

TypeScript 4

Od początkującego do profesjonalisty

Wydanie II

Adam Freeman

Tytuł oryginału: Essential TypeScript 4: From Beginner to Pro, 2nd Edition

Tłumaczenie: Robert Górczyński

ISBN: 978-83-283-8830-7

First published in English under the title Essential TypeScript 4: From Beginner to Pro by Adam Freeman, edition: 2

Copyright © 2021 by Adam Freeman

This edition has been translated and published under licence from APress Media, LLC, part of Springer Nature. APress Media, LLC, part of Springer Nature takes no responsibility and shall not be made liable for the accuracy of the translation.

Polish edition copyright © 2022 by Helion S.A.
All rights reserved.

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 wydawca 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 wydawca nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: <https://helion.pl> (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
<https://ftp.helion.pl/przyklady/tys4o2.zip>

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/tys4o2_ebook
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

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

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

*Książkę dedykuję mojej kochanej żonie Jacqui Griffyth
(a także Peanut)*

Spis treści

	O autorze	15
	O korektorze merytorycznym	16
Część I	Zaczynamy	17
Rozdział 1.	Pierwsza aplikacja w TypeScriptie	19
	Przygotowanie systemu	19
	Krok 1. Instalowanie Node.js	19
	Krok 2. Instalowanie Gita	20
	Krok 3. Instalowanie TypeScriptu	20
	Krok 4. Instalowanie programistycznego edytora tekstu	21
	Utworzenie projektu	22
	Inicjalizacja projektu	22
	Utworzenie pliku konfiguracyjnego kompilatora	22
	Tworzenie pliku kodu TypeScriptu	23
	Kompilowanie i uruchamianie kodu	23
	Definiowanie modelu danych	24
	Dodawanie funkcji do klasy kolekcji	29
	Używanie pakietu zewnętrznego	36
	Dodawanie deklaracji typu dla pakietu JavaScriptu	39
	Dodawanie poleceń	40
	Filtrowanie elementów	40
	Dodawanie zadań	42
	Oznaczanie zadania jako wykonanego	44
	Trwałe przechowywanie danych	47
	Stosowanie klasy trwałego magazynu danych	49
	Podsumowanie	50
Rozdział 2.	Poznajemy TypeScript	51
	Dlaczego powinieneś używać języka TypeScript?	52
	Funkcje języka TypeScript zwiększające produktywność programisty	52
	Poznajanie wersji JavaScriptu	53

Co powinienś wiedzieć?	54
Jak skonfigurować środowisko programistyczne?	54
Jaka jest struktura książki?	54
Czy w książce znajdziesz wiele przykładów?	55
Gdzie znajdziesz przykładowe fragmenty kodu?	57
Co zrobić w przypadku problemów podczas wykonywania przykładów?	57
Co zrobić w sytuacji, gdy znajdę błąd w książce?	57
Jak mogę skontaktować się z autorem?	58
Co zrobić, jeśli książka mi się podobała?	58
Co zrobić, jeśli książka mi się nie podobała?	58
Podsumowanie	59
Rozdział 3. Wprowadzenie do języka JavaScript — część I	60
Przygotowanie projektu	60
Zagmatwany JavaScript	61
Typy języka JavaScript	63
Praca z podstawowymi typami danych	63
Koercja typu	66
Praca z funkcją	69
Praca z tablicą	75
Używanie operatora rozwinięcia w tablicy	77
Destrukturyzacja tablicy	78
Praca z obiektem	79
Dodawanie, modyfikowanie i usuwanie właściwości obiektu	80
Używanie operatorów rozwinięcia i resztowego w obiekcie	83
Definiowanie funkcji typu getter i setter	85
Definiowanie metod	86
Słowo kluczowe this	87
Słowo kluczowe this w oddzielnych funkcjach	89
Słowo kluczowe this w metodach	90
Zmiana zachowania słowa kluczowego this	91
Słowo kluczowe this w funkcji strzałki	92
Powrót do problemu początkowego	93
Podsumowanie	95
Rozdział 4. Wprowadzenie do języka JavaScript — część II	96
Przygotowanie projektu	96
Dziedziczenie obiektu JavaScriptu	97
Analizowanie i modyfikowanie prototypu obiektu	98
Tworzenie własnych właściwości	100
Używanie funkcji konstruktora	101
Sprawdzanie typu prototypu	104
Definiowanie statycznych właściwości i metod	105
Używanie klas JavaScriptu	106
Używanie iteratorów i generatorów	109
Używanie generatora	110
Definiowanie obiektów pozwalających na iterację	112

Używanie kolekcji JavaScriptu	114
Sortowanie danych według klucza przy użyciu obiektu	114
Sortowanie danych według klucza przy użyciu obiektu Map	116
Przechowywanie danych według indeksu	118
Używanie modułów	119
Tworzenie modułu JavaScriptu	119
Używanie modułu JavaScriptu	120
Eksportowanie funkcji z modułu	121
Definiowanie w modelu wielu funkcjonalności nazwanych	123
Podsumowanie	124
Rozdział 5. Używanie kompilatora TypeScript	125
Przygotowanie projektu	125
Struktura projektu	127
Używanie menedżera pakietów Node	128
Plik konfiguracyjny kompilatora TypeScript	131
Kompilacja kodu TypeScript	133
Błędy generowane przez kompilator	134
Używanie trybu monitorowania i wykonywania skompilowanego kodu	135
Używanie funkcjonalności wersjonowania celu	138
Wybór plików biblioteki do kompilacji	140
Wybór formatu modułu	143
Użyteczne ustawienia konfiguracji kompilatora	147
Podsumowanie	149
Rozdział 6. Testowanie i debugowanie kodu TypeScript	150
Przygotowanie projektu	150
Debugowanie kodu TypeScript	151
Przygotowanie do debugowania	151
Używanie Visual Studio Code do debugowania	152
Używanie zintegrowanego debuggera Node.js	154
Używanie funkcji zdalnego debugowania w Node.js	155
Używanie lintera TypeScript	157
Wyłączanie reguł lintowania	159
Testy jednostkowe w TypeScript	161
Konfigurowanie frameworka testów	162
Tworzenie testów jednostkowych	163
Uruchamianie frameworka testów	164
Podsumowanie	166
Część II Praca z językiem TypeScript	167
Rozdział 7. Typowanie statyczne	169
Przygotowanie projektu	170
Typy statyczne	172
Tworzenie typu statycznego za pomocą adnotacji typu	174
Używanie niejawnie zdefiniowanego typu statycznego	175
Używanie typu any	178

