiedź wykluczyć, nie ucząc się tego na pamięć.*
  - ☐ D. Do ich deklarowania w deskrytorze wdrożenia służy znacznik **<session-listener>**.
  - ☒ E. Do ich deklarowania w deskrytorze wdrożenia służy znacznik umieszczany w znaczniku **<web-app>**.
  - ☐ F. Do ich deklarowania w deskrytorze wdrożenia służy znacznik umieszczany w znaczniku **<servlet>**. *– Odpowiedź F: pamiętaj, że sesje mogą się rozciągać na wiele serwetów.*
- 7 Niektórzy użytkownicy skarżą się na dziwne zdarzenia, które mają miejsce, kiedy na jednym komputerze dwa okna przeglądarki internetowej jednocześnie uzyskują dostęp do tej samej aplikacji internetowej. Chcesz przetestować różne przeglądarki pod kątem możliwości współdzielenia sesji przez wiele okien. Decydujesz się na wypisanie na stronie JSP identyfikatora **JSESSIONID**. Jak można osiągnąć ten cel, zakładając, że w testowanej przeglądarce włączono obsługę cookies? (Zaznacz wszystkie prawidłowe odpowiedzi). *Specyfikacja JSP 2.0, punkt 2.2.3; specyfikacja serwetów 2.4, punkt 7.1.1; hf 232 i 390.*
- ☐ A. `${cookie.JSESSIONID}` *– Odpowiedź A zwróci obiekt Cookie, który wyświetli referencję do odpowiedniego obiektu zamiast jego wartości wewnętrznej.*
  - ☒ B. `${cookie.JSESSIONID.value}`
  - ☒ C. `${cookie["JSESSIONID"]["value"]}` *– Odpowiedzi B, C, D i E: obiekt domyślny cookie języka EL reprezentuje mapę obiektów Cookie. Wszystkie te wyrażenia uzyskują identyfikator JSESSIONID obiektu Cookie i wywołują jego metodę getValue().*
  - ☒ D. `${cookie.JSESSIONID["value"]}`
  - ☒ E. `${cookie["JSESSIONID"].value}` *– Odpowiedź F: cookieValues nie jest obiektem domyślnym języka EL.*
  - ☐ F. `${cookieValues[0].value}`

- 8 Który obiekt domyślny zapewnia dostęp do atrybutów obiektu **ServletContext**? *Specyfikacja JSP 2.0, punkt 1.8.3.*
- ☐ A. **server**
  - ☐ B. **context**
  - ☐ C. **request** *– Odpowiedź C jest błędna, ponieważ obiekt domyślny request ma dostęp tylko do atrybutów zasięgu żądania.*
  - ☒ D. **application** *– Odpowiedź D jest prawidłowa. Obiekt domyślny application jest równoważny obiektowi ServletContext.*
  - ☐ E. **servletContext**
- Odpowiedzi A, B i E są błędne, ponieważ użyte nazwy nie reprezentują prawidłowych obiektów domyślnych JSP.*

- 9 Które z poniższych metod zdefiniowano w klasie **HttpServlet**? *Specyfikacja HTTP 1.1; hf rozdział 4.*  
(Zaznacz wszystkie prawidłowe odpowiedzi).
- ☒ A. **doGet**
  - ☒ B. **doTrace**
  - ☐ C. **doError** *– Odpowiedź C: nie istnieje metoda ERROR protokołu HTTP.*
  - ☐ D. **doConnect** *– Odpowiedź D: co prawda istnieje metoda CONNECT protokołu HTTP, jednak jej wyjątkowość polega na braku odpowiedniej metody w klasie HttpServlet.*
  - ☒ E. **doOptions**

- 10 Doszedłeś do wniosku, że niektóre funkcje Twojej aplikacji internetowej będą wymagały od użytkowników przejścia przez proces rejestracji. Co więcej, Twoja aplikacja musi operować na poufnych danych użytkowników, które należy chronić. *hf 677 – 684.*
- Które zdania są prawdziwe? (Zaznacz wszystkie prawidłowe odpowiedzi).
- ☐ A. Możesz zabezpieczać przesyłane dane dopiero po tym, jak Twoja aplikacja zweryfikuje hasło użytkownika.
  - ☐ B. Spośród rozmaitych metod uwierzytelniania obsługiwanych przez kontenery zgodne ze specyfikacją Java EE tylko techniki **BASIC**, **DIGEST** i **FORM** zaimplementowano przez dopasowywanie nazwy i hasła użytkownika.
  - ☒ C. Niezależnie od typu stosowanego mechanizmu uwierzytelniania Javy EE, jego aktywacja następuje dopiero w reakcji na żądanie chronionego zasobu.
  - ☐ D. Wszystkie metody uwierzytelniania specyfikacji Java EE zapewniają bezpieczeństwo danych bez konieczności samodzielnego implementowania jakichkolwiek dodatkowych zabezpieczeń.

- 11 Dysponujemy następującymi fragmentami kodu zaczerpniętymi z pojedynczego znacznika w ramach deskryptora wdrożenia zgodnego ze specyfikacją Java EE:

*Specyfikacja serwletów, rozdział 12.; hf 684.*

```

343. <web-resource-collection>
344. <web-resource-name>Przepisy</web-resource-name>
345. <url-pattern>/Piwo/Aktualizuj/*</url-pattern>
346. <http-method>POST</http-method>
347. </web-resource-collection>
...
367. <auth-constraint>
368. <role-name>Członek</role-name>
369. </auth-constraint>
...
385. <user-data-constraint>
386. <transport-guarantee>CONFIDENTIAL</transport-guarantee>
387. </user-data-constraint>

```

Które zdania są prawdziwe? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☒ A. Deskryptor wdrożenia zgodny ze specyfikacją Java EE może zawierać pojedynczy znacznik, w którym mogą się znajdować wszystkie te znaczniki.
- ☐ B. Przedstawiony powyżej pojedynczy znacznik może zawierać więcej znaczników **<auth-constraint>**.
- ☐ C. Przedstawiony powyżej pojedynczy znacznik może zawierać więcej znaczników **<user-data-constraint>**.  
*– Odpowiedź C: prawidłowy znacznik <security-constraint> może deklarować tylko jeden typ integralności danych.*
- ☒ D. Pojedynczy znacznik **<web-resource-collection>** może zawierać więcej znaczników **<url-pattern>**.
- ☒ E. Inne znaczniki tego samego typu co powyższy (niewidoczny) znacznik mogą zawierać identyczny znacznik **<url-pattern>**.
- ☐ F. Przedstawiony powyżej znacznik deklaruje zasady uwierzytelniania, autoryzacji i integralności danych na potrzeby odpowiedniej aplikacji internetowej.

- 12** Pracujesz nad dokumentem JSP, który ma generować dynamiczny obraz SVG reprezentowany przez strukturę dokumentu w formacie XML. Twój dokument JSP musi deklarować nagłówek odpowiedzi HTTP **'Content-Type'** jako **'image/svg+xml'**, aby przeglądarka internetowa właściwie wyświetliła odpowiedź w formie obrazu SVG.

Specyfikacja JSP 2.0, podrozdział 1.1.

Który fragment kodu strony JSP deklaruje, że dany dokument JSP jest odpowiedzią SVG?

- ☐ A. `<%@ page contentType='image/svg+xml' %>` – Odpowiedź A jest błędna, ponieważ standardowej składni dyrektywy JSP `<%@ ... %>` nie można stosować w dokumentach JSP.
- ☐ B. `<jsp:page contentType='image/svg+xml' />` – Odpowiedź B jest błędna, ponieważ w dokumentach JSP nie istnieje standardowy znacznik `jsp:page`.
- ☒ C. `<jsp:directive.page contentType='image/svg+xml' />` – Odpowiedź C jest prawidłowa, ponieważ znacznik standardowy `jsp:directive.page` można stosować w dokumentach JSP.
- ☐ D. `<jsp:page.contentType>image/svg+xml</jsp:page.contentType>` – Odpowiedź D jest błędna, ponieważ w dokumentach JSP nie istnieje standardowy znacznik `jsp:page.contentType`.

- 13** Dysponujemy stroną JSP z następującym wierszem:

Specyfikacja JSP 2.0, punkt 1.5.2; hf 304.

```
<!-- out.print("Witaj świecie"); -->
```

Co znajdzie się w wynikowym kodzie HTML?

- ☐ A. Witaj świecie
- ☐ B. `out.print("Witaj świecie");`
- ☐ C. `<!-- Witaj świecie -->`
- ☒ D. Przedstawiony wiersz nie wygeneruje żadnych danych wyjściowych.

- 14** Które z poniższych zdań o obsłudze sesji protokołu HTTP są prawdziwe? (Zaznacz wszystkie prawidłowe odpowiedzi).

Specyfikacja serwletów, rozdział 7.; hf 231 – 240.

- ☒ A. Kontenery zgodne ze specyfikacją Java EE muszą obsługiwać ciasteczka HTTP.
- ☐ B. Kontenery zgodne ze specyfikacją Java EE muszą obsługiwać przepisywanie adresów URL. – Odpowiedź B: przepisywanie adresów URL jest niemal zawsze stosowane w roli mechanizmu zastępczego w sytuacji, gdy cookies nie są dostępne, ale specyfikacja NIE nakłada takiego obowiązku na kontenery.
- ☒ C. Kontenery zgodne ze specyfikacją Java EE muszą obsługiwać protokół Secure Sockets Layer (SSL).
- ☒ D. Kontenery zgodne ze specyfikacją Java EE muszą obsługiwać sesje HTTP, nawet jeśli klient nie obsługuje ciasteczek.
- ☐ E. Kontenery zgodne ze specyfikacją Java EE muszą rozpoznawać sygnał końca HTTP wskazujący na wyczerpanie aktywności sesji klienta. – Odpowiedź E: protokół HTTP nie obsługuje sygnału końca sesji.

- 15 Twoja firma zakupiła licencję umożliwiającą stosowanie w budowanych aplikacjach specjalnej biblioteki języka JavaScript stworzonej z myślą o konstruowaniu złożonych menu. Twój zespół długo nie mógł opanować sztuki korzystania z tej biblioteki, a wskutek popełnionych błędów wszyscy użytkownicy mieli dostęp do elementów menu, które powinny być widoczne tylko dla uprawnionych, wcześniej uwierzytelnionych ról. Biblioteka znaczników niestandardowych z prostymi obiektami obsługi powinna uchronić programistów przed błędami składniowymi w kodzie języka JavaScript i jednocześnie zapewnić niezbędne mechanizmy zabezpieczeń.

hf 570 – 573.

Po spotkaniu poświęconym rozwojowi projektu Twój szef opracował dokument, zgodnie z którym przykładowa deklaracja menu powinna mieć następującą postać:

```
<menu:main>
 <menu:headItem text="Moje konto" url="/mojeKonto.do"/>
 <menu:headItem text="Transakcje">
 <menu:subItem text="Przychodzące" url="/przychodzaceTr.do"/>
 <menu:subItem text="Wychodzące" url="/wychadzaceTr.do"/>
 <menu:subItem text="Oczekujące" url="/oczekujaceTr.do"
 requireRole="ksiegowy"/>
 </menu:headItem>
 <menu:headItem text="Administracja" url="/administracja.do"
 requireRole="administrator"/>
</menu:main>
```

Chciałbyś, aby pełna odpowiedzialność za generowanie kodu wynikowego spadła na klasę obsługującą zewnętrzny znacznik `<menu:main>`, ponieważ zakładasz, że centralizacja logiki wyświetlania menu ułatwi jej przyszłe utrzymanie. Klasa obsługi zewnętrznego znacznika będzie oczywiście wymagała dostępu do znaczników wewnętrznych. Które z zaproponowanych poniżej rozwiązań wydaje Ci się najlepsze?

- ☒ A. Każdy znacznik wewnętrzny powinien się rejestrować w swoim bezpośrednim znaczniku macierzystym. Znacznik macierzysty może składać informacje o swoich bezpośrednich potomkach w uporządkowanej kolekcji.
- ☐ B. Każdy znacznik wewnętrzny powinien się rejestrować w klasie obsługi znacznika zewnętrznego, która może składać informacje o wszystkich (pośrednich i bezpośrednich) znacznikach potomnych w jednej strukturze typu **HashSet**.
- ☐ C. W przeciwieństwie do znaczników klasycznych, klasa **SimpleTagSupport** definiuje metody **findDescendentWithClass()** i **getChildren()**, które zapewniają głównemu znacznikowi zewnętrznemu pełen dostęp do jego potomków bez konieczności samodzielnego tworzenia dodatkowych mechanizmów.
- ☐ D. Każdy znacznik wewnętrzny powinien zapisać sam siebie w formie atrybutu zasięgu strony, gdzie wartość **text** będzie pełniła funkcję klucza tego atrybutu.

– Odpowiedź A reprezentuje najprostsze rozwiązanie, ponieważ tworzy prostą strukturę drzewiastą znaczników zapewniającą znacznikowi `<menu:main>` dostęp do wszystkich jego znaczników potomnych.

– Odpowiedzi B i D nie dają znacznikowi zewnętrznemu żadnych wskazówek o strukturze znaczników wewnętrznych.

– Odpowiedź C: wymienione metody w ogóle nie istnieją. Interfejs API oferuje tylko metody `findAncestorWithClass()` i `getParent()`.

- 16** Która faza cyklu życia strony JSP może spowodować zwrócenie kodu błędu 500 protokołu HTTP w odpowiedzi na żądanie danej strony? (Zaznacz wszystkie prawidłowe odpowiedzi). *Specyfikacja JSP 2.0, podrozdział 1.1.*
- ☒ A. Kompilacja strony JSP. *– Odpowiedź A jest prawidłowa, ponieważ w razie niepowodzenia kompilacji kodu serwetu wygenerowanego na podstawie kodu strony JSP kontener musi wygenerować błąd serwera.*
  - ☒ B. Wykonanie metody obsługi żądania. *– Odpowiedź B jest prawidłowa, ponieważ wszelkie wyjątki czasu wykonywania generowane przez kod JSP muszą być obsługiwane przez kontener, który ma wówczas obowiązek wygenerować błąd serwera.*
  - ☐ C. Wykonanie metody niszczenia strony. *– Odpowiedź C jest błędna, ponieważ metoda destroy nie może spowodować błędu 500.*
  - ☒ D. Wykonanie metody inicjalizującej. *– Odpowiedź D jest prawidłowa, ponieważ w razie wygenerowania wyjątku przez metodę inicjalizującą kontener nie może przekazać żądania do kodu JSP — musi wówczas wystąpić błąd serwera.*
- 17** Jeśli przyjmimy, że **session** jest referencją do prawidłowego obiektu **HttpSession**, a **"mojAtrybut"** jest nazwą obiektu skojarzonego z tą referencją, którego wywołania należałoby użyć do usunięcia skojarzenia tych obiektów? (Zaznacz wszystkie prawidłowe odpowiedzi). *API, hf rozdział 6.*
- ☐ A. `session.unbind();`
  - ☒ B. `session.invalidate();`
  - ☐ C. `session.unbind("mojAtrybut");`
  - ☐ D. `session.remove("mojAtrybut");`
  - ☐ E. `session.invalidate("mojAtrybut");` *– Odpowiedź E: metoda invalidate() służy do usuwania skojarzeń wszystkich obiektów związanych z daną sesją.*
  - ☒ F. `session.removeAttribute("mojAtrybut");` *– Odpowiedź F: metoda removeAttribute() służy do usuwania skojarzeń pojedynczych obiektów.*
  - ☐ G. `session.unbindAttribute("mojAtrybut");`
- 18** Jeśli **req** jest referencją do obiektu **HttpServletRequest** i jeśli bieżąca sesja nie istnieje, które zdania o metodzie **req.getSession()** są prawdziwe? (Zaznacz wszystkie prawidłowe odpowiedzi). *API, hf 232 – 233.*
- ☐ A. Wywołanie metody **req.getSession()** zwróci wartość **null**. *– Odpowiedzi A i B: w obu przypadkach zostanie utworzona nowa sesja.*
  - ☐ B. Wywołanie metody **req.getSession(true)** zwróci wartość **null**.
  - ☒ C. Wywołanie metody **req.getSession(false)** zwróci wartość **null**.
  - ☒ D. Wywołanie metody **req.getSession()** zwróci nową sesję.
  - ☒ E. Wywołanie metody **req.getSession(true)** zwróci nową sesję.
  - ☐ F. Wywołanie metody **req.getSession(false)** zwróci nową sesję.

- 19 Dotychczasowy kod aplikacji obejmuje między innymi klasyczną klasę obsługi znacznika. Autor tej klasy stworzył mechanizm przetwarzania ciała znacznika sto razy, aby za jej pomocą można było testować pozostałe znaczniki generujące losową zawartość.

TagSupport API;  
specyfikacja JSP 2.0,  
podrozdział 13.1; hf  
536 – 537.

Mamy następujący kod:

```
06. public class HundredTimesTag extends TagSupport {
07. private int iterationCount;
08. public int doTag() throws JspException {
09. iterationCount = 0;
10. return EVAL_BODY_INCLUDE;
11. }
12.
13. public int doAfterBody() throws JspException {
14. if (iterationCount < 100) {
15. iterationCount++;
16. return EVAL_BODY_AGAIN;
17. } else {
18. return SKIP_BODY;
19. }
20. }
21. }
```

Jaki błąd popełniono w tym kodzie?

- ☐ A. Klasy obsługi znaczników nie gwarantują bezpieczeństwa przetwarzania wielowątkowego, zatem zmienna **iterationCount** może zawierać nieprawidłową wartość, jeśli wielu użytkowników jednocześnie zażąda danej strony.
- ☐ B. Metoda **doAfterBody** nigdy nie zostanie wywołana, ponieważ nie jest częścią cyklu życia klasy obsługi znacznika. Włączenie tej metody do cyklu życia wymagałoby od programisty rozszerzenia klasy **IterationTagSupport**.
- ☒ C. Metodę **doTag** należałoby zastąpić metodą **doStartTag**. W powyższym kodzie jest wywoływana domyślna wersja metody **doStartTag** klasy **TagSupport**, która zwraca wartość **SKIP\_BODY**, co z kolei powoduje, że metoda **doAfterBody** nigdy nie jest wywoływana.
- ☐ D. Kiedy metoda **doAfterBody** zwraca wartość **EVAL\_BODY\_AGAIN**, następuje ponowne wywołanie metody **doTag**. Metoda **doTag** przywraca wartość 0 zmiennej **iterationCount**, zatem cała pętla jest nieskończona, co ostatecznie prowadzi do wygenerowania wyjątku **java.lang.OutOfMemoryError**.

– Odpowiedź A: klasy obsługi znaczników gwarantują bezpieczeństwo przetwarzania wielowątkowego, zatem składowanie stanu w obiektach tych klas nie stanowi problemu.

– Odpowiedź B: *IterationTagSupport* nie jest prawdziwą klasą. Metoda *doAfterBody* jest częścią interfejsu *IterationTag* implementowanego przez klasę *TagSupport*.

– Odpowiedź C: prosta zmiana tej metody powinna rozwiązać problem. Jeśli dany projekt korzysta z platformy Java 5 SE, warto rozważyć użycie adnotacji *@Override* dla tych metod cyklu życia, aby wyeliminować ryzyko popełnienia podobnych błędów.

– Odpowiedź D: zmiana nazwy metody z odpowiedzi C nie doprowadzi do wystąpienia nieskończonej pętli, ponieważ cykl życia klasycznych znaczników nigdy nie obejmuje wywołania metody *doStartTag* więcej niż raz.

20 Mamy dany następujący fragment deskryptora wdrożenia aplikacji internetowej:

API, hf 244 – 245.

```
72. <session-config>
73. <session-timeout>10</session-timeout>
74. </session-config>
```

Przyjmijmy, że **session** jest referencją do prawidłowego obiektu **HttpSession**.  
Nasz serwet zawiera następujący wiersz:

```
30. session.setMaxInactiveInterval(120);
```

Które zdanie prawidłowo opisuje sytuację po wykonaniu wiersza 30?  
(Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. Przedstawiony fragment deskryptora wdrożenia jest błędny.
- ☐ B. Wywołanie metody **setMaxInactiveInterval** zmodyfikuje wartość zadeklarowaną w znaczniku **<session-timeout>**.  
– Odpowiedź B: metoda **setMaxInactiveInterval** przykrywa tylko ustawienia limitu czasowego dla danej sesji.
- ☐ C. Na podstawie przedstawionych fragmentów nie można określić limitu czasowego sesji.
- ☐ D. Jeśli w ramach danej sesji kontener nie otrzyma od klienta żadnego żądania przez 2 godziny, sesja zostanie unieważniona.
- ☒ E. Jeśli w ramach danej sesji kontener nie otrzyma od klienta żadnego żądania przez 2 minuty, sesja zostanie unieważniona.  
– Odpowiedź E: argument tej metody reprezentuje sekundy, natomiast wartość zadeklarowana w znaczniku jest wyrażona w minutach.
- ☐ F. Jeśli w ramach danej sesji kontener nie otrzyma od klienta żadnego żądania przez 10 sekund, sesja zostanie unieważniona.
- ☐ G. Jeśli w ramach danej sesji kontener nie otrzyma od klienta żadnego żądania przez 10 minut, sesja zostanie unieważniona.

21 Utworzyłeś prawidłową strukturę katalogów i poprawny plik **WAR** dla swojej aplikacji internetowej napisanej w technologii Java EE. Jeśli wiemy, że:

Specyfikacja serwetów, rozdział 9.; hf 612.

- plik **WAR** nazwano **PoprawnaAplicacja.war**;
- **WARdir** reprezentuje katalog, który musi istnieć w każdym pliku **WAR**;
- **APPdir** reprezentuje katalog, który musi istnieć w każdej aplikacji internetowej,

które zdanie jest prawdziwe?

- ☐ A. Określenie rzeczywistej nazwy katalogu **WARdir** NIE jest możliwe.  
– Odpowiedź A: ten katalog musi nosić nazwę **META-INF**.
- ☒ B. Określenie nazwy aplikacji NIE jest możliwe.  
– Odpowiedź B: kontener z reguły nazywa aplikację tak, jak nazwano odpowiedni plik **WAR**, chociaż takie zachowanie nie jest wymuszane przez specyfikację.
- ☐ C. W opisywanej strukturze katalogów katalog **APPdir** będzie się znajdował wewnątrz katalogu **WARdir**.
- ☐ D. W opisywanej strukturze katalogów deskryptor wdrożenia będzie się znajdował w tym samym katalogu co katalog **WARdir**.
- ☐ E. Umieszczenie aplikacji w pliku **WAR** stwarza kontenerowi możliwość przeprowadzenia w czasie wykonywania dodatkowych testów, które w przeciwnym razie nie byłyby możliwe.  
Odpowiedź E: plik **WAR** stwarza możliwość wykonywania dodatkowych testów w czasie wdrażania aplikacji.

**22** Które zdanie o metodach **GET** i **POST** protokołu HTTP jest prawdziwe?  
(Zaznacz wszystkie prawidłowe odpowiedzi).

Specyfikacja HTTP 1.1;  
hf rozdział 4.

- ☒ A. Tylko metoda **GET** jest idempotentna. – Odpowiedź A: jeśli formularz nie deklaruje wprost metody protokołu HTTP, stosuje się metodę **GET**.
- ☐ B. Obie metody wymagają zadeklarowania wprost w znaczniku **form** języka HTML.
- ☐ C. Tylko metoda **POST** może obsługiwać wiele parametrów w jednym żądaniu.
- ☒ D. Obie metody obsługują żądania z pojedynczymi parametrami reprezentującymi wiele wartości. – Odpowiedź D: obie metody mogą obsługiwać takie żądania.
- ☐ E. Tylko żądania **POST** powinny być obsługiwane przez nadpisaną metodę **service()** serwletu.

– Odpowiedź E: na potrzeby egzaminu przyjmij, że nigdy nie należy nadpisywać metody **service()**.

**23** Nasz serwlet zawiera następujący wiersz:

Specyfikacja serwletów,  
dodatek B; hf 150.

**82.** `String s = getServletConfig().getInitParameter("mojParametr");`

Który fragment deskryptora wdrożenia spowoduje przypisanie zmiennej **s** wartości **"mojaWartosc"**?

- ☐ A. `<init-param>`  
    `<param>mojParametr</param>`  
    `<value>mojaWartosc</value>`  
    `</init-param>`
- ☐ B. `<init-param>`  
    `<name>mojParametr</name>`  
    `<value>mojaWartosc</value>`  
    `</init-param>`
- ☒ C. `<init-param>`  
    `<param-name>mojParametr</param-name>`  
    `<param-value>mojaWartosc</param-value>`  
    `</init-param>`
- ☐ D. `<servlet-param>`  
    `<name>mojParametr</name>`  
    `<value>mojaWartosc</value>`  
    `</servlet-param>`
- ☐ E. `<servlet-param>`  
    `<param-name>mojParametr</param-name>`  
    `<param-value>mojaWartosc</param-value>`  
    `</servlet-param>`

– Odpowiedź C reprezentuje prawidłową składnię znacznika `<init-param>`.

**24** Wiemy, że jakiś łańcuch jest składowany w jednym z zasięgów w formie atrybutu nazwanego **numerKonta**. Który z poniższych skryptletów wyświetli wartość tego atrybutu? *Specyfikacja, punkt 1.8.3; hf 298.*

- ☒ A. `<%= pageContext.findAttribute("numerKonta") %>` – Odpowiedź A: gdybyś musiał użyć skryptletów, to rozwiązanie byłoby najprostsze.
- ☐ B. `<%= out.print("${numerKonta}") %>`
- ☐ C. `<% Object nrKonta = pageContext.getAttribute("numerKonta");  
if(nrKonta == null) {  
nrKonta = request.getAttribute("numerKonta");  
}  
if(nrKonta == null) {  
nrKonta = session.getAttribute("numerKonta");  
}  
if(nrKonta == null) {  
nrKonta = servletContext.getAttribute("numerKonta");  
}  
out.print(nrKonta);  
%>` – Odpowiedź B: wyrażenia języka EL stosowane w skryptletach nie są przetwarzane. Co więcej, takie użycie skryptletu jest błędne, zatem nie traktuj tej konstrukcji jako sprytnego zabiegu.  
– Odpowiedź C: byłeś blisko. `servletContext` nie jest prawidłowym obiektem domyślnym. Należałoby użyć obiektu `application`.
- ☐ D. `<% requestDispatcher.include("numerKonta"); %>` – Odpowiedź D: `requestDispatcher` nie jest obiektem domyślnym. Nawet gdyby był, to rozwiązanie i tak byłoby błędne.

**25** Odziedziczyłeś stosowaną od dawna aplikację internetową złożoną ze stron JSP zawierających mnóstwo kodu skryptowego. Szef żąda od Ciebie takiej przebudowy każdej z tych stron, aby nie zawierała żadnego kodu skryptowego. Masz mu zagwarantować, że w bazie kodowej JSP nie znajdzie się ani jeden skryptlet — w ten sposób będzie można zrealizować w kontenerze „politykę bezskryptową”.

*Specyfikacja JSP 2.0, punkt 3.3.3.*

Który element konfiguracyjny użyty w pliku **web.xml** pozwoli osiągnąć ten cel?

- ☐ A. `<jsp-property-group>`  
    `<url-pattern> *.jsp </url-pattern>`  
    `<permit-scripting> false </permit-scripting>`  
    `</jsp-property-group>` – Odpowiedź A jest błędna, ponieważ `<permit-scripting>` nie jest prawidłowym elementem konfiguracyjnym.
- ☐ B. `<jsp-config>`  
    `<url-pattern> *.jsp </url-pattern>`  
    `<permit-scripting> false </permit-scripting>`  
    `</jsp-config>` – Odpowiedź B jest błędna, ponieważ ani `<jsp-config>`, ani `<permit-scripting>` nie są prawidłowymi elementami konfiguracyjnymi.
- ☒ C. `<jsp-property-group>`  
    `<url-pattern> *.jsp </url-pattern>`  
    `<scripting-invalid> true </scripting-invalid>`  
    `</jsp-property-group>`
- ☐ D. `<jsp-config>`  
    `<url-pattern> *.jsp </url-pattern>`  
    `<scripting-invalid> true </scripting-invalid>`  
    `</jsp-config>` – Odpowiedź D jest błędna, ponieważ `<jsp-config>` nie jest prawidłowym elementem konfiguracyjnym.

26 Dysponujesz następującym fragmentem kodu:

Specyfikacja JSP 2.0,  
punkt 2.3.7, hf 396.

```
01. <%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
02.
03. <%
04. java.util.List books = new java.util.ArrayList();
05. // tutaj wstaw brakujący wiersz
06. request.setAttribute("myFavoriteBooks", books);
07. %>
08.
09. <c:choose>
10. <c:when test="${not empty myFavoriteBooks}">
11. Oto moje ulubione książki:
12. <c:forEach var="book" items="${myFavoriteBooks}">
13.
 * ${book}
14. </c:forEach>
15. </c:when>
16. <c:otherwise>
17. Nie wybrałem żadnej ulubionej książki.
18. </c:otherwise>
19. </c:choose>
```

Który z poniższych wierszy kodu wstawiony w wierszu 5. przedstawionego listingu spowoduje wyświetlenie tekstu wewnątrz znacznika **c:otherwise**? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. `books.add("");`      – Odpowiedzi A, B i D reprezentują wywołania, które dodają coś do listy `books`, zatem żadne z tych wywołań NIE spowoduje, że nasza lista będzie pusta.
- ☐ B. `books.add(null);`
- ☒ C. `books.clear();`      – Odpowiedź C opróżnia już pustą listę.
- ☐ D. `books.add("Head First");`
- ☒ E. `books = null;`      – Odpowiedź E: przypisanie obiektowi listy wartości `null` spowoduje akceptację operatora `empty`.

27 Pracujesz nad aplikacją zarządzającą katalogami list biznesowych.

Specyfikacja JSP 2.0, podrozdział 2.6,  
hf 388 – 391.

Masz dany następujący kod:

```
29. <c:forEach var="phoneNumber" items='${company.contactInfo.phoneNumbers}'>
30. <c:if test='${verify:isTollFree(phoneNumber)}'>
31.
32. </c:if>
33. ${phoneNumber}

34. </c:forEach>
```

Przedstawiony fragment kodu poprzedza specjalną ikoną numery telefonów, z którymi można się łączyć za darmo. Które zdanie o użyciej powyżej funkcji języka EL na pewno jest prawdziwe?

- ☒ A. Użyta funkcja języka EL musi zostać zadeklarowana jako składowa publiczna i statyczna.
- ☐ B. Użyta funkcja języka EL nie może zwracać żadnej wartości (musi zwracać typ **void**).
- ☐ C. W elemencie **<uri>** deklaracji TLD tej funkcji EL należy użyć wartości **Verify**.
- ☐ D. Klasa implementująca tę funkcję języka EL musi się nazywać **Verify**.
- ☐ E. Jeśli **phoneNumber** jest łańcuchem, element **<function-signature>** odpowiedniej deklaracji TLD powinien zawierać sygnaturę **isTollFree(String)**.

– Odpowiedź A: wszystkie funkcje języka EL muszą być deklarowane jako publiczne i statyczne.

– Odpowiedź B: funkcja powinna zwrócić wartość logiczną, którą będzie można wykorzystać w znaczniku **<c:if>**.

– Odpowiedź C: wartość **<uri>** powinna pasować do tego, co zadeklarowano w dyrektywie taglib JSP (której nie przedstawiono w powyższym kodzie).

– Odpowiedź D: element **<function-class>** pliku TLD odwzorowuje w pełni kwalifikowaną nazwę klasy. Nazwy funkcji języka EL nie muszą być zgodne z żadnymi konwencjami nazewnictwa.

– Odpowiedź E: **<function-signature>** wymaga zadeklarowania zwracanego typu. Co więcej, element **<function-signature>** wymaga też stosowania w pełni kwalifikowanych nazw klas, zatem typ łańcuchowy należy zadeklarować jako **java.lang.String**.

28 Które z wymienionych metod interfejsu **HttpServletRequest** zapewniają dostęp do ciała żądania? (Zaznacz wszystkie prawidłowe odpowiedzi).

API

- ☒ A. **getReader()** – Odpowiedź A: metoda **getReader()** zwraca ciało w formie danych znakowych.
- ☐ B. **getStream()**
- ☐ C. **getInputStreamReader()**
- ☒ D. **getInputStream()** – Odpowiedź D: metoda **getInputStream()** zwraca ciało w formie danych binarnych.
- ☐ E. **getServletReader()**
- ☐ F. **getServletStream()**

- 29 Dysponujesz aplikacją internetową stworzoną w technologii Java EE.  
Do Twojej aplikacji trafia następujące żądanie:

Serv II, hf 616.

**http://www.wickedlysmart.com/MojaAplikacja/mojKatalog/ZrobCos**

Przytoczone żądanie zostanie obsłużone przez pewien serwet.

Które zdania są prawdziwe? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☒ A. Deskryptor wdrożenia musi zawierać instrukcje opisujące, jak obsłużyć żądanie w tej formie.
- ☐ B. Powyższe żądanie może zostać prawidłowo obsłużone mimo braku odpowiednich instrukcji w deskrypcji wdrożenia.
- ☐ C. Serwet obsługujący to żądanie musi się nazywać **ZrobCos.class**.
- ☒ D. Na podstawie podanych informacji określenie nazwy tego serwetu jest niemożliwe.
- ☐ E. Aplikacja musi zawierać katalog nazwany **mojKatalog**.
- ☒ F. Na podstawie podanych informacji określenie nazwy katalogu, w którym umieszczono dany serwet, jest niemożliwe.

– Odpowiedź A: w deskrypcji wdrożenia należy zadeklarować znacznik `<servlet-mapping>`.

– Odpowiedzi C i E: `mojKatalog` i `ZrobCos` to nazwy wirtualne znane tylko deskrypcji wdrożenia.

- 30 Twoja aplikacja internetowa obejmuje prawidłowy deskryptor wdrożenia, w którym zdefiniowano tylko dwie role zabezpieczeń: **uczen** i **mistrz**. Wspomniany deskryptor zawiera też dwa ograniczenia zabezpieczeń odnoszące się do tego samego zasobu. Pierwsze z tych ograniczeń ma następującą postać:

Specyfikacja serwetów, podrozdział 12.8, hf 668 – 669.

234. `<auth-constraint>`  
235. `<role-name>uczen</role-name>`  
236. `</auth-constraint>`

Drugie ograniczenie zdefiniowano w następujący sposób:

251. `<auth-constraint/>`

Które zdania są prawdziwe? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. Deskryptor wdrożenia w tej formie deklaruje, że chroniony zasób jest dostępny dla obu ról.
- ☐ B. Deskryptor wdrożenia w tej formie deklaruje, że chroniony zasób jest dostępny tylko dla użytkowników przypisanych do roli **mistrz**.
- ☐ C. Deskryptor wdrożenia w tej formie deklaruje, że chroniony zasób jest dostępny tylko dla użytkowników przypisanych do roli **uczen**.
- ☒ D. Gdybyśmy usunęli drugi znacznik `<auth-constraint>`, chroniony zasób byłby dostępny dla obu ról.
- ☐ E. Gdybyśmy usunęli drugi znacznik `<auth-constraint>`, chroniony zasób byłby dostępny tylko dla użytkowników przypisanych do roli **mistrz**.
- ☐ F. Gdybyśmy usunęli drugi znacznik `<auth-constraint>`, chroniony zasób byłby dostępny tylko dla użytkowników przypisanych do roli **uczen**.

– Odpowiedzi A, B i C: drugi znacznik jest pusty, zatem żadna rola nie ma dostępu do tego zasobu.

- 31** Który z poniższych znaczników niestandardowych na pewno nie zadziała? (Zaznacz wszystkie prawidłowe odpowiedzi).

Specyfikacja JSP 2.0 str. 1 – 31; hf rozdział 10.

☒ A. `<moje:ramka>`  
`<moje:zdjecia album="${wybranyAlbum}">` – Odpowiedź A: znacznik `<mine:zdjecia>` nie został prawidłowo zagnieżdżony.  
`</moje:ramka>`  
`</moje:zdjecia>`

☐ B. `<moje:ramka>`  
`<moje:zdjecia album="${wybranyAlbum}"/>`  
`</moje:ramka>`

☐ C. `<moje:ramka>`  
`${wybranyAlbum.tytul}`  
`<moje:zdjecia>${wybranyAlbum}</moje:zdjecia>`  
`</moje:ramka>`

– Odpowiedzi B, C i D reprezentują potencjalnie prawidłowe zastosowania znaczników niestandardowych.

☐ D. `<moje:zdjecia includeBorder="${userPreference.ramka}"`  
`album="${wybranyAlbum}" />`

- 32** Twoja  $n$ -warstwowa aplikacja internetowa wykorzystuje najbardziej popularne wzorce projektowe technologii Java EE w operacjach dostępu do zdalnych rejestrów. Jakie są korzyści stosowania tych wzorców? (Zaznacz wszystkie prawidłowe odpowiedzi).

core j2ee str. 315 – 318; hf 754.

☒ A. Większa spójność. ←  
☐ B. Wyższa wydajność.  
☒ C. Łatwość utrzymania. ←

– Tymi dwoma wzorcami projektowymi są *Business Delegate* i *Service Locator*. Łączne stosowanie tych wzorców pozwala poszczególnym komponentom koncentrować się na własnych obszarach odpowiedzialności i ułatwia utrzymanie aplikacji w razie zmian architektonicznych.