Używanie unii typów	181
Używanie asercji typu	183
Asercja typu nieoczekiwanego	184
Używanie wartownika typu	186
Używanie typu never	187
Używanie typu unknown	188
Używanie typów null	189
Ograniczenie przypisywania wartości null	190
Usunięcie null z unii za pomocą asercji	192
Usuwanie wartości null z unii za pomocą wartownika typu	193
Używanie asercji ostatecznego przypisania	194
Podsumowanie	196
Rozdział 8. Używanie funkcji	197
Przygotowanie projektu	198
Definiowanie funkcji	199
Ponowne definiowanie funkcji	199
Parametry funkcji	201
Wynik działania funkcji	208
Przeciążanie typu funkcji	210
Funkcje asercji	212
Podsumowanie	214
Rozdział 9. Tablice, krotki i wyliczenia	215
Przygotowanie projektu	216
Praca z tablicami	217
Używanie automatycznie ustalonego typu tablicy	219
Unikanie problemów z automatycznie ustalonym typem tablicy	220
Unikanie problemów z pustą tablicą	221
Krotka	222
Przetwarzanie krotki	224
Używanie typów krotki	225
Używanie krotki z elementami opcjonalnymi	226
Definiowanie krotki z elementem resztowym	227
Wyliczenie	228
Sposób działania wyliczenia	229
Używanie wyliczenia w postaci ciągu tekstowego	232
Ograniczenia typu wyliczeniowego	233
Używanie typu literału wartości	236
Używanie w funkcji typu literałów wartości	237
Łączenie typów wartości w typie literałów wartości	237
Nadpisywanie za pomocą typu literałów wartości	238
Używanie szablonów typu literałów tekstowych	240
Używanie aliasu typu	241
Podsumowanie	242

Rozdział 10. Praca z obiektami	243
Przygotowanie projektu	244
Praca z obiektami	245
Używanie adnotacji kształtu typu obiektu	246
Dopasowanie kształtu typu obiektu	247
Wymuszenie ścisłego sprawdzania metod	250
Używanie aliasu typu dla kształtu typu	251
Radzenie sobie z nadmiarem właściwości	251
Używanie unii kształtu typu	253
Typy właściwości unii	254
Używanie wartownika typu dla obiektu	255
Używanie złączenia typów	259
Używanie złączenia do korelacji danych	261
Łączenie złączy	262
Podsumowanie	269
Rozdział 11. Praca z klasami i interfejsami	270
Przygotowanie projektu	271
Używanie funkcji konstruktora	272
Używanie klas	275
Używanie słów kluczowych kontroli dostępu	276
Używanie właściwości prywatnych JavaScriptu	279
Definiowanie właściwości tylko do odczytu	282
Upraszczanie klasy konstruktora	283
Używanie dziedziczenia klas	284
Używanie klasy abstrakcyjnej	287
Używanie interfejsu	290
Implementowanie wielu interfejsów	292
Rozszerzanie interfejsu	294
Definiowanie opcjonalnych właściwości i metod interfejsu	296
Definiowanie implementacji interfejsu abstrakcyjnego	298
Wartownik typu interfejsu	299
Dynamiczne tworzenie właściwości	300
Sprawdzanie wartości indeksu	301
Podsumowanie	304
Rozdział 12. Używanie typów generycznych	305
Przygotowanie projektu	306
Zrozumienie problemu	308
Dodawanie obsługi innego typu	309
Tworzenie klasy generycznej	310
Argumenty typu generycznego	311
Używanie argumentów innego typu	312
Ograniczanie wartości typu generycznego	313
Definiowanie parametrów wielu typów	316
Pozostawienie kompilatorowi zadania ustalenia typu argumentu	318
Rozszerzanie klasy generycznej	320

Wartownik typu generycznego	324
Definiowanie metody statycznej w klasie generycznej	326
Definiowanie interfejsu generycznego	328
Rozszerzanie interfejsu generycznego	328
Implementacja interfejsu generycznego	329
Podsumowanie	333
Rozdział 13. Zaawansowane typy generyczne	334
Przygotowanie projektu	334
Używanie kolekcji generycznych	336
Używanie iteratorów generycznych	338
Łączenie iteratora i obiektu możliwego do iteracji	340
Tworzenie klasy umożliwiającej iterację	341
Używanie typów indeksu	342
Używanie zapytania typu indeksu	342
Jawne dostarczanie parametrów typu generycznego dla typów indeksu	343
Używanie zindeksowanego operatora dostępu	344
Używanie typu indeksu dla klasy Collection<T>	346
Używanie mapowania typu	348
Zmiana mapowanych nazw i typów	349
Używanie parametru typu generycznego z typem mapowanym	350
Zmiana modyfikowalności i opcjonalności właściwości	351
Używanie wbudowanych typów mapowanych	352
Łączenie transformacji w pojedyncze mapowanie	354
Tworzenie typu z użyciem mapowania	355
Używanie typów warunkowych	356
Zagnieżdżanie typów warunkowych	357
Używanie typu warunkowego w klasie generycznej	358
Używanie typów warunkowych z uniami typów	359
Używanie typów warunkowych podczas mapowania typów	361
Identyfikowanie właściwości określonego typu	361
Automatyczne ustalanie typów dodatkowych w warunkach	363
Podsumowanie	366
Rozdział 14. Praca z JavaScriptem	367
Przygotowanie projektu	368
Dodawanie kodu TypeScriptu do przykładowego projektu	369
Praca z JavaScriptem	372
Dołączanie kodu JavaScriptu w trakcie kompilacji	373
Sprawdzanie typu kodu JavaScriptu	374
Opisywanie typów używanych w kodzie JavaScriptu	376
Używanie komentarzy do opisywania typów	377
Używanie plików deklaracji typu	379
Opisywanie kodu JavaScriptu przygotowanego przez podmioty zewnętrzne	381
Używanie plików deklaracji pochodzących z projektu Definitely Typed	384
Używanie pakietów zawierających deklaracje typu	386
Generowanie plików deklaracji	389
Podsumowanie	391

Część III	Tworzenie aplikacji internetowych	393
Rozdział 15.	Tworzenie aplikacji internetowej TypeScriptu — część I	395
	Przygotowanie projektu	396
	Przygotowanie zestawu narzędzi	397
	Dodawanie obsługi paczek	397
	Dodawanie programistycznego serwera WWW	400
	Tworzenie modelu danych	404
	Tworzenie źródła danych	405
	Generowanie treści HTML-a za pomocą API modelu DOM	408
	Dodawanie obsługi stylów Bootstrap CSS	409
	Używanie formatu JSX do tworzenia treści HTML-a	411
	Sposób działania JSX	413
	Konfigurowanie kompilatora TypeScriptu	
	i procedury wczytującej pakiet webpack	414
	Tworzenie funkcji fabryki	415
	Używanie klasy JSX	416
	Importowanie funkcji fabryki w klasie JSX	417
	Dodawanie funkcjonalności do aplikacji	418
	Wyświetlanie filtrowanej listy produktów	418
	Wyświetlanie treści i obsługa uaktualnień	422
	Podsumowanie	424
Rozdział 16.	Tworzenie aplikacji internetowej TypeScriptu — część II	425
	Przygotowanie projektu	426
	Dodawanie usługi sieciowej	428
	Wykorzystanie źródła danych w aplikacji	429
	Używanie dekoratorów	431
	Używanie metadanych dekoratora	433
	Dokończenie aplikacji	436
	Dodawanie klasy Header	437
	Dodawanie klasy obsługującej szczegóły zamówienia	437
	Dodawanie klasy obsługującej potwierdzenie zamówienia	439
	Zakończenie pracy nad aplikacją	439
	Wdrażanie aplikacji	442
	Dodawanie pakietu produkcyjnego serwera HTTP	443
	Tworzenie pliku dla trwałego magazynu danych	443
	Utworzenie serwera	444
	Używanie względnych adresów URL do obsługi żądań danych	444
	Kompilacja aplikacji	445
	Testowanie gotowej aplikacji	446
	Umieszczanie aplikacji w kontenerze	447
	Instalowanie Dockera	447
	Przygotowanie aplikacji	448
	Tworzenie kontenera Dockera	448
	Uruchamianie aplikacji	449
	Podsumowanie	451

Rozdział 17. Tworzenie aplikacji internetowej Angulara — część I	452
Przygotowanie projektu	453
Konfigurowanie usługi sieciowej	453
Konfigurowanie pakietu Bootstrap CSS	455
Uruchomienie przykładowej aplikacji	456
Rola TypeScriptu w programowaniu z użyciem frameworka Angular	457
Rola TypeScriptu w łańcuchu narzędzi Angulara	458
Tworzenie modelu danych	459
Tworzenie źródła danych	460
Tworzenie implementacji klasy źródła danych	462
Konfigurowanie źródła danych	464
Wyświetlenie filtrowanej listy produktów	465
Wyświetlanie przycisków kategorii	467
Utworzenie nagłówka	468
Połączenie komponentów produktu, kategorii i nagłówka	469
Konfigurowanie aplikacji	470
Podsumowanie	472
Rozdział 18. Tworzenie aplikacji internetowej Angulara — część II	473
Przygotowanie projektu	474
Dokończenie pracy nad funkcjonalnością aplikacji	474
Dodawanie komponentu obsługującego podsumowanie zamówienia	477
Tworzenie konfiguracji routingu	478
Wdrażanie aplikacji	480
Dodawanie pakietu produkcyjnego serwera HTTP	480
Tworzenie pliku dla trwałego magazynu danych	481
Tworzenie serwera	481
Używanie względnych adresów URL do obsługi żądań danych	482
Kompilowanie aplikacji	483
Testowanie gotowej aplikacji	484
Umieszczanie aplikacji w kontenerze	484
Przygotowanie aplikacji	485
Tworzenie kontenera Dockera	485
Uruchamianie aplikacji	486
Podsumowanie	487
Rozdział 19. Tworzenie aplikacji internetowej React — część I	488
Przygotowanie projektu	489
Konfigurowanie usługi sieciowej	490
Instalowanie pakietu Bootstrap CSS	491
Uruchamianie przykładowej aplikacji	491
TypeScript i programowanie z użyciem frameworka React	492
Definiowanie typów encji	495
Wyświetlanie filtrowanej listy produktów	496
Używanie zaczepów i komponentów funkcyjnych	499
Wyświetlanie listy kategorii i nagłówka	500
Przygotowanie i przetestowanie komponentów	501

Tworzenie magazynu danych	504
Tworzenie klasy żądania HTTP	508
Łączenie komponentów z magazynem danych	509
Podsumowanie	511
Rozdział 20. Tworzenie aplikacji internetowej React — część II	512
Przygotowanie projektu	513
Konfigurowanie routingu URL	513
Dokończenie pracy nad funkcjonalnością aplikacji	515
Dodawanie komponentu obsługującego podsumowanie zamówienia	517
Dodawanie komponentu potwierdzającego złożenie zamówienia	518
Dokończenie konfiguracji routingu	519
Wdrażanie aplikacji	520
Dodawanie pakietu produkcyjnego serwera HTTP	521
Tworzenie pliku dla trwałego magazynu danych	521
Tworzenie serwera	522
Używanie względnych adresów URL do obsługi żądań danych	522
Kompilowanie aplikacji	523
Testowanie gotowej aplikacji	524
Umieszczanie aplikacji w kontenerze	525
Przygotowanie aplikacji	525
Tworzenie kontenera Dockera	525
Uruchamianie aplikacji	526
Podsumowanie	528
Rozdział 21. Tworzenie aplikacji internetowej Vue.js — część I	529
Przygotowanie projektu	530
Konfigurowanie usługi sieciowej	531
Instalowanie pakietu Bootstrap CSS	532
Uruchamianie przykładowej aplikacji	533
TypeScript i programowanie w Vue.js	533
Zestaw narzędzi TypeScriptu podczas programowania z użyciem frameworka Vue.js	535
Tworzenie klas encji	536
Wyświetlanie filtrowanej listy produktów	537
Wyświetlanie listy kategorii i nagłówka	540
Tworzenie i testowanie komponentów	542
Tworzenie magazynu danych	545
Łączenie komponentów z magazynem danych	547
Dodawanie obsługi usługi sieciowej	550
Podsumowanie	553
Rozdział 22. Tworzenie aplikacji internetowej Vue.js — część II	554
Przygotowanie projektu	555
Konfigurowanie routingu URL	555
Dokończenie pracy nad funkcjonalnością aplikacji	558
Dodawanie komponentu obsługującego podsumowanie zamówienia	558
Dodawanie komponentu potwierdzającego złożenie zamówienia	560
Dokończenie konfiguracji routingu	561

Wdrażanie aplikacji	561
Dodawanie pakietu produkcyjnego serwera HTTP	562
Tworzenie pliku dla trwałego magazynu danych	562
Tworzenie serwera	563
Używanie względnych adresów URL do obsługi żądań danych	563
Kompilowanie aplikacji	564
Testowanie gotowej aplikacji	565
Umieszczanie aplikacji w kontenerze	565
Przygotowanie aplikacji	565
Tworzenie kontenera Dockera	566
Uruchamianie aplikacji	567
Podsumowanie	568

0 autorze



Adam Freeman jest doświadczonym specjalistą IT, który zajmował kierownicze stanowiska w wielu firmach, a ostatnio pracował jako dyrektor ds. technologii oraz dyrektor naczelny w międzynarodowym banku. Obecnie przeszedł na emeryturę i poświęca swój czas na pisanie oraz bieganie.

O korektorze merytorycznym

Fabio Claudio Ferracchiati jest starszym konsultantem oraz starszym analitykiem-programistą korzystającym z technologii firmy Microsoft. Pracuje w firmie BluArancio (<http://www.bluarancio.com/>). Posiada certyfikaty Microsoft Certified Solution Developer for .NET, Microsoft Certified Application Developer for .NET, Microsoft Certified Professional. Jest autorem, współautorem i recenzentem technicznym wielu książek o różnej tematyce. W ciągu ostatnich dziesięciu lat pisał artykuły dla włoskich i międzynarodowych czasopism, a także był współautorem kilkunastu książek poświęconych różnym dziedzinom informatyki.

CZĘŚĆ I



Zaczynamy

ROZDZIAŁ 1.



Pierwsza aplikacja w TypeScriptie

Najlepszym sposobem na rozpoczęcie pracy z TypeScriptem jest zagłębienie się w ten język. W tym rozdziale przedstawię prosty proces programistyczny polegający na utworzeniu aplikacji pozwalającej na zdefiniowanie listy rzeczy do zrobienia. W kolejnych rozdziałach dokładniej poznasz sposób działania poszczególnych funkcji TypeScriptu, natomiast pokazany tutaj prosty przykład jest w zupełności wystarczający do zaprezentowania podstawowych funkcji tego języka. Nie przejmuj się, jeśli nie wszystko w rozdziale będzie dla Ciebie zrozumiałe. Moim celem jest jedynie pokazanie ogólnego trybu działania języka TypeScript i sposobu używania jego poszczególnych konstrukcji podczas tworzenia aplikacji.

Przygotowanie systemu

Aby móc wykonywać przykłady przedstawione w książce, konieczne jest zainstalowanie czterech pakietów. Przeprowadź instalację zgodnie z informacjami zamieszczonymi w kolejnych sekcjach i upewnij się, że pakiety działają zgodnie z oczekiwaniami.