☐ D. Mniejsze obciążenie sieci.  
☐ E. Szersze możliwości w zakresie interaktywności przeglądarki.

– Odpowiedź D: jeśli wskazałeś odpowiedź D, nie przejmuj się — implementacja obiektu lokalizatora usługi z pamięcią podręczną rzeczywiście pozwala ograniczyć obciążenie sieci. Z drugiej strony, pamięć podręczna ma też swoje wady, zatem to rozwiązanie nie jest najlepsze.

- 33** Które zdania o cyklu życia serwletu są prawdziwe? (Zaznacz wszystkie prawidłowe odpowiedzi).

API; specyfikacja serwletów; hf 97 – 99.

☒ A. NIE powinieneś pisać konstruktora dla serwletu.  
☐ B. NIE powinieneś nadpisywać metody `init()` serwletu.  
☐ C. NIE powinieneś nadpisywać metody `doGet()` serwletu.  
☐ D. NIE powinieneś nadpisywać metody `doPost()` serwletu.  
☒ E. NIE powinieneś nadpisywać metody `service()` serwletu.  
☐ F. NIE powinieneś nadpisywać metody `destroy()` serwletu.

– Odpowiedzi B i F reprezentują typowe rozwiązania w sytuacji, gdy serwlet musi tworzyć i niszczyć wykorzystywane zasoby, na przykład połączenia z bazą danych.

**34** Mamy dany następujący fragment struktury katalogów aplikacji Javy EE w ramach pliku `.war`: *Specyfikacja serwletów, rozdział 9., hf 612 – 613.*

MojaAplicacja

```

|-- META-INF
|
| |-- MANIFEST.MF
| |-- web.xml
|
|-- WEB-INF
|
| |-- index.html
| |-- TLDs
|
| |-- Naglowek.tag

```

Jaka zmiana (zmiany) jest konieczna, aby przedstawiona struktura była prawidłowa i aby zasoby tej aplikacji były dostępne? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. Nie potrzeba żadnych zmian.
- ☒ B. Należy przenieść w inne miejsce plik **web.xml**. *– Odpowiedź B: plik web.xml musi się znajdować w katalogu WEB-INF.*
- ☒ C. Należy przenieść w inne miejsce plik **index.html**. *– Odpowiedź C: plik index.html musi być poza katalogiem WEB-INF, aby był dostępny dla klientów.*
- ☒ D. Należy przenieść w inne miejsce plik **Naglowek.tag**. *– Odpowiedź D: pliki znaczników muszą się znajdować w części WEB-INF/ tags/ struktury katalogów.*
- ☐ E. Należy przenieść w inne miejsce plik **MANIFEST.MF**.
- ☐ F. Należy przenieść w inne miejsce katalog **WEB-INF**.
- ☐ G. Należy przenieść w inne miejsce katalog **META-INF**.

**35** Zastanawiasz się nad implementacją pewnej odmiany wzorca MVC w swojej *n*-warstwowej aplikacji Javy EE. Które zdania są prawdziwe? (Zaznacz wszystkie prawidłowe odpowiedzi). *core j2ee str. 166.; hf rozdział 14.*

- ☐ A. Tego rodzaju projekty często wykorzystują obiekty delegatów biznesowych. *– Odpowiedź A: obiekty delegatów biznesowych są wykorzystywane przez kontrolery*
- ☐ B. Składowanie danych pobieranych ze zdalnego komponentu często pozwala ograniczyć obciążenie sieci. *– Odpowiedź B: obiekty implementujące wzorec MVC mogą stosować pamięć podręczną, chociaż sam wzorec MVC tego nie przewiduje.*
- ☐ C. Postawiony cel projektowy upraszcza komunikację z heterogenicznymi rejestrami zasobów. *– Odpowiedź C: to zadanie obiektu lokalizatora usługi.*
- ☒ D. Rozwiązania na bazie wzorca MVC mają co prawda wiele zalet, ale często komplikują projekt aplikacji.
- ☒ E. Warto rozważyć realizację tego celu projektowego z wykorzystaniem wzorca Front Controller (kontrolera frontowego) i frameworku Struts. *– Odpowiedź F: to zadanie obiektu filtra przechwytyjącego, który można stosować łącznie ze wzorcem MVC, chociaż sam wzorec Intercepting Filter jest odrębnym wzorcem.*
- ☐ F. Gotowy projekt powinien w przyszłości ułatwić przebudowę klas odpowiedzialnych za obsługę żądań i odpowiedzi.

**36** Masz daną stronę JSP z następującym wierszem:

Specyfikacja JSP 2.0, podpunkt 1.1.0.1.

```
<% List mojaLista = new ArrayList(); %>
```

Który z poniższych fragmentów kodu JSP można wykorzystać do zaimportowania tych typów danych? (Zaznacz wszystkie prawidłowe odpowiedzi).

☐ A. `<%! import java.util.*; %>`

– Odpowiedź A jest błędna, ponieważ znacznik deklaracji JSP nie może być wykorzystywany do wstawiania wyrażeń importujących do tłumaczonego kodu serwletu.

☐ B. `<%@ import java.util.List java.util.ArrayList %>`

– Odpowiedź B jest błędna, ponieważ nie istnieje dyrektywa import.

☒ C. `<%@ page import='java.util.List,java.util.ArrayList' %>`

☐ D. `<%! import java.util.List; import java.util.ArrayList; %>`

– Odpowiedź D jest błędna, ponieważ znacznik deklaracji JSP nie może być wykorzystywany do wstawiania wyrażeń importujących do tłumaczonego kodu serwletu.

☐ E. `<%@ page import='java.util.List' %> <%@ page import='java.util.ArrayList' %>`

– Odpowiedź E jest prawidłowa, ponieważ atrybut import dyrektywy page można stosować więcej niż raz.

**37** Otrzymałeś zadanie dodania kilku mechanizmów zabezpieczeń do używanej w Twojej firmie aplikacji internetowej Javy EE. W szczególności musisz opracować szereg klas użytkowników, by na tej podstawie określać, które grupy mają dostęp do poszczególnych stron aplikacji. Kontrola dostępu do chronionych stron wymaga potwierdzania tożsamości użytkowników.

Specyfikacja serwletów, rozdział 12.; hf rozdział 12.

Które zdania są prawdziwe? (Zaznacz wszystkie prawidłowe odpowiedzi).

☐ A. Jeśli musisz sprawdzać, czy użytkownicy rzeczywiście są tymi, za których się podają, powinieneś odpowiednio wymaganie zadeklarować w deskrytorze wdrożenia aplikacji.

– Odpowiedź A: uwierzytelnianie można realizować także programowo.

☐ B. Do sprawdzania, czy użytkownicy są tymi, za których się podają, należy wykorzystać mechanizmy autoryzacji Javy EE.

– Odpowiedź B: w tym przypadku pytanie dotyczy uwierzytelniania, nie autoryzacji.

☒ C. Aby uprościć proces weryfikacji, czy użytkownicy są tymi, za których się podają, można użyć znaczników `<login-config>` deskryptora wdrożenia.

☐ D. Aby uprościć proces weryfikacji, czy użytkownicy są tymi, za których się podają, można użyć znaczników `<user-data-constraint>` deskryptora wdrożenia.

– Odpowiedź D: ten znacznik służy do implementowania integralności danych.

☒ E. W zależności od stosowanego rozwiązania określanie, czy użytkownicy są tymi, za których się podają, może wymagać dołączenia domeny (`realm`).

- 38** PoprawnaApplikacja to aplikacja internetowa Javy EE z prawidłową strukturą katalogów. PoprawnaApplikacja obejmuje między innymi pliki graficzne .gif składowane w trzech katalogach tej struktury:

Specyfikacja serwetów, rozdział 9.; hf 614.

- PoprawnaApplikacja/imageDir/
- PoprawnaApplikacja/META-INF/
- PoprawnaApplikacja/WEB-INF/

W którym z wymienionych poniżej katalogów należy umieścić pliki .gif, aby były bezpośrednio dostępne dla klientów?

- ☐ A. Tylko w katalogu PoprawnaApplikacja/META-INF/.
- ☒ B. Tylko w katalogu PoprawnaApplikacja/imageDir/.
- ☐ C. We wszystkich wymienionych katalogach.
- ☐ D. Tylko w katalogach PoprawnaApplikacja/imageDir/ i PoprawnaApplikacja/WEB-INF/.
- ☐ E. Tylko w katalogach PoprawnaApplikacja/imageDir/ i PoprawnaApplikacja/META-INF/.

– Odpowiedź B: w odpowiedzi na próby uzyskania przez klienta dostępu do plików składowanych w katalogach WEB-INF lub META-INF kontener musi zwrócić kod 404.

- 39** Przyjmijmy, że req jest referencją do obiektu HttpServletRequest oraz że dysponujemy kodem w postaci:

API.

```
13. String[] s = req.getCookies();
14. Cookie[] c = req.getCookies();
15. req.setAttribute("mojAtrybut1", "42");
16. req.setAttribute("mojAtrybut2", 42);
17. String[] s2 = req.getAttributeNames();
18. String[] s3 = req.getParameterValues("attr");
```

Które wiersze tego kodu zostaną odrzucone przez kompilator? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☒ A. Wiersz 13.
- ☐ B. Wiersz 14.
- ☐ C. Wiersz 15.
- ☐ D. Wiersz 16.
- ☒ E. Wiersz 17.
- ☐ F. Wiersz 18.

– Odpowiedź A: metoda getCookies() zwraca tablicę obiektów klasy Cookie.

– Odpowiedź D: metoda setAttribute() otrzymuje na wejściu obiekty typów String i Object, a w Javie 5 istnieje możliwość opakowania wartości 42 w obiekcie klasy Integer.

– Odpowiedź E: metoda getAttributeNames() zwraca wartość typu Enumeration.

Zdajemy sobie sprawę z tego, że tego rodzaju pytania wymagają nauki na pamięć. Przykro nam, ale z podobnymi pytaniami możesz się zetknąć na prawdziwym egzaminie.

**40** Plik znacznika nazwany **Products.tag** wyświetla listę produktów.

Specyfikacja serwetów 2.0, punkty 8.5.1 – 8.5.2; hf 506 – 508.

Dysponujemy następującym fragmentem tego pliku:

1. `<%@ attribute name="header" required="false" rtexprvalue="false" %>`
2. `<%@ attribute name="products" required="true" rtexprvalue="true" %>`
3. `<%@ tag body-content="tagdependent" %>`

Które z poniższych zastosowań tego pliku znacznika są prawidłowe?  
(Zaznacz wszystkie prawidłowe odpowiedzi).

- ☒ A. `<display:Products header="Koszyk z zakupami" products="${shoppingCart}"/>`
- ☐ B. `<display:Products header="Lista życzeń" products="${wishList}" body-content="${body}"/>` – Odpowiedź B: `body-content` nie jest prawidłowym atrybutem.
- ☒ C. `<display:Products header="Podobne produkty" products="${similarProducts}"`  
     Klienci, którzy kupili ten produkt, kupili też: – Odpowiedź C: definiowanie ciała znacznika jest możliwe, ponieważ w dyrektywie `tag` atrybutowi `body-content` przypisano wartość `tagdependent`.  
     `</display:Products>`
- ☐ D. `<display:Products header='<%= request.getParameter("listType") %>' />`  
     – Odpowiedź D: `products` to wymagany atrybut. Co więcej, atrybut `header` nie może zawierać skryptletu, ponieważ zdefiniowano go z wartością `false` atrybutu `rtexprvalue`.

**41** Bierzesz udział w przedsięwzięciu polegającym na eliminowaniu skryptletów z kodu JSP przestarzałej aplikacji dużego banku. Odkrywasz w modyfikowanym kodzie następujące wiersze:

Specyfikacja JSP 2.0, punkt 2.3.4; hf 370 – 378.

```
<% if((com.yourcompany.Account)request.getAttribute("account")).
isPersonalChecking()){ %>
```

    Weryfikacja zgodności z Twoim stylem życia.

```
<% } %>
```

Którym znacznikiem biblioteki JSTL można zastąpić tę konstrukcję? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☒ A. `<c:if test='${account.personalChecking}'>Weryfikacja zgodności z Twoim stylem życia.</c:if>`
- ☒ B. `<c:if test='${account["personalChecking"]}'>Weryfikacja zgodności z Twoim stylem życia.</c:if>` – Odpowiedzi B i C: zwróć uwagę na możliwość stosowania apostrofów i cudzysłowów oraz na to, że w języku EL symbole otaczające wyrażenia nie muszą być identyczne w ramach przetwarzanego znacznika. Przytoczona reguła nie znajduje zastosowania w przypadku znaczników tekstu szablonowego, które nie są przetwarzane: `<a href="${initParam}"contact-email">email</a>`.
- ☒ C. `<c:if test='${account['personalChecking']}'>Weryfikacja zgodności z Twoim stylem życia.</c:if>`
- ☐ D. `<c:if test='${account.isPersonalChecking}'>Weryfikacja zgodności z Twoim stylem życia.</c:if>` – Odpowiedź D: to wyrażenie będzie szukało metody `getIsPersonalChecking` obiektu `Account` i wygeneruje stosowny wyjątek, kiedy jej odnalezienie okaże się niemożliwe.

**42** Mamy następujące typy zdarzeń:

API; hf 264

- `HttpSessionEvent`
- `HttpSessionBindingEvent`
- `HttpSessionAttributeEvent`

Dopasuj powyższe typy zdarzeń do odpowiednich interfejsów nasłuchujących.  
(Uwaga: pojedynczy typ zdarzeń można dopasować do więcej niż jednego interfejsu).

<code>HttpSessionAttributeListener</code>	..... <code>HttpSessionBindingEvent</code> .....
<code>HttpSessionListener</code>	..... <code>HttpSessionEvent</code> .....
<code>HttpSessionActivationListener</code>	..... <code>HttpSessionEvent</code> .....
<code>HttpSessionBindingListener</code>	..... <code>HttpSessionBindingEvent</code> .....

**43** Które zdanie o cyklu życia serwletu jest prawdziwe? (Zaznacz wszystkie prawidłowe odpowiedzi). *Specyfikacja serwletów 2.0; hf 97 – 101.*

- ☐ A. Metoda **`service()`** jest wywoływana przez kontener jako pierwsza po otrzymaniu nowego żądania. *– Odpowiedź A: metoda `init()` jest wywoływana jako pierwsza.*
- ☐ B. Metoda **`service()`** jest wywoływana albo przez metodę **`doPost()`**, albo przez metodę **`doGet()`** już po przetworzeniu żądania przez te metody. *– Odpowiedź B: metoda `service()` wywołuje metodę `doGet()` lub `doPost()`.*
- ☒ C. Każde wywołanie metody **`doPost()`** jest realizowane w odrębnym wątku. *– Odpowiedź D: kontener wywołuje metodę `destroy()` w momencie, w którym decyduje się na usunięcie danego serwletu.*
- ☐ D. Metoda **`destroy()`** jest każdorazowo wywoływana po wykonaniu metody **`doGet()`**.
- ☒ E. Kontener tworzy odrębny wątek dla każdego żądania klienta.

**44** Kiedy następuje tłumaczenie kodu strony JSP? (Zaznacz wszystkie prawidłowe odpowiedzi). *Specyfikacja JSP 2.0, punkt 1.1.4; hf 308.*

- ☐ A. Kiedy programista kompiluje kod umieszczony w folderze **`src`**. *– Odpowiedź A: stron JSP nie umieszcza się w folderze `src`, a programista nie kompiluje ich jak zwykłego kodu.*
- ☒ B. W chwili uruchamiania aplikacji.
- ☒ C. Po otrzymaniu pierwszego żądania danej strony JSP od użytkownika. *– Odpowiedzi B i C: tłumaczenie strony JSP może nastąpić w dowolnym momencie od chwili jej początkowego wdrożenia w kontenerze JSP do czasu przetworzenia żądania klienta wskazującego na tę stronę.*
- ☐ D. Po wywołaniu metody **`jspDestroy()`**. *– Odpowiedź D: metoda `jspDestroy()` nie powoduje ponownego przetłumaczenia tej samej strony.*

**45** Mamy dany następujący fragment prawidłowej metody `doGet()`:

API; hf 205 – 207.

```
12. OutputStream os = response.getOutputStream();
13. byte[] ba = {1,2,3};
14. os.write(ba);
15. RequestDispatcher rd = request.getRequestDispatcher("moja.jsp");
16. rd.forward(request, response);
```

Jeśli przyjmiemy, że plik **moja.jsp** dodaje bajty 4, 5 i 6 do odpowiedzi, jaki będzie wynik?

- ☐ A. 123
- ☒ B. 456 – Odpowiedź B: ponieważ nie wywołano metody `os.flush()`, niezatwierdzone dane wyjściowe (123) zostaną utracone, a metoda `forward()` nie wygeneruje wyjątku. Gdyby wywołano metodę `os.flush()` przed metodą `forward()`, zostałby wygenerowany wyjątek `IllegalStateException`.
- ☐ C. 123456
- ☐ D. 456123
- ☐ E. Zostanie wygenerowany wyjątek.

**46** Programista musi tak zaktualizować parametry inicjalizacji żywego, działającego serwletu, aby dana aplikacja internetowa natychmiast zaczęła uwzględniać nowe parametry.

Specyfikacja serwletów 2.0;  
hf 151 – 155.

Które zdanie prawidłowo opisuje niezbędne (choć niekoniecznie wystarczające) działania?  
(Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. Dla każdego parametru należy zmodyfikować znacznik deskryptora wdrożenia opisujący nazwę serwletu, nazwę samego parametru oraz jego nową wartość.
- ☐ B. Konstruktor serwletu musi odczytać zaktualizowane parametry (zadeklarowane w deskrytorze wdrożenia) za pośrednictwem obiektu **ServletConfig**.
- ☒ C. Kontener musi zniszczyć i ponownie zainicjalizować dany serwlet.
- ☒ D. Dla każdego parametru deskryptor wdrożenia musi zawierać osobny znacznik **<init-param>**.

– Odpowiedź A: znacznik `<init-param>` musi się znajdować wewnątrz znacznika `<servlet>`, zatem nie obejmuje nazwy serwletu.

– Odpowiedź B: obiektu `ServletConfig` nie można uzyskać przed zakończeniem wykonywania konstruktora.

– Odpowiedź C: uzyskanie nowego obiektu `ServletConfig` wymaga zainicjalizowania nowego obiektu `Servlet`

**47** Które typy można stosować łącznie z metodami interfejsu **HttpServletResponse** do kierowania danych wyjściowych do odpowiedniego strumienia? (Zaznacz wszystkie prawidłowe odpowiedzi).

API; hf 132.

- ☐ A. **java.io.PrintStream** – Odpowiedź A: metoda `getWriter()` zwraca obiekt `PrintWriter`.
- ☒ B. **java.io.PrintWriter**
- ☐ C. **javax.servlet.OutputStream**
- ☐ D. **java.io.FileOutputStream**
- ☒ E. **javax.servlet.ServletOutputStream** – Odpowiedź E: metoda `getOutputStream()` zwraca obiekt `ServletOutputStream`.
- ☐ F. **java.io.ByteArrayOutputStream**

**48** Twoja aplikacja internetowa obejmuje prawidłowy deskryptor wdrożenia z pojedynczym znacznikiem **<security-constraint>**. Wspomniany znacznik zawiera:

Specyfikacja serwletów,  
podrozdział 12.8; hf 666.

- pojedynczy wzorzec adresów URL z katalogiem **katalog1**;
- pojedynczą metodę protokołu HTTP z wartością **POST**;
- pojedynczą nazwę roli wskazującą rolę **GOSC**.

Jeśli przyjąć, że wszystkie zasoby Twojej aplikacji znajdują się w katalogach **katalog1** i **katalog2**, i jeśli inną prawidłową rolą jest **CZLONEK**, które zdania są prawdziwe? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. Użytkownicy przypisani do roli **GOSC** nie mogą kierować żądań **GET** do zasobów w katalogu **katalog1**.
- ☒ B. Użytkownicy przypisani do roli **GOSC** mogą kierować żądania **GET** do zasobów w obu katalogach.
- ☐ C. Użytkownicy przypisani do roli **GOSC** mogą kierować żądania **POST** tylko do zasobów w katalogu **katalog2**.
- ☒ D. Użytkownicy przypisani do roli **CZLONEK** mogą kierować żądania **GET** do zasobów w obu katalogach.
- ☒ E. Użytkownicy przypisani do roli **GOSC** mogą kierować żądania **POST** do zasobów w obu katalogach.
- ☐ F. Użytkownicy przypisani do roli **CZLONEK** mogą kierować tylko żądania **POST** do zasobów w katalogu **katalog1**.

– Zgodnie z opisanym ograniczeniem tylko użytkownicy przypisani do roli **GOSC** mogą kierować żądania **POST** do zasobów w katalogu **katalog1**.

**49** Dysponujesz następującym kodem:

Specyfikacja JSP 2.0, podpunkt 1.1.0.2; hf 314, 502.

1. `<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>`
2. `<%@ taglib prefix="tables" uri="http://www.javaranh.com/tables" %>`
3. `<%@ taglib prefix="jsp" tagdir="/WEB-INF/tags" %>`
4. `<%@ taglib uri="UtilityFunctions" prefix="util" %>`

Które elementy powyższych dyrektyw **taglib** uniemożliwią właściwe funkcjonowanie tej strony JSP?

- ☐ A. Wiersz 4. zawiera błąd, ponieważ atrybut **prefix** musi się znajdować przed atrybutem **uri**.
- ☐ B. Wiersz 3. zawiera błąd, ponieważ zabrakło atrybutu **uri**.
- ☐ C. Wiersz 4. zawiera błąd, ponieważ wartość atrybutu **uri** musi się rozpoczynać od sekwencji **http://**.
- ☒ D. Wiersz 3. zawiera błąd, ponieważ przedrostek **jsp** jest zarezerwowany dla akcji standardowych.

– Odpowiedź A: kolejność atrybutów nie ma znaczenia.

– Odpowiedź B: jeśli korzystamy z plików znaczników, zamiast atrybutu **uri** należy stosować atrybut **tagdir**

– Odpowiedź C: identyfikator URI musi odpowiadać sposobowi identyfikacji pliku TLD przez kontener.

– Odpowiedź D: przedrostek **jsp** jest zarezerwowany dla akcji standardowych.

- 50 Przyjmijmy, że **resp** jest referencją do prawidłowego obiektu **HttpServletResponse** zawierającego między innymi następujące nagłówki:

Specyfikacja serwletów, rozdział 5.; hf 133.

**Content-Type: text/html**

**MojNaglowek: mojedane**

Kod serwletu zawiera następujące wywołania:

```
25. resp.addHeader("MojNaglowek", "mojedane2");
26. resp.setHeader("MojNaglowek", "mojedane3");
27. resp.addHeader("MojNaglowek", "mojedane");
```

Jakie dane ostatecznie znajdują się w nagłówku **MojNaglowek**?

- ☐ A. mojedane
- ☐ B. mojedane3
- ☒ C. mojedane3,mojedane
- ☐ D. mojedane3,mojedane2
- ☐ E. mojedane,mojedane2,mojedane3
- ☐ F. mojedane,mojedane2,mojedane3,mojedane

– Odpowiedź C: metoda `setHeader()` zastępuje wszystkie już istniejące dane nagłówka; metoda `addHeader()` dopisuje nowe dane do już istniejących.

- 51 Dysponujesz następującym fragmentem pliku *web.xml* starej, odziedziczonej aplikacji:

Specyfikacja JSP 2.0, punkt 7.3.4; hf 485.

```
<jsp-config>
 <taglib>
 <taglib-uri>prettyTables</taglib-uri>
 <taglib-location>/WEB-INF/tlds/prettyTables.tld</taglib-location>
 </taglib>
</jsp-config>
```

Przyjmij, że serwer wykonujący Twój kod jest teraz zgodny ze specyfikacją Java 1.4 EE lub nowszą. Co możesz zrobić, aby usunąć powyższy znacznik **<jsp-config>** i jednocześnie zachować możliwość wykonywania tego kodu?

- ☐ A. Można tak zmienić atrybut **uri** dyrektywy **taglib** w kodzie stron JSP, aby zawierał wartość **"\*"**, która zostanie automatycznie odwzorowana przez kontener.
- ☒ B. W pliku TLD należy umieścić znacznik **<uri>prettyTables</uri>**.
- ☐ C. Należy usunąć z kodu JSP dyrektywy **taglib**, które wykorzystywały to odwzorowanie. Kontener automatycznie obsłuży tę zmianę.
- ☐ D. To niemożliwe. Element **<jsp-config>** jest niezbędny, aby kontener mógł odwzorowywać ten TLD na identyfikator **uri** wykorzystywany w kodzie stron JSP.

– Odpowiedź A: \* w dyrektywie nie jest symbolem wieloznacznym taglib.

– Odpowiedź B: prawidłowa odpowiedź. Plik TLD umieszczony w katalogu WEB-INF zostanie odnaleziony przez kontener. Jeśli ten plik TLD zawiera element `<uri>`, kontener automatycznie odwzorowuje jego wartość na odpowiednią lokalizację TLD.

– Odpowiedź C: usunięcie dyrektyw `taglib` z kodu stron JSP spowoduje, że znaczniki dla `prettyTables` zostaną przekazane dalej jako tekst szablonowy.

– Odpowiedź D: to niemożliwe. Patrz Odpowiedź B!

52

Strona prezentująca zawartość koszyka z zakupami powinna wyświetlać komunikat *Twój koszyk jest pusty* w sytuacji, gdy użytkownik nie doda do koszyka żadnych produktów. Który z przedstawionych poniżej fragmentów kodu prawidłowo realizuje to zadanie, zakładając, że atrybut **cart** reprezentuje listę produktów? (Zaznacz wszystkie prawidłowe odpowiedzi).

Specyfikacja JSTL 1.1,  
podrozdziały 5.3 – 5.6  
oraz 6.2; hf 447 – 454.



A. `<c:if test='${empty cart}'>`

Twój koszyk jest pusty.

`</c:if>`

`<c:forEach var="itemInCart" items="${cart}">`

`<shop:displayItem item="${itemInCart}"/>`

`</c:forEach>`

– Odpowiedzi A, C i D są prawidłowe.  
Odpowiedź A reprezentuje najprostsze  
i – tym samym – zalecane rozwiązanie.



B. `<c:forEach var="itemInCart" items="${cart}">`

`<c:choose>`

`<c:when test='${empty itemInCart}'>`

Twój koszyk jest pusty.

`</c:when>`

`<c:otherwise>`

`<shop:displayItem item="${itemInCart}"/>`

`</c:otherwise>`

`</c:choose>`

`</c:forEach>`

– Odpowiedź B: jeśli koszyk jest pusty lub jeśli `cart` ma wartość `null`, znacznik `c:forEach` nigdy nie wykona swojego ciała. Oznacza to, że dla pustego koszyka stosowny komunikat nigdy nie zostanie wyświetlony.



C. `<c:choose>`

`<c:when test='${empty cart}'>`

Twój koszyk jest pusty.

`</c:when>`

`<c:when test='${not empty cart}'>`

`<c:forEach var="itemInCart" items="${cart}">`

`<shop:displayItem item="${itemInCart}"/>`

`</c:forEach>`

`</c:when>`

`</c:choose>`



D. `<c:choose>`

`<c:when test='${empty cart}'>`

Twój koszyk jest pusty.

`</c:when>`

`<c:otherwise>`

`<c:forEach var="itemInCart" items="${cart}">`

`<shop:displayItem item="${itemInCart}"/>`

`</c:forEach>`

`</c:otherwise>`

`</c:choose>`

- 53** Przyjmijmy, że nasz serwlet zawiera następujący kod i że **mojaZmienna** jest referencją albo do obiektu **HttpSession**, albo do obiektu **ServletContext**.

Specyfikacja serwletów,  
rozdział 2.; hf 190 – 199.

```
15. mojaZmienna.setAttribute("mojaNazwa", "mojaWartosc");
16. String s = (String) mojaZmienna.getAttribute("mojaNazwa");
17. // dalszy kod
```

Które zdanie opisujące sytuację po wykonaniu wiersza 16. jest prawdziwe?  
(Zaznacz wszystkie prawidłowe odpowiedzi).

- ☒ A. Nie można jednoznacznie określić wartości zmiennej **s**.
- ☐ B. Jeśli **mojaZmienna** jest referencją do obiektu **HttpSession**, próba kompilacji zakończy się niepowodzeniem.
- ☐ C. Jeśli **mojaZmienna** jest referencją do obiektu **ServletContext**, próba kompilacji zakończy się niepowodzeniem.
- ☐ D. Jeśli **mojaZmienna** jest referencją do obiektu **HttpSession**, zmienna **s** na pewno będzie zawierała łańcuch **"mojaWartość"**.
- ☐ E. Jeśli **mojaZmienna** jest referencją do obiektu **ServletContext**, zmienna **s** na pewno będzie zawierała łańcuch **"mojaWartość"**.

– Odpowiedź A: brak synchronizacji oznacza, że nawet obiekt **HttpSession** może się nieoczekiwanie zmienić (wystarczy sobie wyobrazić użytkownika otwierającego drugie okno przeglądarki).

- 54** Dysponujemy następującym fragmentem deskryptora wdrożenia aplikacji internetowej Javy EE:

Specyfikacja serwletów,  
dodatek B; hf 627.

```
62. <error-page>
63. <exception-type>IOException</exception-type>
64. <location>/mainError.jsp</location>
65. </error-page>
66. <error-page>
67. <error-code>404</error-code>
68. <location>/notFound.jsp</location>
69. </error-page>
```

Które zdanie jest prawdziwe?

- ☒ A. Deskryptor wdrożenia w tej formie jest nieprawidłowy.
- ☐ B. Jeśli aplikacja wygeneruje wyjątek **IOException**, nie zostanie zwrócona żadna strona.
- ☐ C. Jeśli aplikacja wygeneruje wyjątek **IOException**, zostanie zwrócona strona **notFound.jsp**.
- ☐ D. Jeśli aplikacja wygeneruje wyjątek **IOException**, zostanie zwrócona strona **mainError.jsp**.

– Odpowiedź A: element deklarujący typ wyjątku w deskrytorze wdrożenia musi zawierać w pełni kwalifikowaną nazwę klasy (na przykład **java.io.IOException**).

55 Dysponujemy następującym kodem JSP:

Specyfikacja JSP 2.0,  
punkty 6.2.2 i 6.3.2; hf 629.

1. `<%! String GREETING = "Witaj na mojej stronie"; %>`
2. `<% request.setAttribute("greeting", GREETING); %>`
3. `Pozdrowienie: ${greeting}`
4. `Jeszcze raz: <%= request.getAttribute("greeting") %>`

Podjęto próbę konwersji tej strony na następujący dokument JSP:

01. `<jsp:declaration>`
02. `String GREETING = "Witaj na mojej stronie";`
03. `</jsp:declaration>`
04. `<jsp:scriptlet>`
05. `request.setAttribute("greeting", GREETING);`
06. `</jsp:scriptlet>`
07. `Pozdrowienie: ${greeting}`
08. `Jeszcze raz: <jsp:expression>`
09. `request.getAttribute("greeting");`
10. `</jsp:expression>`

Jaki błąd popełniono w nowym dokumencie JSP? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. Nie zadeklarowano elementu `<jsp:root>`. – Odpowiedź A: `<jsp:root>` nie jest znacznikiem wymaganym.
- ☒ B. Tekst szablonu należałoby opakować w znaczniku `<jsp:text>`. – Odpowiedź B: w przeciwnym razie nie będzie to prawidłowy kod XML!
- ☐ C. W dokumentach JSP nie można stosować wyrażeń języka EL.
- ☒ D. W zawartości znacznika `<jsp:expression>` nie należy stosować średników. – Odpowiedź D: Ojej! Literówka!

56 Która z poniższych składowych aplikacji internetowej ma NAJMNIEJSZE szanse na otrzymanie wywołania za pośrednictwem sieci?

core j2ee str. 302; hf 761.

- ☐ A. serwer JNDI – Odpowiedź A: kiedy widzisz wzorzec projektowy lub komponent, o którym nie wspomniano w oficjalnych celach egzaminu, możesz bez obaw wyeliminować go jako poprawną odpowiedź.
- ☒ B. obiekt transferu
- ☐ C. lokalizator usługi – Odpowiedź B: obiekty transferu z reguły są wysyłane w ramach wywołań sieciowych, ale same nie inicjują ani nie odpowiadają na tego rodzaju wywołania.
- ☐ D. kontroler frontowy
- ☐ E. filtr przechwytyjący

57 Dysponujemy następującym fragmentem kodu:

Specyfikacja JSTL 1.1, podrozdział 4.2.

```
10. ${questionNumber}: ${question}
11. <c:forEach var="answer" items="${answer}">
...
16. </c:forEach>
```

Atrybut **question** jest łańcuchem, który może zawierać znaczniki XML-a wymagające wyświetlenia w oknie przeglądarki w formie zwykłego tekstu. W powyższym fragmencie brakuje rozwiązań wymuszających na przeglądarce wyświetlanie znaczników XML-a. Co należałoby zmienić, aby te znaczniki były wyświetlane? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☒ A. Zastąpić `${question}` znacznikiem `<c:out value="${question}"/>`.
- ☐ B. Zastąpić `${question}` znacznikiem `<c:out>${question}</c:out>`.
- ☒ C. Zastąpić `${question}` znacznikiem `<c:out escapeXML="true" value="${question}"/>`.
- ☐ D. Zastąpić `${question}` konstrukcją `<%= ${question} %>`.

– Odpowiedź D: przykro nam, ale ta odpowiedź nawet nie zbliża nas do prawidłowego rozwiązania. Nie można umieszczać wyrażeń języka EL w ramach skryptletów.

– Odpowiedź B: atrybut `value` znacznika `<c:out>` jest wymagany. Znacznik `<c:out>` może co prawda zawierać ciało, jednak ciało zastępuje wartość atrybutu `default`, nie atrybutu `value`.

– Odpowiedzi A i C: `escapeXml` domyślnie ma wartość `true`, zatem zarówno odpowiedź A, jak i odpowiedź C jest prawidłowa. Atrybut `escapeXml` znacznika `<c:out>` powoduje, że znaki XML-a (`<`, `>`, `$`, `'`, `"`) są konwertowane na specjalne kody, aby przeglądarka wyświetlała je prawidłowo (zamiast mylić z kodem HTML).

58 Twoja aplikacja internetowa Javy EE cieszy się dużą popularnością, zatem decydujesz się dodać drugi serwer, aby sprawniej obsługiwać rosnącą liczbę żądań. Które zdania o migracji sesji pomiędzy serwerami są prawdziwe? (Zaznacz wszystkie prawidłowe odpowiedzi).

Specyfikacja serwletów, rozdział 7.; hf 257 – 264.

- ☐ A. Taka migracja w ramach jednej sesji jest niemożliwa.
- ☒ B. Wraz z migrującą sesją jest przenoszony obiekt **HttpSession**.
- ☐ C. Wraz z migrującą sesją jest przenoszony obiekt **ServletContext**.
- ☐ D. Wraz z migrującą sesją jest przenoszony obiekt **HttpServletRequest**.
- ☒ E. Warunkiem migracji pomiędzy serwerami obiektu dodawanego do sesji za pośrednictwem metody **HttpSession.setAttribute** jest implementowanie przez ten obiekt interfejsu **Serializable**.
- ☐ F. Jeśli dodamy do sesji obiekt za pośrednictwem metody **HttpSession.setAttribute**, jeśli klasa tego obiektu implementuje metody **Serializable.readObject** i **Serializable.writeObject** oraz jeśli sesja podlega migracji, kontener wywoła wspomniane metody **readObject** i **writeObject**.

– Odpowiedź E: przenoszenie obiektu, który nie jest serializowalny, jest niemożliwe.

– Odpowiedź F: specyfikacja nie gwarantuje, że wymienione metody zostaną wywołane.

- ☐ G. Jeśli atrybut sesji implementuje interfejs **HttpSessionActivationListener**, wymagania kontenera ograniczają się do konieczności informowania obiektów nasłuchujących o aktywacji sesji na nowym serwerze.

– Odpowiedź G: kontener musi dodatkowo wystać informację o dezaktywacji (pasywacji).

- 59 Deskryptor wdrożenia aplikacji internetowej Javy EE deklaruje kilka filtrów, których adresy URL pasują do bieżącego żądania. Deskryptor deklaruje też kilka filtrów, których znaczniki **<servlet-name>** są dopasowywane do tego samego żądania.

Specyfikacja serwetów, rozdział 6.; hf 710.

Które zdania o regułach wywoływania przez kontener filtrów dla tego żądania są prawdziwe? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. Zostaną wywołane tylko filtry z pasującymi znacznikami **<servlet-name>**.
- ☐ B. Spośród wszystkich filtrów z pasującymi adresami URL zostanie wywołany tylko pierwszy filtr.
- ☐ C. Spośród wszystkich filtrów z pasującymi znacznikami **<servlet-name>** zostanie wywołany tylko pierwszy filtr.
- ☐ D. Filtry z pasującymi znacznikami **<servlet-name>** zostaną wywołane przed filtrami z pasującymi adresami URL.
- ☐ E. Zostaną wywołane wszystkie filtry z pasującymi adresami URL, ale kolejność tych wywołań jest niezdefiniowana.
- ☒ F. Zostaną wywołane wszystkie filtry z pasującymi adresami URL w kolejności zgodnej z porządkiem ich deklaracji w deskrytorze wdrożenia.