Krok 1. Instalowanie Node.js

Przed wszystkim musisz pobrać i zainstalować środowisko uruchomieniowe Node.js (znane również pod nazwą Node). Niezbędny pakiet instalacyjny znajdziesz na stronie <https://nodejs.org/dist/v14.15.4/>. Na podanej stronie znajdują się pakiety instalacyjne dla wszystkich platform obsługiwanych przez wydanie 14.15.4 Node.js, czyli w wersji, której użyłem w przykładach omówionych w książce. Podczas instalacji upewnij się o wybraniu opcji powodującej zainstalowanie również menedżera pakietów Node.js (ang. *node package manager*). Po zakończeniu procesu instalacji przejdź do powłoki lub wiersza poleceń, a następnie wydaj polecenie przedstawione na listingu 1.1, aby w ten sposób sprawdzić poprawność działania Node.js i menedżera pakietów (npm).

Listing 1.1. Sprawdzanie poprawności instalacji Node i menedżera pakietów (npm)

```
$ node --version
$ npm -version
```

Wygenerowane dane wyjściowe pierwszego polecenia powinny mieć postać v14.15.4 wskazującą na dostępność Node.js w wymaganej wersji. Natomiast dane wyjściowe drugiego polecenia powinny mieć postać 6.14.10 wskazującą na prawidłowe zainstalowanie menedżera pakietów Node.js.

Krok 2. Instalowanie Gita

Drugim krokiem jest pobranie i zainstalowanie narzędzia systemu kontroli wersji Git — znajdziesz je na stronie <https://git-scm.com/download/>. Wprawdzie system Git nie jest ściśle wymagany podczas programowania w języku TypeScript, ale jest niezbędny dla części pakietów najczęściej używanych z TypeScriptem. Po zakończeniu instalacji użyj powłoki (Linux) lub wiersza poleceń (Windows) w celu wydania polecenia przedstawionego na listingu 1.2 i tym samym sprawdzenia, czy Git faktycznie działa.

Listing 1.2. Sprawdzenie poprawności instalacji narzędzia Git

```
$ git -version
```

W chwili powstawania książki najnowszą dostępną wersją narzędzia Git na wszystkich platformach była wersja 2.30.0.

Krok 3. Instalowanie TypeScriptu

Trzecim krokiem jest zainstalowanie pakietu języka TypeScript. Powłokę lub wiersz poleceń wykorzystaj do wydania polecenia przedstawionego na listingu 1.3.

Listing 1.3. Instalowanie pakietu języka TypeScript

```
$ npm install --global typescript@4.2.2
```

Po zainstalowaniu pakietu wydaj polecenie przedstawione na listingu 1.4, aby mieć pewność o prawidłowym zainstalowaniu kompilatora.

Listing 1.4. Testowanie poprawności instalacji kompilatora TypeScriptu

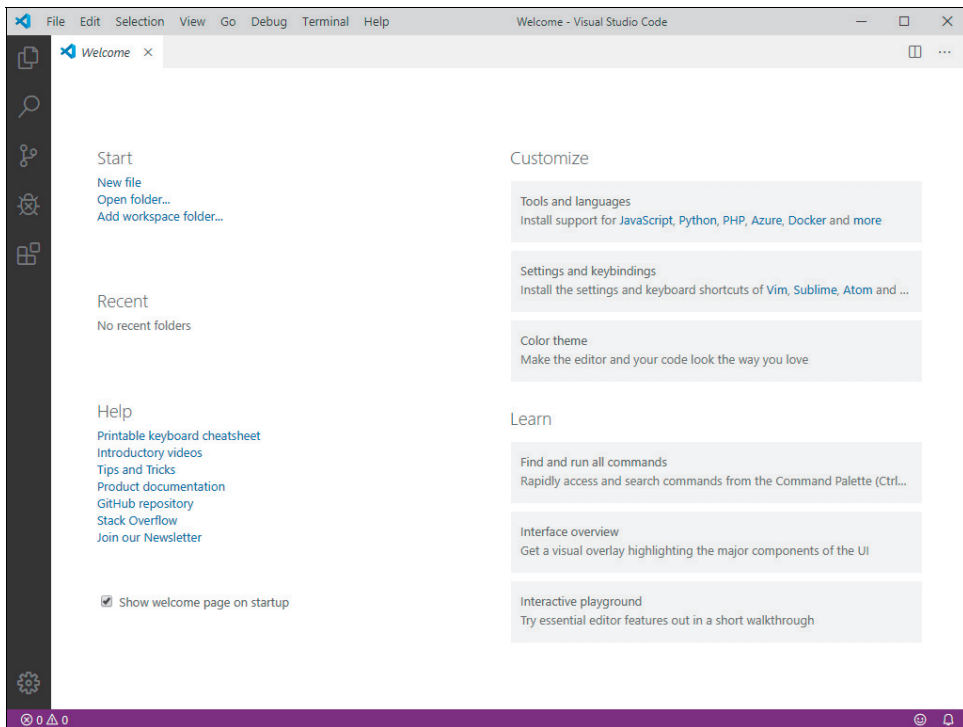
```
$ tsc -version
```

Kompilator języka TypeScript nosi nazwę tsc, a dane wyjściowe po wykonaniu polecenia przedstawionego na listingu 1.4 powinny mieć postać Version 4.2.2.

Krok 4. Instalowanie programistycznego edytora tekstu

Ostatnim krokiem jest zainstalowanie programistycznego edytora tekstu, który zapewnia obsługę języka TypeScript. Podczas programowania w TypeScriptie można skorzystać z większości popularnych edytorów tekstu. Jeżeli nie masz żadnego ulubionego, wówczas pobierz i zainstaluj Visual Studio Code ze strony <https://code.visualstudio.com/>. Visual Studio Code to edytor programistyczny rozprowadzany jako oprogramowanie typu *open source* dostępne dla wielu platform. Z tego edytora będę korzystał podczas tworzenia przykładów omówionych w książce.

Po zainstalowaniu Visual Studio Code polecenie `code` powoduje uruchomienie edytora. Ewentualnie możesz skorzystać z ikony programu dodanej podczas instalacji. Po uruchomieniu zobaczysz okno powitalne podobne do pokazanego na rysunku 1.1. (Konieczne może być zmodyfikowanie systemowej ścieżki dostępu, zanim będzie można korzystać z polecenia `code`).



Rysunek 1.1. Okno powitalne edytora Visual Studio Code

-
- **Wskazówka** Niektóre edytory tekstu pozwalają wybrać inną wersję TypeScriptu niż dodana do projektu. To może prowadzić do wyświetlania błędów w edytorze kodu źródłowego, nawet jeśli komunikaty generowane przez narzędzia powłoki wskazują na kompilację zakończoną sukcesem. W przypadku Visual Studio Code używana wersja TypeScriptu jest wyświetlana w prawym dolnym rogu okna edytora. Kliknij ją, wybierz opcję *Select TypeScript Version*, a następnie wskaż żądaną wersję.
-

Utworzenie projektu

Aby rozpocząć pracę z TypeScriptem, zamierzam utworzyć prostą aplikację pozwalającą na zdefiniowanie listy rzeczy do zrobienia. Język TypeScript jest najczęściej używany do tworzenia aplikacji internetowych, co w trzeciej części książki pokażę na przykładzie kilku najpopularniejszych frameworków (Angular, React i Vue.js). Natomiast w tym rozdziale utworzę aplikację działającą w powłoce, co pozwoli skoncentrować się na samym języku TypeScript i uniknąć niepotrzebnego poziomu skomplikowania narzucanego przez framework aplikacji internetowej.