– W pierwszej kolejności kontener wywoła wszystkie filtry z pasującymi adresami URL (zgodnie z porządkiem ich deklaracji w deskrytorze wdrożenia), by następnie wywołać filtry z pasującymi znacznikami **<servlet-name>** (także w kolejności zgodnej z porządkiem deklaracji).

- 60 Które zdania o parametrach inicjalizacji serwetu i parametrach inicjalizacji kontekstu są prawdziwe? (Zaznacz wszystkie prawidłowe odpowiedzi).

Specyfikacja serwetów, rozdziały 9. i 13.; hf 157 – 160.

- ☒ A. Znaczniki deskryptora wdrożenia deklarujące oba typy parametrów zawierają znaczniki **<param-name>** i **<param-value>**.
- ☐ B. Znaczniki deskryptora wdrożenia deklarujące oba typy parametrów są umieszczane bezpośrednio pod znacznikiem **<web-app>**.
- ☒ C. Metody zwracające wartości parametrów inicjalizacji obu typów nazwano **getInitParameter**.
- ☐ D. Dostęp do parametrów obu typów można uzyskiwać bezpośrednio z poziomu kodu JSP.
- ☐ E. Tylko zmiany parametrów inicjalizacji kontekstu (wprowadzane w deskrytorze wdrożenia) są uwzględniane bez konieczności ponownego wdrożenia aplikacji internetowej.

– Odpowiedź B: tylko znacznik **<context-param>** jest umieszczany bezpośrednio pod znacznikiem **<web-app>**.

– Odpowiedź D: bezpośredni dostęp z poziomu kodu JSP można uzyskiwać tylko do parametrów kontekstu.

– Odpowiedź E: zmiany wprowadzane w deskrytorze wdrożenia nigdy nie są uwzględniane dynamicznie

- 61 Programista JSP chce dołączyć zawartość pliku **copyright.jsp** do wszystkich właściwych stron JSP swojej aplikacji.

Specyfikacja JSP 2.0, podpunkt 1.1.0.5.

Które mechanizmy umożliwiają takie dołączenie? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☒ A. **<jsp:directive.include file="copyright.jsp" />**
- ☒ B. **<%@ include file="copyright.jsp" %>**
- ☐ C. **<%@ page include="copyright.jsp" %>**
- ☒ D. **<jsp:include page="copyright.jsp" />**
- ☐ E. **<jsp:insert file="copyright.jsp" />**

– Odpowiedź A jest prawidłowa, ponieważ reprezentuje poprawną składnię dokumentów JSP.

– Odpowiedź B jest prawidłowa, ponieważ reprezentuje poprawną składnię stron JSP.

– Odpowiedź C jest błędna, ponieważ za pomocą dyrektywy **page** nie można importować treści zewnętrznych.

– Odpowiedź D jest prawidłowa, ponieważ ta akcja standardowa dołącza treść zewnętrzną w czasie wykonywania aplikacji.

– Odpowiedź E jest błędna, ponieważ użyta akcja standardowa nie istnieje.

- 62** Pracujesz nad aplikacją zarządzającą kontami klientów operatora telefonii stacjonarnej, telewizji kablowej i internetu. Znaczna część stron tej aplikacji zawiera funkcjonalność wyszukiwania. Pole wyszukiwanej frazy powinno co prawda wyglądać tak samo na wszystkich stronach, jednak niektóre strony powinny ograniczać zakres przeszukiwania do kont telefonicznych, telewizji kablowej lub internetowych.

Specyfikacja JSP 2.0, podrozdziały 5.4 i 5.6; hf 400 – 408.

Dysponujesz odrębną stroną JSP nazwaną Search.jsp:

1. `<form action="/search.go">`
2.     Znajdź konto `${param.accountType}`:
3.     `<input type="text" name="searchText"/>`
4.     `<input type="hidden" name="accountType" value="${param.accountType}"/>`
5.     `<input type="submit" value="Szukaj"/>`
6. `</form>`

Którego z poniższych znaczników należałoby użyć w kodzie strony JSP oferującej możliwość wyszukiwania kont telewizji kablowej?

- ☐ A. `<jsp:include page="Search.jsp" accountType="telewizji kablowej"/>`
- ☒ B. `<jsp:include page="Search.jsp">`  
       `<jsp:param name="accountType" value="telewizji kablowej"/>`  
       `</jsp:include>`
- ☐ C. `<jsp:include file="Search.jsp" accountType="telewizji kablowej"/>`
- ☐ D. `<jsp:include file="Search.jsp">`  
       `<jsp:attribute name="accountType" value="telewizji kablowej"/>`  
       `</jsp:include>`

– Odpowiedź A: znacznik `<jsp:include>` nie może zawierać atrybutu nazwanego `accountType`.

– Odpowiedź B: `${param.accountType}` odnajdzie nasz parametr "telewizji kablowej" przekazany za pomocą znacznika `<jsp:params>`.

– Odpowiedzi C i D: element `<jsp:include>` wykorzystuje atrybut `page`. Atrybut `file` stosuje się w dyrektywach `include`.

- 63** W czasie testów rozmaitych znaczników i skryptletów programista zdecydował się utworzyć następujący kod JSP:

Specyfikacja JSP 2.0, punkt 1.3.1 i podrozdział 1.5; hf 304, 483.

1. `<% request.setAttribute("name", "świecie"); %>`
2. `<!-- Test -->`
3. `<c:out value='Witaj ${name}'/>`

Ku jego zdziwieniu, w odpowiedzi na żądanie tej strony przeglądarka niczego nie wyświetla. Co programista znajdzie w kodzie źródłowym zwróconej strony HTML?

- ☐ A. `<!-- Test -->`
- ☐ B. `<!-- Test -->`  
       `<c:out value='Witaj ${name}'/>`
- ☒ C. `<!-- Test -->`  
       `<c:out value='Witaj świecie'/>`
- ☐ D. Nie zostaną zwrócone żadne dane wynikowe.

– Odpowiedź C: wyrażenie `${name}` języka EL jest przetwarzane, ale znacznik `<c:out>` nie jest rozpoznawany — jest traktowany jako tekst szablonowy, ponieważ w danej stronie JSP nie zadeklarowano dyrektywy taglib

64

Aplikacja usług randkowych zadaje swoim użytkownikom serie pytań. Przyjmijmy, że istnieje już atrybut zasięgu sesji typu **HashMap** nazwany **compatibilityProfile**, w którym składujemy identyfikatory poszczególnych pytań wraz z udzielonymi odpowiedziami.

Specyfikacja JSTL 1.1,  
podrozdział 4.3;  
hf 455 – 457.

Dysponujemy następującym kodem:

```
22. <% ((java.util.HashMap)request.getSession().getAttribute(
 "compatibilityProfile")).put(
23. request.getParameter("questionIdSubmitted"),
24. request.getParameter("answerSubmitted"));
25. %>
```

Jak należałoby zastąpić tę konstrukcję bez stosowania skryptletów?  
(Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. `<c:map target="${compatibilityProfile}"` – Odpowiedź A: `<c:map>` nie jest prawdziwym znacznikiem.
- `key="${param.questionIdSubmitted}"`
- `value="${param.answerSubmitted}"/>`
- ☐ B. `<jsp:useBean id="compatibilityProfile" class="java.util.HashMap"` – Odpowiedź B: znacznik `<jsp:useBean>` można stosować tylko dla komponentów, nie dla map!
- `scope="session">`
- `<jsp:setProperty name="compatibilityProfile"`
- `property="${param.questionIdSubmitted}"`
- `value="${param.answerSubmitted}"/>`
- `</jsp:useBean>` – Odpowiedź C: samo wyrażenie języka EL nie może ustawić wartości obiektu.
- ☐ C. `${compatibilityProfile[param.questionIdSubmitted]} = param.answerSubmitted}`
- ☒ D. `<c:set target="${compatibilityProfile}"` – Odpowiedź D: znacznik `<c:set>` można wykorzystywać do umieszczania wartości w mapie.
- `property="${param.questionIdSubmitted}"`
- `value="${param.answerSubmitted}"/>`

- 65** Programista pracuje nad filtrem dla swojej aplikacji internetowej Javy EE. Dysponujemy następującym fragmentem kodu: API; hf 707.

```

7. public class MojFiltr implements Filter {
8. public void init(FilterConfig config) throws FilterException { }
9.
10. public void doFilter(HttpServletRequest request,
11. HttpServletResponse response,
12. FilterChain chain)
13. throws IOException, ServletException { }
14.
15. }
```

Która z zaproponowanych poniżej zmian jest niezbędna do utworzenia prawidłowego filtra?  
(Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. Żadne zmiany nie są konieczne.
- ☒ B. Należy dodać metodę **destroy()**. – Odpowiedź C: metoda doFilter() musi wywołać przynajmniej metodę chain.doFilter() (nawet jeśli nie podejmuje żadnych innych działań).
- ☒ C. Należy zmienić ciało metody **doFilter()**.
- ☒ D. Należy zmienić sygnaturę metody **init()**. – Odpowiedź D: metoda init() generuje wyjątek ServletException.
- ☒ E. Należy zmienić argumenty metody **doFilter()**. – Odpowiedź E: metoda doFilter() otrzymuje na wejściu obiekty ServletRequest i ServletResponse.
- ☐ F. Należy zmienić listę wyjątków generowanych przez metodę **doFilter()**.

- 66** Twoja firma chce dodać do istniejącej aplikacji stronę powitalną nazwaną **SplashAd.jsp**, która będzie reklamowała użytkownikom odwiedzającym daną witrynę oferty innych przedsiębiorstw. Na stronie powitalnej użytkownicy będą mieli do dyspozycji pole wyboru *Nie wyświetlaj więcej tej oferty* oraz przycisk *Przejdź do mojego konta*. Jeśli użytkownik wyśle ten formularz z zaznaczonym polem wyboru, docelowy serwet ustawi cookie (znacznik kontekstu klienta) nazwane **skipSplashAd**, po czym zwróci sterowanie do głównej strony JSP. Specyfikacja JSP 2.0, podrzdział 5.5; hf 409 – 410.

Główna strona JSP będzie odpowiedzialna za skierowanie danego żądania na stronę powitalną. Jaki fragment kodu należy dodać na początek strony głównej, aby odsyłała strony powitalne tym użytkownikom, którzy jeszcze nie zaznaczyli odpowiedniego pola wyboru?

- ☒ A. `<c:if test="${empty cookie.skipSplashAd and pageContext.session.new}">`  
`<jsp:forward page="SplashAd.jsp"/>`  
`</c:if>` – Odpowiedź B: atrybut flush nie rozwiązuje problemu.
- ☐ B. `<jsp:forward page="SplashAd.jsp" flush="${empty cookie.skipSplashAd}"/>`
- ☐ C. `<jsp:redirect page="SplashAd.jsp"/>` – Odpowiedzi C i D: znacznik <jsp:redirect> nie istnieje.
- ☐ D. `<jsp:redirect file="SplashAd.jsp"/>`
- ☐ E. `<% if(cookie.get("skipSplashAd") == null && session.isNew()){ %>`  
`<jsp:forward page="SplashAd.jsp"/>`  
`<% } %>` – Odpowiedź E: zaproponowany skryptlet jest nieprawidłowy. cookie jest obiektem domyślnym w języku EL, ale nie w skryptletach.

**67** Programista chce zaimplementować interfejs **ServletContextListener**.  
Dysponujemy następującym fragmentem deskryptora wdrożenia:

*API; specyfikacja serwletów,  
dodatek A; hf 171 – 174.*

```
101. <!-- tutaj wstaw znacznik1 -->
102. <param-name>mojParametr</param-name>
103. <param-value>mojaWartosc</param-value>
104. <!-- tutaj zamknij znacznik1 -->
105. <listener>
106. <!-- tutaj wstaw znacznik2 -->
107. com.wickedlysmart.MojaKlasaNasluchujaca
108. <!-- tutaj zamknij znacznik2 -->
109. </listener>
```

Poniżej przedstawiono pseudokod samej klasy nasłuchującej:

```
5. // tutaj powinna się znaleźć deklaracja pakietu i instrukcje importu
6. public class MojaKlasaNasluchujaca implements ServletContextListener {
7. // tutaj wstaw metodę metoda1
8. // tutaj zakończ odpowiednią metodę
9. }
```

Które zdania są prawdziwe? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. Fragment deskryptora wdrożenia w tej formie nie może być prawidłowy.
- ☒ B. Znacznikiem **znacznik1** powinien być **<context-param>**. *– Czasem po prostu musisz zapamiętać pewne reguły.*
- ☐ C. Znacznikiem **znacznik1** powinien być **<servlet-param>**.
- ☒ D. Znacznikiem **znacznik2** powinien być **<listener-class>**.
- ☐ E. Znacznikiem **znacznik2** powinien być **<servlet-context-class>**.
- ☐ F. Metodą **metoda1** powinna być **initializeListener**.
- ☒ G. Metodą **metoda1** powinna być **contextInitialized**.

- 68 Witryna internetowa *wickedlysmart.com* udostępnia prawidłowo wdrożoną aplikację internetową Javy EE. Deskryptor wdrożenia tej aplikacji zawiera następujący fragment:

Specyfikacja serwetów,  
rozdział 9.; hf 625.

```
<welcome-file-list>
 <welcome-file>welcome.html</welcome-file>
 <welcome-file>howdy.html</welcome-file>
 <welcome-file>index.html</welcome-file>
</welcome-file-list>
```

Poniżej przedstawiono fragment struktury katalogów tej aplikacji:

```
MojaAplikacjaInternetowa
|
|-- index.html
|
|-- welcome
| |-- welcome.html
|
|-- foobar
| | howdy.html
```

Założmy, że do opisywanej aplikacji trafiają dwa następujące żądania:

**http://www.wickedlysmart.com/MojaAplikacjaInternetowa/foobar**

**http://www.wickedlysmart.com/MojaAplikacjaInternetowa**

Która z wymienionych sekwencji odpowiedzi zostanie zwrócona?

- ☐ A. **howdy.html** i kod 404
- ☐ B. **index.html** i kod 404
- ☐ C. **welcome.html** i kod 404
- ☒ D. **howdy.html** i **index.html**
- ☐ E. **index.html** i **index.html**
- ☐ F. **howdy.html** i **welcome.html**
- ☐ G. **welcome.html** i **index.html**

– Odpowiedź D: jeśli deskryptor wdrożenia nie zawiera elementów odwzorowania serwetu, zostanie przeszukany katalog wskazany w żądaniu i zwrócony pierwszy odnaleziony plik z listy zadeklarowanej za pomocą znaczników `<welcome-file>`.

69 Twoja aplikacja internetowa obejmuje prawidłowy deskryptor wdrożenia z pojedynczym znacznikiem **<security-constraint>**. W ciele tego znacznika zadeklarowano:

*Specyfikacja serwletów,  
podrozdział 12.8;  
hf 664 – 665.*

– pojedynczą metodę **GET** protokołu HTTP.

Wszystkie zasoby Twojej aplikacji umieszczono w katalogach **katalog1** i **katalog2**, a w deskryptorze wdrożenia zdefiniowano tylko dwie role: **NOWICJUSZ** oraz **EKSPERT**.

Które zdania prawidłowo opisują adresy URL i znaczniki ról niezbędne do ograniczenia dostępu do zasobów w katalogu **katalog2** użytkowników przypisanych do roli **NOWICJUSZ**? (Zaznacz wszystkie prawidłowe odpowiedzi).

- ☐ A. Pojedynczy znacznik adresu URL powinien deklarować katalog **katalog1**, a pojedynczy znacznik roli powinien deklarować rolę **EKSPERT**.
- ☒ B. Pojedynczy znacznik adresu URL powinien deklarować katalog **katalog2**, a pojedynczy znacznik roli powinien deklarować rolę **EKSPERT**.
- ☐ C. Pojedynczy znacznik adresu URL powinien deklarować katalog **katalog1**, a pojedynczy znacznik roli powinien deklarować rolę **NOWICJUSZ**.
- ☐ D. Pojedynczy znacznik adresu URL powinien deklarować katalog **katalog2**, a pojedynczy znacznik roli powinien deklarować rolę **NOWICJUSZ**.
- ☐ E. Jeden znacznik adresu URL powinien deklarować wartość **ANY**, a jego znacznik roli powinien deklarować rolę **EKSPERT**. Drugi znacznik adresu URL powinien deklarować katalog **katalog2**, a jego znacznik roli powinien deklarować rolę **NOWICJUSZ**.
- ☐ F. Jeden znacznik adresu URL powinien deklarować oba katalogi, a jego znacznik roli powinien deklarować rolę **EKSPERT**. Drugi znacznik adresu URL powinien deklarować katalog **katalog1**, a jego znacznik roli powinien deklarować rolę **NOWICJUSZ**.