Działanie aplikacji budowanej w rozdziale polega na wyświetleniu listy rzeczy do zrobienia. Użytkownik będzie miał możliwość dodawania nowych zadań, a także oznaczania istniejących zadań jako już wykonanych. Dostępna będzie również opcja pozwalająca filtrować na liście wykonane zadania. Gdy podstawowa funkcjonalność aplikacji zostanie ukończona, przystąpię do implementacji obsługi trwałego magazynu danych, aby zmiany wprowadzone w aplikacji nie były tracone po zakończeniu jej działania.

Inicjalizacja projektu

Aby przygotować katalog projektu, w powłoce lub w wierszu poleceń przejdź do wybranego katalogu, a następnie utwórz w nim podkatalog o nazwie *todo*. Teraz wydaj polecenia przedstawione na listingu 1.5, które spowodują przejście do nowo utworzonego katalogu i inicjalizację nowego projektu.

Listing 1.5. Inicjalizacja katalogu projektu

```
$ cd todo
$ npm init --yes
```

Działanie polecenia `npm init` polega na utworzeniu pliku *package.json*, który jest używany do monitorowania pakietów wymaganych przez projekt, a także do konfiguracji narzędzi programistycznych.

Utworzenie pliku konfiguracyjnego kompilatora

Pakiet TypeScriptu zainstalowany po wykonaniu polecenia przedstawionego na listingu 1.3 we wcześniejszej części rozdziału obejmuje kompilator `tsc` odpowiedzialny za kompilację kodu TypeScriptu na postać czystego kodu JavaScriptu. W celu zdefiniowania konfiguracji kompilatora TypeScriptu należy w katalogu projektu (tutaj *todo*) utworzyć plik o nazwie *tsconfig.json* z zawartością przedstawioną na listingu 1.6.

Listing 1.6. Zawartość pliku tsconfig.json w katalogu todo

```
{
  "compilerOptions": {
    "target": "es2018",
    "outDir": "./dist",
    "rootDir": "./src",
```

```
    "module": "commonjs"
  }
}
```

Dokładne omówienie kompilatora TypeScriptu znajdziesz w rozdziale 5. Teraz wystarczy wiedzieć, że przedstawione tutaj polecenia nakazują kompilatorowi użycie najnowszej wersji JavaScriptu, wskazują mu katalog *src* jako zawierający pliki TypeScriptu projektu, określają katalog *dist* jako docelowy dla wygenerowanych danych wyjściowych, a także nakazują użycie standardu *commonjs* podczas wczytywania kodu z oddzielnych plików.

Tworzenie pliku kodu TypeScriptu

Plik kodu TypeScriptu ma rozszerzenie *.ts*. W celu dodania pierwszego pliku kodu do projektu zacznij od utworzenia w katalogu projektu *todo* podkatalogu *src*, a następnie umieść w nim plik o nazwie *index.ts* z kodem przedstawionym na listingu 1.7. Ten plik przedstawia popularną konwencję nadania plikowi głównemu aplikacji nazwy *index* z rozszerzeniem *.ts* wskazującym na to, że zawartością pliku jest kod JavaScriptu.

Listing 1.7. Zawartość pliku *index.ts* w katalogu *src*

```
console.clear();
console.log("Lista Adama");
```

Ten plik zawiera zwykle polecenia JavaScriptu używające obiektu *console* do usunięcia zawartości okna powłoki, a następnie do wyświetlenia prostego komunikatu. To wystarczająca funkcjonalność do sprawdzenia, czy wszystko na pewno działa, zanim przystąpimy do pracy nad rzeczywistą aplikacją.

Kompilowanie i uruchamianie kodu

Plik TypeScriptu musi zostać skompilowany — dopiero wtedy na jego podstawie jest generowany kod JavaScriptu, który później będzie mógł być wykonany przez przeglądarkę WWW lub środowisko uruchomieniowe Node.js przygotowane na początku rozdziału. Polecenie przedstawione na listingu 1.8 należy wydać w katalogu *todo*, aby w ten sposób uruchomić kompilator.

Listing 1.8. Uruchamianie kompilatora TypeScriptu

```
$ tsc
```

Kompilator odczytuje ustawienia konfiguracyjne zapisane w pliku *tsconfig.json*, a potem odszukuje pliki TypeScriptu w katalogu *src*. Następnie tworzy podkatalog o nazwie *dist* i umieszcza w nim wygenerowany kod JavaScriptu. Jeżeli przeanalizujesz zawartość podkatalogu *dist*, to znajdziesz w nim plik *index.js*. Rozszerzenie *.js* wskazuje, że plik zawiera kod JavaScriptu. Z kolei po przeanalizowaniu pliku *index.js* przekonasz się, że zawiera on następujące polecenia:

```
console.clear();
console.log("Lista Adama");
```

Pliki TypeScriptu i JavaScriptu zawierają te same polecenia, ponieważ jeszcze nie zostały użyte żadne funkcje oferowane przez język TypeScript. Gdy aplikacja zacznie nabierać kształtu, zawartość plików TypeScriptu będzie różniła się od plików JavaScriptu wygenerowanych przez kompilator.

■ **Ostrzeżenie** Nie wprowadzaj zmian w plikach znajdujących się w katalogu *dist*, ponieważ będą one nadpisane w trakcie następnego uruchomienia kompilatora. Podczas programowania w języku TypeScript zmiany są wprowadzane w plikach z rozszerzeniem *.ts*, na podstawie których później są generowane pliki JavaScriptu z rozszerzeniem *.js*.

Aby uruchomić skompilowany kod, w powłoce przejdź do katalogu projektu, a następnie wydaj polecenie przedstawione na listingu 1.9.

Listing 1.9. *Uruchomienie skompilowanego kodu*

```
$ node dist/index.js
```

Polecenie node powoduje przejście do środowiska uruchomieniowego Node.js JavaScriptu, a argument wskazuje na plik, którego zawartość ma być wykonana. Jeżeli narzędzia programistyczne zostały zainstalowane prawidłowo, zawartość powłoki będzie usunięta i zobaczysz wyświetlone następujące dane wyjściowe:

```
Lista Adama
```

Definiowanie modelu danych

Działanie przykładowej aplikacji ma polegać na zarządzaniu listą rzeczy do zrobienia. Użytkownik będzie miał możliwość wyświetlenia listy, dodawania nowych elementów, oznaczania zadań jako wykonanych, a także filtrowania zawartości listy. W tej sekcji zacznę używać języka TypeScript do zdefiniowania modelu danych opisującego dane aplikacji i operacje, które mogą być na nich przeprowadzane. Pracę trzeba rozpocząć od utworzenia w katalogu *src* pliku o nazwie *todoItem.ts* i umieszczenia w nim kodu przedstawionego na listingu 1.10.

Listing 1.10. *Zawartość pliku *todoItem.ts* w katalogu *src**

```
export class TodoItem {
  public id: number;
  public task: string;
  public complete: boolean = false;

  public constructor(id: number, task: string, complete: boolean = false) {
    this.id = id;
    this.task = task;
    this.complete = complete;
  }
}
```

```

    public printDetails() : void {
        console.log(`${this.id}\\t${this.task} ${this.complete
            ? "\\t(wykonane)": ""}\\n`);
    }
}

```

Klasa to szablon opisujący typ danych. Dokładne omówienie klas znajdziesz w rozdziale 4. Kod przedstawiony na listingu 1.10 będzie wyglądał znajomo dla każdego, kto ma doświadczenie w programowaniu w językach takich jak C# lub Java, nawet jeśli dokładny sposób działania tego kodu pozostaje niejasny.

Klasa przedstawiona na listingu 1.10 nosi nazwę `TodoItem`, definiuje właściwości `id`, `task` i `complete`, a także metodę `printDetails()` wyświetlającą w konsoli podsumowanie danego zadania. TypeScript jest zbudowany na bazie JavaScriptu, więc nic dziwnego, że kod znajdujący się na listingu 1.10 jest połączeniem standardowej funkcjonalności JavaScriptu z usprawnieniami oferowanymi przez TypeScript. JavaScript obsługuje klasy z konstruktorami, właściwościami i metodami, natomiast funkcje takie jak słowa kluczowe kontroli dostępu (np. `public`) są dostarczane przez TypeScript. Ważną cechą języka TypeScript jest statyczne typowanie, co pozwala na wyraźne zdefiniowanie typu każdej właściwości i każdego parametru klasy `TodoItem`:

```

...
public id: number;
...

```

To jest przykład *adnotacji typu* wskazującej kompilatorowi TypeScriptu, że właściwości `id` można przypisać jedynie wartości typu `number`. Jak wyjaśnię w rozdziale 3., JavaScript stosuje dość elastyczne podejście do typów. Największą zaletą oferowaną przez TypeScript jest zapewnienie większej zgodności typów danych z innymi językami programowania przy jednoczesnym zachowaniu dostępu do normalnego podejścia JavaScriptu, gdy zachodzi potrzeba.

■ **Wskazówka** Nie przejmuj się, jeśli nie znasz sposobu, w jaki JavaScript obsługuje typy danych. W rozdziałach 3. i 4. znajdziesz więcej informacji o funkcjach JavaScriptu, które trzeba opanować, aby móc efektywnie programować w języku JavaScript.

Klasę przedstawioną na listingu 1.10 utworzyłem w celu podkreślenia podobieństwa zachodzącego między TypeScriptem a językami takimi jak C# i Java. Jednak to nie jest sposób, w jaki zwykle definiuje się klasę w kodzie TypeScriptu. Dlatego też na listingu 1.11 przedstawiłem zmodyfikowaną wersję klasy `TodoItem` wykorzystującą funkcje TypeScriptu pozwalające na zwięzłe definiowanie klas.

Listing 1.11. Zwięźlejsza postać kodu pliku `todoItem.ts` znajdującego się w katalogu `src`

```

export class TodoItem {

    constructor(public id: number,
                public task: string,
                public complete: boolean = false) {
        // Polecenia nie są wymagane.
    }

    printDetails() : void {

```

```

        console.log(`${this.id}\\t${this.task} ${this.complete}
            ? "\\t(wykonane)": ""}`);
    }
}

```

Obsługa statycznych typów danych to tylko jeden z aspektów języka TypeScript mających na celu zapewnienie możliwości wygenerowania przewidywalnego i działającego bezpiecznie kodu JavaScriptu. Związła składnia przedstawiona na listingu 1.11 pozwala klasie `TodoItem` otrzymywać parametry i używać ich do tworzenia właściwości egzemplarza w jednym kroku, unikając tym samym podatnego na błędy podejścia polegającego na definiowaniu właściwości i wyraźnego przypisywania jej wartości przekazywanych za pomocą parametru.

Zmiana w metodzie `printDetails()` polega na usunięciu kontrolującego sposób dostępu słowa kluczowego `public`, które tak naprawdę jest niepotrzebne — w języku TypeScript przyjęto założenie, że wszystkie metody i właściwości są domyślnie publiczne, o ile nie zostało użyte słowo kluczowe wskazujące na inny rodzaj dostępu. (W konstruktorze nadal znajduje się słowo kluczowe `public`, ponieważ w ten sposób kompilator TypeScriptu rozpoznaje, że ma do czynienia ze związłą składnią konstruktora. Więcej informacji na ten temat znajdziesz w rozdziale 11.).

Tworzenie klasy kolekcji elementów listy rzeczy do zrobienia

Następnym krokiem jest utworzenie klasy zbierającej wszystkie elementy listy rzeczy do zrobienia, co pozwoli na znacznie łatwiejsze zarządzanie nimi. W katalogu `src` utwórz plik o nazwie `todoCollection.ts` i umieść w nim kod przedstawiony na listingu 1.12.

Listing 1.12. Zawartość pliku `todoCollection.ts` w katalogu `src`

```

import { TodoItem } from "./todoItem";

export class TodoCollection {
    private nextId: number = 1;

    constructor(public userName: string, public todoItems: TodoItem[] = []) {
        // Polecenia nie są wymagane.
    }

    addTodo(task: string): number {
        while (this.getTodoById(this.nextId)) {
            this.nextId++;
        }
        this.todoItems.push(new TodoItem(this.nextId, task));
        return this.nextId;
    }

    getTodoById(id: number) : TodoItem {
        return this.todoItems.find(item => item.id === id);
    }

    markComplete(id: number, complete: boolean) {
        const todoItem = this.getTodoById(id);
        if (todoItem) {
            todoItem.complete = complete;
        }
    }
}

```

Implementowanie podstawowych funkcji modelu danych

Zanim przejdę dalej, chciałbym się upewnić o prawidłowym działaniu początkowo zaimplementowanych funkcji klasy `TodoCollection`. Temat przeprowadzania testów jednostkowych w projektach TypeScriptu zostanie omówiony w rozdziale 6., natomiast w tym miejscu wystarczające będzie utworzenie obiektów typu `TodoItem` i umieszczenie ich w obiekcie `TodoCollection`. Na listingu 1.13 przedstawiłem kod, którym należy zastąpić dotychczasową zawartość pliku `index.ts` utworzonego na początku rozdziału.

Listing 1.13. Testowanie modelu danych za pomocą kodu pliku `index.ts` znajdującego się w katalogu `src`

```
import { TodoItem } from "./todoItem";
import { TodoCollection } from "./todoCollection";

let todos = [
  new TodoItem(1, "Kupić kwiaty"), new TodoItem(2, "Odebrać buty"),
  new TodoItem(3, "Zamówić bilety"), new TodoItem(4, "Zadzwonić do Janka", true)];

let collection = new TodoCollection("Adam", todos);

console.clear();
console.log(`Lista ${collection.userName}a`);

let newId = collection.addTodo("Iść pobiegać");
let todoItem = collection.getTodoById(newId);
console.log(JSON.stringify(todoItem));
```

Wszystkie polecenia przedstawione na listingu 1.13 to standardowe funkcje JavaScriptu. Polecenie `import` jest używane w celu zadeklarowania zależności od klas `TodoItem` i `TodoCollection`. To polecenie jest częścią funkcji modułów w JavaScriptcie pozwalającej na definiowanie kodu w wielu plikach (dokładnie omówię to w rozdziale 4.). Definiowanie tablicy i używanie słowa kluczowego `new` w celu utworzenia egzemplarza klas to również standardowe funkcje JavaScriptu, podobnie jak wywołania obiektu `console`.