*– Pamiętaj, że w deskryptorze wdrożenia zawsze deklarujesz ograniczenia.*



# Skorowidz

- \_jspService(), 324, 333, 335, 378, 433
- <!-- -->, 35, 332
- <% %>, 312, 316
- <%-- --%>, 332
- <%@ %>, 115, 316
- <%@ page import= %>, 315
- <%= %>, 316
- <a>, 35
- <align>, 35
- <attribute>, 533
- <auth-constraint>, 696, 697, 699
- <auth-method>, 691, 706
- <body>, 35, 439
- <br>, 35
- <c:catch>, 500
  - var, 501
- <c:choose>, 482
- <c:forEach>, 474, 475, 476
  - varStatus, 476
  - zagnieżdżanie, 477
- <c:if>, 479, 481
- <c:import>, 488, 489, 529
- <c:otherwise>, 482
- <c:out>, 471
  - default, 473
- <c:param>, 491
- <c:remove>, 486
- <c:set>, 483, 485
  - komponenty, 484
  - mapy, 484
  - target, 484
- <c:url>, 267, 493
  - <c:param>, 494
  - łańcuch zapytania, 494
- <c:when>, 482
- <center>, 35
- <context-param>, 185, 186, 188, 202, 206
- <ejb-local-ref>, 658, 659
- <ejb-ref>, 658, 659
- <ejb-ref-name>, 658, 659
- <ejb-ref-type>, 658, 659
- <el-ignored>, 350, 352
- <env-entry>, 660
- <env-entry-name>, 660
- <env-entry-value>, 660
- <error-code>, 498
- <error-page>, 497, 498, 654
- <exception-type>, 498, 654
- <extension>, 661
- <filter>, 738
- <filter-class>, 738
- <filter-mapping>, 738, 739
- <filter-name>, 738
- <form>, 35, 145
  - method="POST", 145
- <function>, 421, 423
- <function-class>, 421
- <function-name>, 423
- <function-signature>, 421, 423
- <h1>, 35
- <head>, 35
- <home>, 658, 659
- <html>, 35, 39, 439
- <http-method>, 695, 698
- <img>, 36
- <init-param>, 178, 182, 188
- <input type>, 35
  - type="SUBMIT", 145
- <jsp:attribute>, 509
- <jsp:declaration>, 657
- <jsp:directive>, 657
- <jsp:expression>, 657
- <jsp:forward>, 441, 444
- <jsp:getProperty>, 377, 395

<jsp:include>, 432, 433, 434, 447, 488, 529  
nazwy atrybutów, 437  
pierwsze żądanie, 436  
<jsp:param>, 440, 531  
<jsp:scriptlet>, 657  
<jsp:setProperty>, 379, 391, 483  
param, 388  
<jsp:text>, 657  
<jsp:useBean>, 377, 378  
ciało, 380, 381  
class, 385  
scope, 385  
typ bez klasy, 384  
type, 383  
<jsp-file>, 338  
<listener>, 197, 202, 206  
<listener-class>, 202  
<load-on-startup>, 656, 803  
<local>, 658  
<local-home>, 658, 659  
<location>, 498, 654  
<login-config>, 691, 706  
<mime-mapping>, 661  
<mime-type>, 661  
<p>, 35  
<param-name>, 178  
<param-value>, 178  
<remote>, 658  
<required>, 533  
<role>, 692  
<role-name>, 692, 696  
<rtexprvalue>, 508, 533  
<scripting-invalid>, 349, 352  
<security-constraint>, 693, 694  
<security-role>, 692, 703  
<security-role-ref>, 703  
<select>, 570, 578  
<servlet>, 58, 76, 106, 644  
<servlet-class>, 58, 77  
<servlet-mapping>, 58, 76, 106, 644  
<servlet-name>, 58, 76, 106, 107, 646, 738

<session-config>, 273  
<session-timeout>, 273  
<table>, 475  
<td>, 475  
<title>, 35  
<tlib-version>, 421  
<tomcat-users>, 692  
<tr>, 475  
<transport-guarantee>, 712, 714  
<uri>, 421, 422, 512  
<url-pattern>, 58, 106, 646, 647, 738  
<user-data-constraint>, 712  
<web-app>, 92  
<web-resource-collection>, 694, 696  
<welcome-file>, 650  
<welcome-file-list>, 650

## A

Action, 799  
Action.execute(), 797  
ActionErrors, 799  
ActionForward, 799, 801  
ActionServlet, 799, 801  
addCookie(), 279, 281  
addHeader(), 161, 168, 281  
adres IP, 48, 258  
adres poczty elektronicznej, 177  
adres URL, 48  
odwzorowanie na serwlety, 74  
akcje, 351  
komponenty JavaBean, 376, 446  
standardowe, 351  
aktualizacja, 684  
ANT, 632  
Apache, 38, 50  
aplikacja przeglądarki, 33  
aplikacja serwera WWW, 32, 53  
aplikacje internetowe, 29, 65, 97, 768  
MVC, 95, 114  
widok, 98  
aplikacje pomocnicze serwera WWW, 56

- application, 326
- applicationScope, 413
- architektura aplikacji internetowej, 65, 66
- architektura MVC, 97
- archiwum internetowe, 639
- ArrayList, 233
- ataki XSS, 472
- atrybuty, 132, 175, 183, 200, 213, 214, 373
  - body-content, 536
  - class, 385
  - default, 473
  - dynamiczne, 584
  - escapeXml, 471
  - interfejs API, 217
  - niełańcuchowe, 374
  - odczytywanie, 341
  - odnajdywanie, 341
  - param, 388
  - type, 385
  - ustawianie, 218, 341
  - zasięg, 215, 216, 237
- atrybuty JSP, 339
- atrybuty kontekstu, 215
  - bezpieczeństwo wątków, 220
  - przetwarzanie, 222
  - synchronizacja metody obsługującej żądania, 224
  - wątki, 222
- atrybuty plików znaczników, 532
- atrybuty sesji, 215, 282
  - bezpieczeństwo wątków, 226
- atrybuty sesyjne, 200
- atrybuty znaczników HTML, 35
- atrybuty żądania, 215, 232, 233
  - bezpieczeństwo przetwarzania wielowątkowego, 232
  - przydzielanie żądań, 233
- attribute, 534
- attributeAdded(), 290, 293
- attributeRemoved(), 290, 293
- attributeReplaced(), 290, 293
- autoFlush, 343

- automatyczna konwersja łańcuchów, 392
- automatyzacja znacznika select, 570
- autoryzacja, 681, 683, 685, 686, 691
  - definiowanie ograniczeń, 693
  - definiowanie ról, 692

## B

- BASIC, 705
- baza danych, 192, 255
- bezpieczeństwo, 75, 677, 682, 688
  - aplikacje internetowe, 677
  - autoryzacja, 681, 683, 691
  - HTTPS, 710
  - integralność danych, 681
  - poufność, 681, 683
  - przesyłanie danych, 710
  - serwlety, 681
  - uwierzytelnianie, 681, 683, 684, 686
  - zagrożenia, 679
- bezpieczeństwo deklaratywne, 69
- bezpieczeństwo wątków, 220, 222, 226, 232
  - atrybuty, 227
- bezpośrednie udostępnianie stron JSP, 642
- bezpośrednie udostępnianie zawartości statycznej, 642
- bezkryptowe strony JSP, 371
- biblioteki znaczników, 342
- blokowanie atrybutów
  - poziom kontekstu, 225
  - poziom serwletu, 225
- body-content, 536
- BodyContent, 592
- BodyTag, 558, 590, 591
- BodyTagSupport, 558, 590, 591, 592
- brak konwersji elementów XML, 471
- budowanie aplikacji, 102
- buffer, 343
- buforowanie zawartości znacznika, 592
- Business Delegate, 781, 783, 788, 789, 806

### C

- certyfi­kat klucza publicz­nego, 705
- CGI, 55, 63
- chronione połączenia warstwy transportowej, 709
- ciało komunikatu, 44
- ciało znaczników, 510
- ciasteczka, 150, 260, 278, 418
  - Cookie, 279
  - długość życia znacznika kontekstu, 279
  - HttpServletRequest, 279
  - HttpServletResponse, 279
  - interfejs API serwetów, 279
  - JSESSIONID, 278
  - niestandardowe, 280
  - odczytywanie z żądania klienta, 279
  - wyłączona obsługa, 264
  - wysyłanie do klienta, 279
- class, 385
- CLIENT-CERT, 705
- CONFIDENTIAL, 712
- config, 326
- CONNECT, 41, 137
- container, 65, 67
- contentType, 318, 343
- contextDestroyed(), 194, 198, 199
- contextInitialized(), 198
- cookie, 413, 418
- Cookie, 260, 279
- CORBA, 56
- count, 478
- cykl życia filtrów, 734, 736
- cykl życia obiektu obsługi znacznika prostego, 544
- cykl życia serwetu, 124, 127
  - interfejs API, 152
- cykl życia sesji, 282
- cykl życia strony JSP, 334
- cykl życia znacznika klasyczny, 564

### D

- dane wyjściowe, 160
- DataSource, 192

- DD, 58, 76, 92
- default, 473
- default.jsp, 650
- definiowanie
  - role, 692
  - żądania, 33
- deklaracja
  - dane środowiskowe JNDI, 660
  - filtry, 738
  - JSP, 322, 323
  - konwersja elementów XML, 471
  - metody, 323
  - odzworowanie filtra na nazwę serwetu, 738
  - odzworowanie filtra na wzorec URL, 738
  - strona błędu, 654
  - zawartość znacznika dla pliku znacznika, 536
  - zmienne, 320, 323
- deklaratywna kontrola nad aplikacją, 772
- deklaratywna weryfikacja poprawności, 797
- deklaratywne bezpieczeństwo programowe, 703
- deklaratywne dostosowywanie aplikacji internetowych, 77
- deklaratywne określanie bezpieczeństwa, 687
- dekonstrukcja języka wyrażeń JSP, 397
- dekorator odpowiedzi, 754
- delegat biznesowy, 781, 789, 806
- DELETE, 41, 137
- Deployment Descriptor, 58
- deskryptor biblioteki znaczników, 420
- deskryptor wdrożenia, 58, 76, 92, 104, 632, 664
  - <auth-constraint>, 696, 697, 699
  - <auth-method>, 691, 706
  - <context-param>, 185
  - <ejb-local-ref>, 658, 659
  - <ejb-ref>, 658, 659
  - <ejb-ref-name>, 658, 659
  - <ejb-ref-type>, 658, 659
  - <env-entry>, 660
  - <env-entry-name>, 660
  - <env-entry-value>, 660
  - <error-code>, 498
  - <error-page>, 497, 498, 654

- <exception-type>, 498, 654
- <extension>, 661
- <filter>, 738
- <filter-class>, 738
- <filter-mapping>, 738, 739
- <filter-name>, 738
- <home>, 658, 659
- <http-method>, 695
- <init-param>, 178
- <jsp-file>, 338
- <listener>, 197, 202
- <listener-class>, 202
- <load-on-startup>, 656
- <local>, 658
- <local-home>, 658, 659
- <login-config>, 691, 706
- <mime-mapping>, 661
- <mime-type>, 661
- <param-name>, 178
- <param-value>, 178
- <remote>, 658
- <role-name>, 692, 696
- <scripting-invalid>, 349
- <security-constraint>, 693, 694
- <security-role>, 692
- <servlet>, 58, 76
- <servlet-class>, 58
- <servlet-mapping>, 58, 76
- <servlet-name>, 58, 76
- <session-timeout>, 273
- <transport-guarantee>, 712, 714
- <url-pattern>, 58
- <user-data-constraint>, 712
- <web-resource-collection>, 694, 696
- <welcome-file>, 650
- <welcome-file-list>, 650
- adres poczty elektronicznej, 177
- filtry, 733
- inicjalizacja serwletu, 656
- komponenty EJB, 658
- limit czasowy sesji, 273
- obiekt nasłuchujący, 202
- obiekt nasłuchujący zdarzeń kontekstu, 197
- odzworowanie adresu URL na serwlet, 76
- parametry inicjalizacji kontekstu, 185
- parametry inicjalizacji serwletu, 178
- pliki powitalne, 650
- reguły ograniczeń, 694
- strony błędów, 654
- web.xml, 105
- deskryptor wdrożenia Struts, 802
- destroy(), 66, 125, 736
- DIGEST, 705
- długość życia znacznika kontekstu, 279
- doAfterBody(), 563, 565, 567, 569, 590, 594
- dodawanie nagłówka odpowiedzi, 161
- doEndTag(), 560, 563, 564, 567, 590, 594
- doFilter(), 736, 737, 742
- doGet(), 65, 67, 71, 72, 124, 127, 133, 136, 146, 218, 219
- doInitBody(), 590, 594
- dokumenty
  - JSP, 657
  - PDF, 32
  - XML, 657
- dołączanie, 447, 529, 530
  - zasoby spoza aplikacji internetowej, 489
- domyślna metoda HTTP, 146
- domyślne obiekty JSP, 326
- doPost(), 65, 67, 71, 72, 111, 126, 127, 133, 136, 146, 148
- doStartTag(), 560, 563, 564, 567, 569, 590, 592, 594
- dostęp do istniejącej sesji, 263
- dostęp do list, 402
- dostęp do parametrów inicjalizacji, 182
- dostęp do parametrów inicjalizacji serwletu, 183
- dostęp do tablic, 402
- dostęp do zasobów, 643
- dostęp do zawartości znacznika, 590
- dostęp do żądanych zasobów, 139
- dostosowywanie dołączanej zawartości, 440, 490
- doTag(), 507, 542, 544, 548, 551, 586
- dynamic-attributes, 589
- DynamicAttributes, 584, 585, 588

dynamiczne strony WWW, 54  
dyrektywy, 315, 316, 322, 342  
    attribute, 534  
    include, 342, 431, 434  
    page, 315, 318, 342  
    tag, 536  
    taglib, 342, 420, 530  
dziedzina bezpieczeństwa, 691

## E

egzamin SCWCD, 24  
EJB, 77, 93, 255, 374, 380, 658  
EL, 347, 348, 371, 396, 428, 469  
    \${foo.1}, 407  
    cookie, 418  
    dostęp do list, 402  
    dostęp do właściwości, 398  
    funkcje, 420  
    funkcje statyczne, 422  
    header, 415  
    ignorowanie wyrażeń, 350  
    indeksy łańcuchowe tablic i list, 402  
    informacje z ządania, 415  
    initParam, 418  
    listy, 399, 402  
    mapy zasięgów, 416  
    metoda ządania HTTP, 415  
    nazwy identyfikatorów, 398  
    nieprzetworzony kod HTML, 412  
    obiekty domyślne, 413  
    obiekty domyślne zasięgów, 417  
    odczytywanie ciasteczek klienta, 418  
    odwzorowywanie wartości, 398  
    operator [], 399, 400, 404  
    operator kropki, 398, 404  
    operatory, 424  
    param, 414  
    parametry inicjalizacji, 418  
    parametry ządania, 414  
    paramValues, 414  
    requestScope, 416, 417

    składnia, 397  
    tablice, 399, 401, 402  
    TLD, 420, 421  
    wartości null, 427  
    wartość mapy, 398  
    wyrażenia, 348, 397  
    wyrażenia zagnieżdżone, 406  
    znaczniki niestandardowe, 547  
elastyczność, 770  
elementy XML, 471  
encodeRedirectURL(), 267  
encodeURL(), 266, 267, 268  
Enterprise JavaBeans, 77, 93, 374  
errorPage, 343, 497  
escapeXml, 471  
EVAL\_BODY\_AGAIN, 565  
EVAL\_BODY\_BUFFERED, 591, 592  
EVAL\_PAGE, 564, 567  
EVAL\_PAGE\_INCLUDE, 564  
exception, 326, 499  
execute(), 801  
Expression Language, 347  
extends, 343

## F

Filter, 732, 735  
FilterChain, 735, 736  
filtr przechwytyjący, 809  
filtrowanie odpowiedzi, 744  
filtry, 729, 732  
    cykl życia, 734, 736  
    deklaracja, 738  
    destroy(), 736  
    doFilter(), 736  
    init(), 736  
    kompresja odpowiedzi, 750  
    kompresja wyników, 741  
    konfiguracja, 733, 738  
    kontener, 734  
    odpowiedzi, 732, 741, 742  
    określanie kolejności, 738

- stos wywołań, 737
- tworzenie, 732, 735
- zastosowania, 732
- żądania, 732
- findAncestorWithClass(), 602
- findAttribute(), 341, 605
- fizyczna struktura katalogów, 645
- flush(), 235, 444
- forEach, 474, 589
- FORM, 705, 707
- formatowanie stron HTML, 60, 62
- formularze, 54, 146
  - parametry, 147
  - pojedynczy parametr, 147
  - przesyłanie pojedynczego parametru, 147
  - przesyłanie wielu parametrów, 148
  - wiele parametrów, 148
- forward(), 166, 233, 234
- framework Struts, 795
- Front Controller, 765, 797, 811
- funkcje EL, 420

**G**

- GenericServlet, 72, 108, 126, 129, 135, 152
- GET, 40, 47, 63, 127, 136, 137, 138, 142
  - anatomia żądania, 43
  - dane parametrów, 139
  - łączna liczba znaków, 42
  - ścieżka do zasobu, 43
- getAttribute(), 190, 201, 205, 214, 217, 277, 340, 341
- getAttributeNames(), 190, 217
- getAttributeNamesInScope(), 340
- getContextParameter(), 188
- getCookies(), 150, 279, 418
- getCreationTime(), 272
- getHeader(), 150, 151, 415
- getInitParameter(), 178, 185, 186, 188, 190, 191, 201, 214
- getInitParameterNames(), 190
- getInputStream(), 150
- getIntHeader(), 151

- getJspBody.invoke(), 542
- getJspContext(), 550
- getLastAccessedTime(), 272
- getLiczba(), 317
- getLocal(), 151
- getLocalPort(), 151
- getMaxInactiveInterval(), 272
- getMethod(), 150, 415
- getName(), 279, 293
- getOutputStream(), 157, 158, 160, 168, 749
- getParameter(), 112, 147, 148, 151, 153, 183, 387
- getParent(), 596, 597, 598, 599, 602
- getPorada(), 506
- getRemotePort(), 151
- getRemoteUser(), 702
- getRequestDispatcher(), 166, 190, 233, 234, 280
- getResource(), 643
- getResourceAsStream(), 157, 643
- getServerPort(), 151
- getServletConfig(), 178, 179, 186, 191, 338
- getServletContext(), 185, 186, 191, 201, 338, 339
- getServletName(), 182
- getSession(), 150, 261, 262, 263, 264, 266, 271, 293
- getUserPrincipal(), 702
- getValue(), 293
- getWriter(), 79, 112, 154, 158, 160, 177
- globalna obsługa wyjątków, 797
- gniazdo połączenia, 68
- graficzny interfejs użytkownika, 81
- GUI, 29, 81
- GZIPOutputStream, 750
- GZIPServletOutputStream, 753

## H

- HEAD, 41, 137, 142
- header, 413, 415
- headerValues, 413
- hierarchia katalogów serwerów WWW, 48
- hiperłącza, 493
- HTML, 34, 39, 470

- HTTP, 30, 34, 38, 63, 153, 695
  - kompresja, 751
  - metody, 40
  - odpowiedzi, 38, 39, 45
  - połączenia bezstanowe, 258
  - uwierzytelnianie, 684
  - zadania, 38, 40
- HTTP 1.1, 142
- HTTP GET, 42
- HTTP POST, 44
- HttpJspPage, 333
- HttpRequest, 233
- HTTPS, 258, 710, 716
- HttpServlet, 72, 79, 86, 108, 111, 112, 126, 129, 152, 201, 229
- HttpServletRequest, 70, 108, 112, 123, 134, 135, 150, 153, 168, 213, 217, 279, 326, 702
- HttpServletRequestWrapper, 747
- HttpServletResponse, 70, 108, 112, 123, 134, 135, 153, 154, 168, 267, 279, 326, 745, 746
- HttpServletResponseWrapper, 747
- HttpSession, 150, 217, 228, 255, 271, 272, 282, 285, 326
- HttpSessionActivationListener, 210, 283, 288, 292, 293
- HttpSessionAttributeListener, 210, 283, 292, 293
- HttpSessionBindingEvent, 210, 262, 283, 292, 293
- HttpSessionBindingListener, 210, 211, 284, 292, 293
- HttpSessionEvent, 210, 262, 283, 292, 293
- HttpSessionListener, 210, 262, 283, 289, 292, 293
- HyperText Markup Language, 34
- I**
- IDE, 22
- identyfikator sesji, 259
- IETF, 38
- if, 381
- ignorowanie wyrażeń EL, 350
- IllegalStateException, 235, 277
- implementacja uwierzytelniania, 706
- import, 314, 343
- importowanie, 529
- importowanie pakietów, 315
- include, 342, 430, 431, 434, 447, 488
  - nazwy atrybutów, 437
  - pierwsze żądanie, 435
- include(), 234, 235
- index.html, 51, 650
- info, 343
- informacje o platformie, 150
- informacje o przeglądarce klienta, 150
- informacje o serwerze, 132
- inicjalizacja
  - aplikacje internetowe, 187
  - serwlety, 127, 130, 131
  - strony JSP, 338
- init(), 127, 130, 131, 179, 736
- initParam, 413, 418
- InputStream, 151
- instalacja
  - JSTL 1.1, 468
  - Struts, 804
- INTEGRAL, 712
- integralność danych, 681, 708, 712
- Intercepting Filter, 809
- interfejs
  - BodyTag, 590, 591
  - DynamicAttributes, 584, 585
  - Filter, 732
  - FilterChain, 736
  - HttpServletRequest, 134, 150, 153, 168, 279
  - HttpServletResponse, 134, 154, 279, 746
  - HttpSession, 271, 272
  - HttpSessionActivationListener, 288
  - HttpSessionBindingListener, 211, 284
  - IterationTag, 565
  - JspTag, 543, 558
  - nasłuchujący, 208, 209, 236
  - PageContext, 605
  - Serializable, 200, 288
  - Servlet, 126
  - ServletContext, 190
  - ServletContextListener, 194, 196, 197, 198
  - ServletRequest, 134, 150, 153