■ **Uwaga** Kod przedstawiony na listingu 1.13 używa funkcji będących dodatkami do języka JavaScript. Jak to wyjaśnię w rozdziale 5., kompilator TypeScriptu znacznie ułatwia korzystanie z nowoczesnych funkcji języka JavaScript, takich jak słowo kluczowe `let`, nawet jeśli nie są one obsługiwane przez środowisko uruchomieniowe przeznaczone do wykonania tego kodu, np. starsze wersje przeglądarek WWW. Funkcje JavaScriptu niezbędne do efektywnego programowania w języku TypeScript omówię w rozdziałach 3. i 4.

Kompilator TypeScriptu stara się być użyteczny dla programisty. W trakcie kompilacji następuje sprawdzenie używanych typów danych, a informacje podane w definicjach klas `TodoItem` i `TodoCollection` okazują się pomocne podczas ustalania typów danych wykorzystywanych w kodzie przedstawionym na listingu 1.13. Kod wygenerowany przez kompilator nie będzie zawierał żadnych statycznych informacji o typie, mimo to kompilator i tak może przeprowadzić sprawdzenie kodu pod kątem zapewnienia bezpieczeństwa w trakcie jego wykonywania. Aby przekonać się, jak to działa, dodaj do pliku `index.ts` polecenia przedstawione na listingu 1.14.

Listing 1.14. Dodawanie polecenia do pliku *index.ts* znajdującego się w katalogu *src*

```
import { TodoItem } from "./todoItem";
import { TodoCollection } from "./todoCollection";

let todos = [
    new TodoItem(1, "Kupić kwiaty"), new TodoItem(2, "Odebrać buty"),
    new TodoItem(3, "Zamówić bilety"), new TodoItem(4, "Zadzwoń do Janka", true)];

let collection = new TodoCollection("Adam", todos);

console.clear();
console.log(`Lista ${collection.userName}a`);

let newId = collection.addToDo("Iść pobiegać");
let todoItem = collection.getToDoById(newId);
todoItem.printDetails();

collection.addToDo(todoItem);
```

Ostatnie polecenie wywołuje metodę `collection.addToDo()` z argumentem w postaci obiektu typu `TodoItem`. Kompilator sprawdza definicję metody `addToDo()` w pliku *todoItem.ts*, a następnie ustala, że oczekuje ona innego typu danych:

```
...
addToDo(task: string): number {
    while (this.getToDoById(this.nextId)) {
        this.nextId++;
    }
    this.todoItems.push(new TodoItem(this.nextId, task));
    return this.nextId;
}
...
```

Informacje o typie metody `addToDo()` wskazują kompilatorowi, że parametr `task` musi być typu `string`, a wynik musi być typu `number`. (Typy `string` i `number` są typami wbudowanymi w JavaScript; więcej informacji na ich temat znajdziesz w rozdziale 3.). Wykonanie w katalogu *todo* polecenia przedstawionego na listingu 1.15 spowoduje kompilację kodu.

Listing 1.15. Uruchamianie kompilatora

```
$ tsc
```

Kompilator TypeScriptu przetworzy kod projektu i odkryje, że wartość parametru używanego do wywoływania metody `addToDo()` ma nieprawidłowy typ danych. W efekcie zostanie wygenerowany następujący komunikat błędu:

```
src/index.ts:17:20 - error TS2345: Argument of type 'TodoItem' is not assignable to
parameter of type 'string'.
17 collection.addToDo(todoItem);
                        ~~~~~~

Found 1 error.
```

TypeScript doskonale radzi sobie z wykrywaniem problemów i ustaleniem tego, co tak naprawdę się dzieje w kodzie. Dlatego też w projekcie możesz podać jedynie minimalną ilość informacji o typie. W książce będę podawał informacje o typie, aby listingi były łatwiejsze do zrozumienia, ponieważ wiele omówionych przykładów ma związek ze sposobem, w jaki kompilator TypeScriptu obsługuje typy danych. Na listingu 1.16 przedstawiłem dodanie do pliku *index.ts* poleceń definiujących typy i umieszczenie w komentarzu polecenia, które spowodowało wygenerowanie błędu.

Listing 1.16. Dodawanie informacji o typach do pliku *index.ts* znajdującego się w katalogu *src*

```
import { TodoItem } from "./todoItem";
import { TodoCollection } from "./todoCollection";

let todos: TodoItem[] = [
  new TodoItem(1, "Kupić kwiaty"), new TodoItem(2, "Odebrać buty"),
  new TodoItem(3, "Zamówić bilety"), new TodoItem(4, "Zadzwonić do Janka", true)];

let collection: TodoCollection = new TodoCollection("Adam", todos);

console.clear();
console.log(`Lista ${collection.userName}a`);

let newId: number = collection.addToDo("Iść pobiegać");
let todoItem: TodoItem = collection.getToDoById(newId);
todoItem.printDetails();

//collection.addToDo(todoItem);
```

Wprowadź informacje dodane przez nowe polecenia przedstawione na listingu 1.16 nie zmieniają sposobu działania kodu, ale powodują wyraźne użycie typów danych, co może ułatwić zrozumienie przeznaczenia tego kodu, a kompilator nie musi już samodzielnie ustalać niezbędnych typów danych. W katalogu *todo* wydaj polecenia przedstawione na listingu 1.17, aby w ten sposób skompilować i uruchomić kod.

Listing 1.17. Kompilowanie i wykonywanie kodu przykładowej aplikacji

```
$ tsc
$ node dist/index.js
```

Po wykonaniu kodu zostaną wygenerowane następujące dane wyjściowe:

```
Lista Adama
5      Iść pobiegać
```

Dodawanie funkcji do klasy kolekcji

Następnym krokiem jest dodanie nowych możliwości do klasy *TodoCollection*. Przede wszystkim trzeba zmienić sposób przechowywania obiektów *TodoItem* — teraz będzie użyty obiekt *Map* JavaScriptu, jak pokazałem na listingu 1.18.

Listing 1.18. Użycie obiektu typu *Map* w kodzie pliku *todoCollection.ts* znajdującym się w katalogu *src*

```
import { TodoItem } from "./todoItem";

export class TodoCollection {
  private nextId: number = 1;
  private itemMap = new Map<number, TodoItem>();

  constructor(public userName: string, todoItems: TodoItem[] = []) {
    todoItems.forEach(item => this.itemMap.set(item.id, item));
  }

  addTodo(task: string): number {
    while (this.getTodoById(this.nextId)) {
      this.nextId++;
    }
    this.itemMap.set(this.nextId, new TodoItem(this.nextId, task));
    return this.nextId;
  }

  getTodoById(id: number) : TodoItem {
    return this.itemMap.get(id);
  }

  markComplete(id: number, complete: boolean) {
    const todoItem = this.getTodoById(id);
    if (todoItem) {
      todoItem.complete = complete;
    }
  }
}
```

TypeScript obsługuje typy generyczne, czyli tak naprawdę miejsca zarezerwowane, które są wypełniane podczas tworzenia obiektu. Przykładowo obiekt *Map* w JavaScriptcie to ogólnego przeznaczenia kolekcja przechowująca pary klucz-wartość. Ponieważ JavaScript ma system typów dynamicznych, kolekcję *Map* można wykorzystać do przechowywania elementów różnych typów danych, używając do tego odmiennych typów kluczy. Aby ograniczyć typy możliwe do obsługi przez kolekcję *Map*, w kodzie przedstawionym na listingu 1.18 dostarczyłem ogólne argumenty typu wskazujące kompilatorowi TypeScriptu typy dozwolone do używania przez klucze i wartości:

```
...
private itemMap = new Map<number, TodoItem>();
...
```

Argumenty typu generycznego zostały umieszczone w nawiasie ostrym, a kolekcja *Map* na listingu 1.18 otrzymująca te argumenty wskazuje kompilatorowi, że będzie przechowywała obiekty typu *TodoItem* z kluczami w postaci wartości typu *number*. Kompilator wygeneruje komunikat błędu, jeśli polecenie spróbuje przechowywać w tym obiekcie inny typ danych lub jeśli klucz nie będzie wartością typu *number*. Typy generyczne to ważna cecha języka TypeScript, a ich dokładne omówienie znajdziesz w rozdziale 12.