- ServletResponse, 134, 154, 160, 746
- SimpleTag, 543
- SingleThreadModel, 229, 231
- Tag, 565
- interfejs API, 108
- interfejs GUI, 29
- interfejs programowy klas obsługi znaczników, 558
- interfejs programowy znaczników prostych, 543
- invalidate(), 272, 282
- invoke(), 550
- isELIgnored, 318, 343, 350, 352
- isErrorPage, 343, 496
- isNew(), 262, 263, 264, 277
- isThreadSafe, 343
- isUserInRole(), 702
- iteracyjne przetwarzanie ciała znacznika, 548
- iteracyjny proces wytwarzania i testowania, 97
- IterationTag, 558, 565, 590

## J

- J2EE, 56, 93, 689
- J2EE 1.4, 93
- JAR, 156, 159, 636
- Java, 22, 55
- JAVA HOME, 22
- Java Naming and Directory Interface, 775
- Java SE 1.5, 22
- JavaBeans, 374
- JavaScript, 371
- javax.servlet.http, 129
- JBoss AS, 93
- język
  - EL, 347, 348, 371, 396, 428
  - HTML, 34
  - UML, 23
- JIT, 54
- JMS, 93
- JNDI, 93, 659, 775, 783
  - kontroler, 778
- jsessionid, 493
- JSESSIONID, 260, 268, 270, 278
- JSP, 61, 62, 309
  - akcje, 351
  - atrybuty, 339
  - biblioteki znaczników, 342
  - deklaracja, 322
  - deklaracja metody, 323
  - deklaracja zmiennej, 323
  - dyrektywy, 322
  - include, 342
  - komentarze, 332
  - obiekty domyślne, 326
  - obsługa atrybutów, 340
  - page, 342
  - PageContext, 340
  - składowe, 322
  - skryptlety, 322
  - taglib, 342
  - wrażenia, 322, 332
  - zmienne, 320
- JSP 2.0, 371, 396
- JSP Standard Tag Library, 469
- JSP Standard Tag Library 1.1, 445
- JspContext, 340, 605
- jspDestroy(), 324, 333
- JspFragment, 550
- jspInit(), 324, 333, 335, 338
- JspPage, 333
- jspService(), 333
- JspTag, 543
- JspWriter, 326
- JSTL, 267, 445, 467, 469
  - <c:catch>, 500
  - <c:choose>, 482
  - <c:forEach>, 475
  - <c:if>, 479
  - <c:import>, 488
  - <c:otherwise>, 482
  - <c:out>, 471
  - <c:remove>, 486
  - <c:set>, 483
  - <c:url>, 493

### JSTL

- <c:when>, 482
- biblioteka formatowania, 503
- biblioteka podstawowa, 503
- biblioteka SQL, 503
- biblioteka XML, 503
- nazwa znacznika, 504
- niestandardowe znaczniki, 506
- odczytywanie deskryptora TLD, 504
- pętle, 474
- składnia znacznika, 504
- strony o błędach, 496
- TLD, 505
- URI biblioteki znaczników, 504
- usuwanie atrybutu, 486
- użycie znacznika spoza biblioteki, 503
- wartości null, 473
- warunkowe dołączanie kodu, 479

JSTL 1.1, 445, 469

jstl.jar, 468

JVM, 56

## K

katalogi, 50

- META-INF, 643
- WEB-INF, 59, 101

klasa atrybutu, 200

klasa modelu, 110

klasa nasłuchująca, 199

klasa obsługi znaczników, 347, 527

- znaczniki niestandardowe, 540

klasa serwletu, 74, 201, 664

klastry, 129

klasy

- BodyTagSupport, 592
- Cookie, 279
- GenericServlet, 72, 126
- HttpServlet, 72, 126
- HttpServletRequest, 153
- HttpServletResponse, 154
- RequestDispatcher, 234
- ServletContextListener, 194
- SimpleTagSupport, 543
- TagSupport, 565, 567

klasy opakowań, 747

klasyczna klasa obsługi znacznika, 560

klient, 33

kod błędu, 63

kod HTML, 36, 39, 103, 470

kod Javy, 68

kod JSP, 115, 321

kod serwletu, 72

kod statusu HTTP, 654

kodowanie adresów URL, 266, 267, 494

kodowanie według interfejsów, 771

kolejkowanie żądań, 229, 230

komentarze, 35

- HTML, 332
- JSP, 332

kompilacja

- aplikacja, 204
- serwlet, 109, 113, 118
- strona JSP, 336

komponenty

- biznesowe, 93
- EJB, 374
- encyjne, 211
- lokalne, 658
- wielokrotnego użytku, 438
- zdalne, 658, 784, 786

komponenty JavaBeans, 374

- akcje, 376
- atrybuty, 377
- inicjalizacja atrybutu, 377
- konwersja typów prostych, 392
- odczytywanie wartości właściwości atrybutu, 377
- parametry żądania, 390
- przypisywanie wartości parametru żądania, 388
- referencje polimorficzne, 382
- tworzenie, 378

- typ bez klasy, 384
- właściwości, 391
- właściwości żądania, 390
- znaczniki standardowych akcji, 393
- kompresja wyników, 741
- komunikaty, 132
- konfiguracja
  - filtry, 738
  - inicjalizacja serwletu, 656
  - licznik sesji, 289
  - ładowanie w trakcie uruchamiania aplikacji, 656
  - obiekt nasłuchujący, 202
  - odwołania do komponentów EJB, 658
  - odwzorowania serwletów, 73
  - pliki powitalne, 650
  - serwlety, 132
  - strony błędów, 498, 654
- kontener, 65, 67, 69, 87, 123
  - autoryzacja, 685
  - bezpieczeństwo deklaratywne, 69
  - filtry, 734
  - kod serwletu, 72
  - lokalizacja plików TLD, 514
  - obsługa ciasteczek, 261
  - obsługa JSP, 69
  - obsługa komunikacji, 69
  - obsługa wielowątkowości, 69
  - obsługa żądania HTTP, 70
  - odwzorowanie adresu URL na serwlet, 73
  - parametry inicjalizacji serwletu, 180
  - przekazywanie żądań, 116
  - rozpoznawanie klienta, 258
  - strona JSP, 324
  - technika łączenia w klastry, 129
  - uwierzytelnianie, 685
  - wybór pliku powitalnego, 653
  - zarządzanie cyklem życia, 69
  - żądania HTTP, 70
- kontener EJB, 93
- kontener WWW, 93
- kontrola deklaratywna, 772

- kontroler, 82, 108, 373, 774, 792, 793
  - frontowy, 765, 811
  - JNDI, 778
  - RMI, 778
  - wywołanie modelu, 112
- konwersacja HTTP, 38
- konwersacja z klientem, 254
- konwersja typów prostych, 392
- korporacyjne wzorce projektowe, 765
- kotwica, 35

## L

- language, 343
- LDAP, 691
- licznik sesji, 289
- limit czasowy sesji, 273
- List, 399
- listy, 402
- log(), 191
- Log4j, 191
- logiczna struktura katalogów, 645
- logika biznesowa, 79, 81, 253
- logowanie, 716
- lokalizacje plików TLD, 514
- lokalizator usługi, 782, 789, 807
- LoopTagStatus, 476

## Ł

- łańcuch zapytania, 63
- łatwość rozbudowy, 770
- łatwość utrzymania, 770
- łączenie stron JSP z kontenerem, 190
- łączenie w klastry, 129

## M

- MANIFEST.MF, 641, 643
- Map, 398
- mechanizm dołączania, 447
- mechanizm równoważenia obciążeń, 285
- mechanizm zabezpieczeń serwletów, 681
- Menu, 601

META-INF, 514, 636, 641, 643  
META-INF/tags, 537, 632  
metapoznanie, 19  
metody, 323  
    \_jspService(), 324, 333, 378  
    destroy(), 125  
    doEndTag(), 560  
    doFilter(), 736  
    doGet(), 71, 72, 124, 127, 133, 136  
    doPost(), 71, 72, 127, 133, 136  
    doStartTag(), 560  
    doTag(), 548  
    getSession(), 262  
    init(), 127, 130, 131, 736  
    jspDestroy(), 324  
    jspInit(), 324, 338  
    service(), 71, 123, 127, 128  
metody HTTP, 40, 124, 136, 137  
    GET, 40, 138  
    POST, 40, 138  
migracja sesji, 285, 286, 288  
MIME, 45, 158  
model, 82, 110, 373, 774  
model jednowątkowy, 229  
model lokalny, 785  
model STM, 229  
model technologii serwletów, 122  
Model View Controller, 81  
model zdalny, 785  
model-widok-kontroler, 82  
modularność, 770  
modyfikacja specyfikacji, 82  
mózg, 17, 21  
MVC, 65, 81, 82, 95, 97, 99, 774, 790, 810  
    kontroler, 108, 792, 793  
    model, 110  
    projektowanie kontrolera, 794  
    stosowanie, 83  
    widok, 794  
MVĆ, 765  
myślenie, 19

## N

nagłówki odpowiedzi, 45, 161  
nasłuchiwanie zdarzeń obiektu ServletContext, 197  
nauka, 18  
nazwy  
    katalogi, 633, 636  
    pliki, 74  
    serwlety, 74  
    ścieżki, 74  
    wdrożenie, 74  
    wewnętrzne, 74, 76  
niebezpieczne znaki, 494  
niepowtarzalne żądania, 140  
nieprzetworzony kod HTML, 412  
niszczenie sesji, 269  
null, 427, 473  
NullPointerException, 205  
numery portów TCP, 49

## O

obiekt akcji, 801  
obiekt odpowiedzi, 154  
    operacje wejścia-wyjścia, 155  
    wysyłanie bajtów, 156  
obiekt sesji, 262  
obiekt transferu, 789, 808  
obiekt żądania, 150, 184  
obiekty domyślne, 326  
obiekty domyślne języka EL, 413  
obiekty nasłuchujące, 175  
    zastosowanie, 193  
obiekty nasłuchujące kontekstu, 194, 197, 206  
Object, 326  
obsługa aplikacji wielojęzycznych, 797  
obsługa atrybutów, 340  
obsługa błędów, 500  
obsługa ciasteczek, 261  
obsługa hiperłączy, 493  
obsługa JSP, 69  
obsługa komunikacji, 69  
obsługa obiektów zdalnych, 775

- obsługa pobierania pliku JAR, 157
- obsługa sesji, 264
- obsługa wielowątkowości, 69
- obsługa wyjątków, 797
- obsługa znaczników, 527
- obsługa żądania, 121, 133, 324
- obsługa żądania HTTP, 70
- ochrona
  - dane żądania, 713
  - informacje niezbędne do zalogowania, 716
  - zmienne egzemplarzy, 229
- odczytywanie
  - atrybuty, 341
  - atrybuty sesji, 341
  - atrybuty strony, 341
  - ciasteczka klienta, 418
  - ciasteczka z żądania klienta, 279
  - deskryptor TLD, 504, 506
  - identyfikator sesji, 261
  - właściwości, 375
  - znacznik kontekstu klienta, 281
- odnajdywanie atrybutów, 341
- odpowiedzi HTTP, 38, 39, 45, 63
  - typ MIME, 45
- odpowiedź serwera, 32
- odwołania do komponentu EJB, 658
- odwołania do komponentu lokalnego, 658
- odwołania do komponentu zdalnego, 658
- odwzorowania między rozszerzeniem
  - a typem MIME, 661
- odwzorowania serwetów, 75, 644
  - reguły, 647
- odwzorowanie adresu URL na serwlety, 73, 74
  - deskryptor wdrożenia, 76
- odwzorowanie adresu URL na treść, 51
- odwzorowanie nazw logicznych w pliki
  - klas serwetów, 106
- ograniczenia dostępu, 700
- ograniczenia parametrów inicjalizacji, 184
- ograniczenia parametrów kontekstu, 192
- określanie kolejności filtrów, 738

- opakowania, 747
  - dodawanie, 748
  - kompresja odpowiedzi, 752
  - strumień wyjściowy, 749
- OpcjaMenu, 601
- opcjonalny łańcuch zapytania, 48
- operacje wejścia-wyjścia, 154, 155
- operatory EL, 424
  - [], 399
  - kropka, 398
- oprogramowanie obsługi aplikacji internetowej, 768
- OPTIONS, 41, 137
- out, 326
- out.print(), 317
- OutputStream, 160

## P

- page, 315, 318, 326, 342
  - autoFlush, 343
  - buffer, 343
  - contentType, 343
  - errorPage, 343
  - extends, 343
  - import, 343
  - info, 343
  - isELIgnored, 343, 350
  - isErrorPage, 343
  - isThreadSafe, 343
  - language, 343
  - pageEncoding, 343
  - session, 343
- pageContext, 326, 413, 416
- PageContext, 326, 340, 605
- pageEncoding, 343
- pageScope, 413
- pakiety, 315
- param, 388, 413, 414
- parametry, 214, 531
  - aplikacja internetowa, 132
  - formularz, 148

- parametry inicjalizacji, 188, 418
  - kontekst, 185, 186
  - ograniczenia, 184
  - serwlet, 178, 179, 180, 186
- parametry żądania, 390
  - język EL, 414
- paramValues, 413, 414
- pętle, 474
- PKC, 705
- plain, old Java objects, 783
- platforma J2EE, 93
- pliki
  - deskryptor biblioteki znaczników, 420, 421
  - JAR, 156, 664
  - powitalne, 650
  - TLD, 664
  - WAR, 639, 640
- pliki znaczników, 527, 530, 538, 664
  - atrybuty, 532
  - attribute, 534
  - położenie, 537
- PlugIn, 797
- pobieranie
  - dane o użytkowniku, 774
  - plik JAR, 157
  - Tomcat, 22
- podglądanie, 679
- podsluchiwanie, 680
- podszywanie się, 679
- podział odpowiedzialności, 80
- POJO, 783
- połączenia bezstanowe, 258
- położenie plików znaczników, 537
- położenie zasobów, 632
- ponowne wdrażanie w czasie pracy, 181
- POP3, 49
- port TCP, 48, 49
- porzucone sesje, 270
- POST, 40, 44, 47, 63, 80, 127, 136, 137, 138
  - anatomia żądania, 44
  - ciało komunikatu, 44
  - dane użytkownika, 41
  - ładunek, 44
  - powtarzalność, 142
- poufna nazwa wewnętrzna, 74
- poufność, 681, 683, 708, 712
- powtarzalność danych, 139
- prezentacja, 80
- println(), 60, 62, 160, 177
- PrintWriter, 160, 177, 326
- proces budowy aplikacji, 102
- proces wytwarzania oprogramowania, 82
- program CGI, 55, 56
- programowe wzorce projektowe, 769
- programowe zapewnianie bezpieczeństwa, 702
- projektowanie kontrolera, 794
- projektowanie obiektowe, 772
- prosta klasa obsługi znacznika, 541
- protokoły
  - HTTP, 34, 38, 63, 153
  - HTTPS, 258, 710
- przechowywanie stanu konwersacyjnego, 255
- przechwytywanie żądań, 729
- przeglądarka internetowa, 32, 33, 37
- przeglądarka klienta, 150
- przekazywanie informacji
  - do plików znaczników, 532
  - do znacznika zewnętrznego, 600
- przekazywanie żądania
  - do obiektów z puli, 229
  - do strony JSP, 116
  - do strony JSP bez pośrednictwa serwletu, 387
- przekierowanie serwletu, 164
- przekierowanie żądania, 162, 167
- przekształcanie obiektu wyjątku w atrybut strony, 501
- przepisywanie adresów URL, 265, 266, 492, 493
  - sendRedirect(), 267
- przesyłanie ciasteczka klienta
  - z identyfikatorem sesji, 261
- przesyłanie
  - dane, 710
  - dane przeznaczone do przetworzenia, 139

- identyfikator sesji, 278
- parametry, 531
- przetwarzanie atrybutów kontekstu, 222
- przydzielanie żądania, 166, 167, 233
- przyznawanie wyższych uprawnień, 679
- Public Key Certificate, 705
- publiczna nazwa URL, 74, 76
- pula obiektów serwletu, 230
- pula wątków, 124
- PUT, 41, 137, 142

## R

- readObject(), 288
- referencje polimorficzne, 382
- rejestrwanie zdarzeń, 191
- Remote Method Invocation, 775
- removeAttribute(), 190, 217
- request, 326
- RequestDispatcher, 183, 233, 234
  - uzyskiwanie obiektu, 234
- requestScope, 413, 416, 417
- response, 326
- RFC 2616, 38
- RMI, 775, 776
  - etapy stosowania, 777
  - klient, 777
  - kontroler, 778
  - serwer, 777
- rodzaj zawartości, 158
- role, 689
- rozbudowa możliwości całej aplikacji internetowej, 731
- rozpoznawanie klienta, 258
- rozszerzalność, 770
- rozszerzenia plików, 159
- równoważenie obciążeń, 285
- rzeczywista struktura katalogów, 646

## S

- scope, 385, 485
- scriptlet, 312
- SCWCD, 24

- SelectTagHandler, 571, 574, 579, 582, 584
- sendError(), 655
- sendRedirect(), 164, 168, 267
- Serializable, 200, 288
- serializacja, 200, 288
- Service Locator, 782, 783, 788, 789, 807
- service(), 71, 123, 127, 128, 745
- Servlet, 108, 126, 152
- ServletConfig, 132, 133, 179, 182, 187, 188, 285, 311, 326
- ServletContext, 132, 133, 179, 182, 185, 187, 188, 190,
  - 201, 203, 213, 217, 285, 311, 326
- ServletContextAttributeEvent, 210
- ServletContextAttributeListener, 210
- ServletContextEvent, 194, 206, 207, 210
- ServletContextListener, 194, 196, 197, 198, 199, 203,
  - 208, 210
- ServletOutputStream, 160
- ServletRequest, 108, 134, 150, 153, 217, 745
- ServletRequestAttributeEvent, 210
- ServletRequestAttributeListener, 210
- ServletRequestEvent, 210
- ServletRequestListener, 210
- ServletRequestWrapper, 747
- ServletResponse, 108, 134
- ServletResponseWrapper, 747
- serwer, 32, 48
  - Apache, 38
  - J2EE, 93
  - Tomcat, 93
  - WWW, 31, 32, 53, 87
- serwlety, 29, 56, 58, 62, 65, 87, 121, 324, 734
  - bezpieczeństwo, 681
  - cykl życia, 124, 127
  - inicjalizacja, 125, 127, 130
  - interfejs API, 126
  - konfiguracja, 132
  - kontener, 67, 123
  - kontroler, 108
  - metody HTTP, 136
  - nazwy, 74
  - obiekt odpowiedzi, 124

### serwlety

- obsługa żądania, 133
  - odpowiedź, 134
  - odzworowania, 644
  - odzworowanie nazwy, 75
  - stan zainicjalizowany, 131
  - strona JSP, 311
  - wątek, 123, 124, 128
  - wczytywanie, 130
  - wdrażanie, 76
  - zainicjalizowany, 125
  - zwalnianie zasobów, 125
  - żądanie, 129, 134
- sesja, 150, 251, 256, 493
- aktywacja, 283
  - atrybuty, 282, 283, 284
  - ciasteczka, 260, 261
  - cykl życia, 282, 283
  - czas pierwszego utworzenia, 272
  - dezaktywacja, 282, 283
  - dostęp do istniejącej sesji, 263
  - getSession(), 271
  - HttpSession, 271
  - HttpSessionActivationListener, 283, 288, 293
  - HttpSessionAttributeListener, 283, 293
  - HttpSessionBindingEvent, 283, 293
  - HttpSessionBindingListener, 284, 293
  - HttpSessionEvent, 283, 293
  - HttpSessionListener, 283, 293
  - identyfikator, 259, 260
  - interfejsy nasłuchujące zdarzeń, 292
  - klasa atrybutu, 291
  - klasa nasłuchująca dla atrybutów, 290
  - kończenie, 272
  - licznik, 289
  - migracja, 285, 286
  - niszczenie, 282
  - obsługa, 264
  - porzucona, 270
  - przekroczenie limitu czasowego, 282
  - przeniesienie, 283
  - przepisywanie adresów URL, 265
  - przesyłanie identyfikatorów, 278
  - równoważenie obciążeń, 285
  - serializacja, 288
  - sprawdzanie istnienia, 262
  - tworzenie, 282
  - ustawianie limitu czasowego, 273
  - usuwanie, 269
  - wyłączona obsługa ciasteczek, 264
  - zdarzenia, 283
- session, 318, 326, 343
- sessionCreated(), 262, 289, 293
- sessionDestroyed(), 289, 293
- sessionDidActivate(), 291, 293
- sessionScope, 413
- sessionWillPassivate(), 291, 293
- sesyjne komponenty EJB, 255
- setAttribute(), 190, 214, 217, 221, 233, 339, 341, 474
- setBodyContent(), 590, 592, 594
- setClass(), 579
- setContentType(), 112, 154, 157, 158, 161, 168
- Set-Cookie, 264, 266
- setCookie(), 281
- setDir(), 580
- setHeader(), 161, 168, 281
- setId(), 579
- setInitParameter(), 188
- setIntHeader(), 161
- setJspBody(), 544
- setJspContext(), 544
- setLang(), 580
- setMaxAge(), 279
- setMaxInactiveInterval(), 272, 273
- setMultiple(), 581
- setName(), 581
- setOnblur(), 582
- setOnchange(), 582
- setOnClick(), 580
- setOndbclick(), 580, 581

- setOnfocus(), 582
- setOnkeydown(), 581
- setOnkeypress(), 581
- setOnkeyup(), 581
- setOnmousedown(), 580
- setOnmousemove(), 581
- setOnmouseout(), 581
- setOnmouseover(), 580
- setOnmouseup(), 580
- setOptionsList(), 579
- setPageContext(), 563
- setParent(), 563
- setParent(), 544
- setSize(), 581
- setStyle(), 580
- setTabIndex(), 582
- setTitle(), 580
- sieć WWW, 31
- Simple Tags, 527
- SimpleTag, 543, 544, 555, 596
- SimpleTagSupport, 507, 541, 543, 556, 558, 571, 584
- SingleThreadModel, 229, 231
- SKIP\_BODY, 564, 591, 592
- SKIP\_PAGE, 564
- SkipPageException, 551, 552, 554, 560
- składnia
  - dokumenty JSP, 657
  - strony JSP, 657
- składowe, 322
- skryptlety, 312, 316, 322, 324, 344
  - bezpieczeństwo, 345
  - zmienne, 320
- skrypty, 371
  - CGI, 55
  - program ANT, 632
- SMTP, 49
- spójność, 771
- sprawdzanie
  - istnienie sesji, 262
  - metoda żądania HTTP, 150
- sprzęt, 767
- SSL, 710, 716
- stan konwersacyjny, 251, 255
  - HttpSession, 255
- standardowa biblioteka znaczników JSP, 469
- statyczne strony WWW, 52
- STM, 229, 230
- stos TCP/IP, 38
- stosowanie
  - model zdalny, 778
  - wzorzec projektowy MVC, 83
  - zdalny obiekt, 778
- String, 192
- strona bezskryptowa, 371
- strona dynamiczna, 54
- strona HTML, 32
- strona JSP, 29, 30, 61, 80, 82, 309
  - atributy, 339
  - biblioteki znaczników, 342
  - cykl życia, 334
  - deklaracje, 322
  - dostęp do parametrów inicjalizacji serwletu, 183
  - dyrektywy, 316
  - import pakietów, 315
  - inicjalizacja, 338
  - jspInit(), 338
  - kompilacja, 336
  - kontener, 324
  - obsługa żądań, 116
  - serwlety, 311
  - skryptlety, 316
  - testowanie, 313
  - tłumaczenie, 336
  - tworzenie, 312
  - wdrażanie, 313
  - wyrażenia, 316
- strona just-in-time, 54
- strona otwierająca, 104
- strona statyczna, 52
- strona WWW, 32

- strony o błędach, 496, 654
  - <error-code>, 498
  - <error-page>, 497
- dla konkretnych wyjątków, 498
- exception, 499
- kod stanu protokołu HTTP, 498
- konfiguracja, 498, 654
- struktura aplikacji, 635
- struktura katalogów, 100, 645
  - witryna internetowa, 50
- strumień wejściowy z danego żądania, 150
- Struts, 795
  - ActionServlet, 801
  - aplikacje, 798
  - architektura aplikacji, 799
  - automatyczne przekazywanie żądań, 797
  - deklaratywna weryfikacja poprawności, 797
  - deskryptor wdrożenia, 795, 802
  - Front Controller, 797
  - globalna obsługa wyjątków, 797
  - instalacja, 804
  - komponent formularza, 800
  - komponenty, 795
  - kontener, 796
  - kontrola deklaratywna, 797
  - obiekt akcji, 801
  - obsługa aplikacji wielojęzycznych, 797
  - pluginy, 797
  - struts-config.xml, 795, 799, 802
  - web.xml, 803
  - wtyczki, 797
  - znaczniki niestandardowe, 797
  - źródła danych, 797
- struts-config.xml, 795, 799, 802
- Swing, 29
- synchronizacja dostępu do sesji, 228
- synchronizacja metody obsługującej żądania, 223, 224
- synchronized, 223, 225
- system obiektowy, 772
- szablony układu, 430

## Ś

- ścieżka, 48
  - względna, 234
- śledzenie
  - odpowiedzi klienta, 255
  - żądania, 735
- środowisko wdrożeniowe, 97, 805
  - tworzenie, 101
- środowisko wytwarzania aplikacji, 95, 97
  - tworzenie, 100

## T

- tabele HTML, 475
- tablice, 402
- tag, 536
  - body-content, 536
- Tag, 558, 565, 590, 596
- Tag Files, 527
- tag handler, 347
- taglib, 342, 420, 422, 512, 515, 530
  - <uri>, 512
- TagSupport, 558, 565, 567, 590
- TCP, 49, 51
- TCP/IP, 38
- technika dzierżawy, 269
- technika łączenia w klastry, 129
- technologia JSP, 62, 309
- technologia serwletów, 30
- testowanie, 97
  - aplikacje internetowe, 113, 118
  - klasa modelu, 110
  - model, 110
  - serwlet kontrolera, 109
  - strona otwierająca, 104
- text/html, 45
- Throwable, 326
- TLD, 420, 421, 504, 533, 538
- Tomcat, 22, 50, 68, 93, 105
- TOMCAT HOME, 22
- tomcat-users.xml, 692

TRACE, 41, 137  
 Transfer Object, 770, 787, 789, 808  
 tworzenie  
   deskryptor wdrożenia, 58  
   filtry, 732  
   klasa atrybutu, 200  
   klasa nasłuchująca, 199  
   klasa serwletu, 201  
   komponenty JavaBean, 378  
   pliki znacznika, 530  
   prosta klasa obsługi znacznika, 541  
   serwlety, 58  
   strona JSP, 312  
   środowisko wdrożeniowe, 101, 805  
   środowisko wytwarzania aplikacji, 100  
 typ MIME, 45, 158, 343  
 type, 383, 385  
 typy użytkowników, 689

## **U**

udostępnianie  
   statyczne strony internetowe, 52  
   strony JSP, 642  
 ukrywanie wyszukiwania JNDI, 780  
 ukrywanie złożoności, 771  
 umieszczanie zasobów, 632  
 UML, 23  
 Uniform Resource Locator, 48  
 uri, 513  
 URI, 48  
 URL, 48, 63  
   budowa, 48  
 URN, 48  
 useBean, 381  
 ustawianie  
   atrybuty, 341  
   atrybuty sesji, 341  
   atrybuty strony, 341  
   limit czasowy sesji, 273  
   nagłówek odpowiedzi, 161

  rodzaj zawartości, 159  
   zmiennne atrybutu, 483  
 usuwanie sesji, 269  
 uwierzytelnianie, 258, 681, 683, 686, 691, 705, 708  
   BASIC, 705  
   CLIENT-CERT, 705  
   deklaratywne, 706  
   DIGEST, 705  
   FORM, 705, 707  
   HTTP, 684  
   implementacja, 706  
   kontener, 686  
   włączanie, 691  
 uzyskiwanie obiektu RequestDispatcher, 234  
 użytkownicy, 689

## **V**

valueBound(), 211, 291, 293  
 valueUnbound(), 211, 291, 293  
 var, 501  
 varStatus, 476, 478

## **W**

w pełni kwalifikowana nazwa klasy, 76  
 WAR, 639, 640, 641  
 warstwa biznesowa, 767  
 warstwa internetowa, 767  
 warstwa prezentacji, 767  
 wartości null, 427, 473  
 warunkowe dołączanie kodu, 479  
 warunkowe przekazywanie żądań, 442  
 wątki, 128  
   atrybuty kontekstu, 222  
   atrybuty sesji, 226  
   atrybuty żądania, 232  
   synchronizacja dostępu do sesji, 228  
   synchronizacja metody obsługującej żądania, 223  
   zasięg kontekstu, 220  
   żądania, 129  
 wczytywanie serwletu, 130

### wdrażanie

- aplikacja, 204
- aplikacja wykorzystujące funkcje, 422
- serwlet, 76
- serwlet kontrolera, 109
- strona JSP, 115
- strona otwierające, 104

wdrażanie aplikacji internetowych, 113, 118, 122, 629

- dokumenty JSP, 657
- konfiguracja ładowania w trakcie uruchamiania aplikacji, 656
- lokalizacje dostępne bezpośrednio, 642
- MVC, 95
- odwołania do komponentów EJB, 658
- odzworowania między rozszerzeniem a typem MIME, 661
- odzworowania serwletów, 644
- plik WAR, 639, 640
- pliki powitalne, 650
- strony błędów, 654
- zasoby, 631, 632

web.xml, 58, 92, 101, 105, 177, 632, 636

webapps, 157, 641, 691

WEB-INF, 59, 101, 157, 514, 636, 643

WEB-INF/classes, 101, 197, 643

WEB-INF/lib, 468, 514

WEB-INF/tags, 537, 632

WebLogic, 93

widok, 82, 97, 98, 373, 774, 794

wielowątkowość, 69

wirtualna maszyna Javy, 56

wirtualna struktura katalogów, 646

witryny WWW, 31

wizualizowanie strony internetowej, 33

właściwości obiektów, 394

właściwości żądania, 390

włączanie uwierzytelniania, 691

writeObject(), 288

wtyczki, 797

WWW, 31

wybór pliku powitalnego, 653

### wydajność, 770

### wyjątki, 499

- SkipPageException, 551

wyliczanie wyrażeń EL, 352

wyłączona obsługa ciasteczek, 264

wymiana informacji o identyfikatorze sesji, 260

wyodrębnianie problemów, 771

wyrażenia, 316, 317, 322, 324, 332

wyrażenia EL, 348, 371, 397

wysokopoziomowa architektura aplikacji internetowych, 66

wysyłanie

- bajty w obiekcie odpowiedzi, 156
- ciasteczka do klienta, 279

wyszukiwanie JNDI, 780

wyświetlanie właściwości, 395

wytwarzanie aplikacji, 95

wywołanie modelu z poziomu serwletu kontrolera, 112

wywołanie znacznika, 510

względny adres URL, 164

wzorce projektowe, 765, 769

- Business Delegate, 781, 788, 789, 806
- Front Controller, 797, 811
- Intercepting Filter, 809
- MVC, 81, 774, 790, 810
- Service Locator, 782, 788, 789, 807
- Transfer Object, 787, 789, 808

warstwa biznesowa, 789

wzorce URL, 698

## X

XHTML, 471

XML, 69, 95, 192, 657

XPath, 371

XSS, 472

## Z

### zabezpieczanie

- atributy kontekstu, 225
- przesyłane informacje, 710
- serwlety, 681, 690

- zagrożenia, 679
  - podśluchiwanie, 680
- zależności pomiędzy atrybutami a bezpieczeństwem wątków, 227
- zapisywanie danych na serwerze, 54
- zarządzanie cyklem życia, 69
- zarządzanie sesjami, 251
- zasady projektowania obiektowego, 772
- zasięg atrybutów, 216, 237
- zasięg kontekstu, 220
- zasoby, 32, 48, 631, 632
- zawartość dynamiczna, 54
- zdalne komponenty modelu, 773
- zdalne obiekty, 778
- zdalne obiekty pośredniczące, 772
- zdarzenia cyklu życia sesji, 283
- zdarzenia kontekstu, 197
- zintegrowane środowisko programistyczne, 22
- złożoności, 771
- zmienne, 323
  - klasowe, 231
  - lokalne, 232
  - skryptlet, 320
- znaczniki HTML, 35
  - atrybuty, 35
- znaczniki JSTL, 445
- znaczniki kontekstu klienta, 278, 281
- znaczniki niestandardowe, 467, 469, 506, 527
  - atrybuty, 578, 579
  - atrybuty dynamiczne, 584
  - atrybuty nieprawidłowe, 588
  - atrybuty plików znaczników, 532
  - atrybuty przekazywane, 580
  - BodyTag, 590, 591, 592
  - BodyTagSupport, 590, 592
  - buforowanie zawartości, 592
  - ciało, 510
  - cykl życia obiektu obsługi znacznika prostego, 544
  - cykl życia znacznika klasycznego, 564
  - deklaracja zawartości znacznika dla pliku znacznika, 536
  - deskryptor TLD, 506, 511
  - dodawanie atrybutów dynamicznych, 586
  - doEndTag(), 560
  - doStartTag(), 560
  - dostęp do zawartości znacznika, 590
  - duża zawartość atrybutu, 535
  - DynamicAttributes, 584, 585
  - dynamicznie generowana zawartość wierszy, 548
  - getParent(), 598, 599
  - identyfikator URI, 506
  - interfejs programowy klas obsługi znaczników, 558
  - interfejs programowy znaczników prostych, 543
  - iteracyjne przetwarzanie ciała znacznika, 548
  - IterationTag, 565
  - JspFragment, 550
  - JspTag, 558
  - klasa obsługi, 507, 511, 540, 557, 561
  - klasyczna klasa obsługi, 560
  - klasyczne, 556, 560, 563, 604
  - Menu, 601
  - modele klas obsługi, 561
  - obsługa atrybutów, 574, 580
  - odnajdywanie przodków, 602
  - odwołanie do znacznika zewnętrznego, 596, 602
  - OpcjaMenu, 601
  - PageContext, 605
  - pliki znaczników, 530, 538, 589
  - położenie plików znaczników, 537
  - poziom zagnieżdżenia znacznika, 597
  - prosta klasa obsługi, 541
  - proste, 556, 598, 604
  - przekazywanie informacji do znacznika
    - zewnętrznego, 600
  - przesyłanie parametrów, 531
  - przetwarzanie iteracyjne w znacznikach klasycznych, 562
  - referencja do znacznika zewnętrznego, 596
  - SelectTagHandler, 579
  - SimpleTag, 543, 555
  - SimpleTagSupport, 507, 556
  - SkipPageException, 551, 552, 554
  - strona JSP, 511
  - tag, 536

### znaczniki niestandardowe

Tag, 558

TagSupport, 565, 567

tworzenie, 533

wewnętrzne, 595

wielokrotne przetwarzanie zawartości znacznika, 565

współdziałanie klas obsługi, 598

wyrażenia EL, 547

wywoływanie, 510

znacznik prosty z atrybutem, 549

znacznik prosty z zawartością, 542

### znaczniki proste, 527

znaki specjalne HTML, 472

zwyczajne, tradycyjne obiekty Javy, 783

## Ż

żądania HTTP, 30, 38, 40, 63

GET, 40, 145

kontener, 70

obsługa, 70

POST, 40, 145, 147, 148

żądania idempotentne, 144

żądania klienta, 32

żądania nieidempotentne, 140



# PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION

- 
1. ZAREJESTRUJ SIĘ
  2. PREZENTUJ KSIĄŻKI
  3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW  
w działający bankomat!

**Dowiedz się więcej i dołącz już dzisiaj!**

<http://program-partnerski.helion.pl>

GRUPA WYDAWNICZA  
 **Helion SA**