Zapewnienie dostępu do elementów listy rzeczy do zrobienia

Klasa `TodoCollection` definiuje metodę `getTodoById()`, ale aplikacja musi mieć możliwość wyświetlenia listy elementów, które opcjonalnie będą filtrowane, aby wyeliminować już wykonane zadania. Na listingu 1.19 pokazałem dodanie metody zapewniającej dostęp do obiektów `TodoItem`, którymi zarządza `TodoCollection`.

Listing 1.19. Modyfikacja kodu pliku `todoCollection.ts` w katalogu `src` mająca na celu zapewnienie dostępu do elementów listy

```
import { TodoItem } from "../todoItem";

export class TodoCollection {
  private nextId: number = 1;
  private itemMap = new Map<number, TodoItem>();

  constructor(public userName: string, todoItems: TodoItem[] = []) {
    todoItems.forEach(item => this.itemMap.set(item.id, item));
  }

  addTodo(task: string): number {
    while (this.getTodoById(this.nextId)) {
      this.nextId++;
    }
    this.itemMap.set(this.nextId, new TodoItem(this.nextId, task));
    return this.nextId;
  }

  getTodoById(id: number) : TodoItem {
    return this.itemMap.get(id);
  }

  getTodoItems(includeComplete: boolean): TodoItem[] {
    return [...this.itemMap.values()]
      .filter(item => includeComplete || !item.complete);
  }

  markComplete(id: number, complete: boolean) {
    const todoItem = this.getTodoById(id);
    if (todoItem) {
      todoItem.complete = complete;
    }
  }
}
```

Metoda `getTodoItems()` pobiera obiekty z kolekcji `Map`, używając jej metody `values()`, a następnie na podstawie tych obiektów tworzy tablicę za pomocą operatora JavaScriptu w postaci trzech kropek. Obiekty są przetwarzane z użyciem metody `filter()` w celu pobrania wymaganych obiektów na podstawie parametru `includeComplete`.

Kompilator TypeScriptu wykorzystuje otrzymane informacje o typach do ich monitorowania na każdym kroku. Argumenty typu generycznego zostały użyte do utworzenia kolekcji `Map` wskazującej kompilatorowi, że zawiera ona obiekty `TodoItem`. Dlatego też kompilator wie, że wartością zwrótną metody `values()` są obiekty typu `TodoItem`, który jest jednocześnie typem

obiektów w tablicy. Podążając za tym, kompilator wie, że funkcja przekazana metodzie `filter()` będzie przetwarzała obiekty `TodoItem`, z których każdy zdefiniuje właściwość `complete`. Jeżeli nastąpi próba odczytania właściwości lub metody niezdefiniowanej przez klasę `TodoItem`, kompilator wygeneruje komunikat błędu. Błąd zostanie zgłoszony również wtedy, gdy wynik zwrócony przez polecenie `return` nie będzie odpowiadał typowi wyniku zadeklarowanego przez metodę.

Na listingu 1.20 przedstawiłem uaktualniony kod pliku `index.ts`, który teraz używa nowej funkcji klasy `TodoCollection` i wyświetla użytkownikowi listę rzeczy do zrobienia.

Listing 1.20. Pobieranie elementów kolekcji przez kod pliku `index.ts` znajdującego się w katalogu `src`

```
import { TodoItem } from "../todoItem";
import { TodoCollection } from "../todoCollection";

let todos: TodoItem[] = [
    new TodoItem(1, "Kupić kwiaty"), new TodoItem(2, "Odebrać buty"),
    new TodoItem(3, "Zamówić bilety"), new TodoItem(4, "Zadzwoń do Janka", true)];

let collection: TodoCollection = new TodoCollection("Adam", todos);

console.clear();
console.log(`Lista ${collection.userName}a`);

//collection.addToDo(todoItem);
collection.getToDoItems(true).forEach(item => item.printDetails());
```

Nowe polecenie wywołuje metodę `getToDoItems()` zdefiniowaną na listingu 1.19 i używa standardowej metody JavaScriptu `forEach()` w celu wyświetlenia opisu poszczególnych obiektów `TodoItem` za pomocą obiektu `console`.

W katalogu `todo` wydaj polecenia przedstawione na listingu 1.21, aby w ten sposób skompilować i uruchomić kod.

Listing 1.21. Kompilowanie i wykonywanie kodu przykładowej aplikacji

```
$ tsc
$ node dist/index.js
```

Po wykonaniu kodu zostaną wygenerowane następujące dane wyjściowe:

```
Lista Adama
1      Kupić kwiaty
2      Odebrać buty
3      Zamówić bilety
4      Zadzwoń do Janka      (wykonane)
```

Usuwanie wykonanych zadań

Dodawanie nowych zadań i później oznaczanie ich jako wykonanych będzie prowadziło do zwiększania się liczby elementów kolekcji, która w pewnym momencie stanie się trudna w zarządzaniu. Dlatego też na listingu 1.22 przedstawiłem metodę pozwalającą na usunięcie z kolekcji już wykonanych zadań.

Listing 1.22. Modyfikacja kodu w pliku *todoCollection.ts* znajdującym się w katalogu *src* pozwalająca na usuwanie wykonanych zadań z kolekcji

```
import { TodoItem } from "../todoItem";

export class TodoCollection {
  private nextId: number = 1;
  private itemMap = new Map<number, TodoItem>();

  constructor(public userName: string, todoItems: TodoItem[] = []) {
    todoItems.forEach(item => this.itemMap.set(item.id, item));
  }

  addTodo(task: string): number {
    while (this.getTodoById(this.nextId)) {
      this.nextId++;
    }
    this.itemMap.set(this.nextId, new TodoItem(this.nextId, task));
    return this.nextId;
  }

  getTodoById(id: number) : TodoItem {
    return this.itemMap.get(id);
  }

  getTodoItems(includeComplete: boolean): TodoItem[] {
    return [...this.itemMap.values()]
      .filter(item => includeComplete || !item.complete);
  }

  markComplete(id: number, complete: boolean) {
    const todoItem = this.getTodoById(id);
    if (todoItem) {
      todoItem.complete = complete;
    }
  }

  removeComplete() {
    this.itemMap.forEach(item => {
      if (item.complete) {
        this.itemMap.delete(item.id);
      }
    })
  }
}
```

Funkcja `removeComplete()` używa metody `Map.forEach()` do przeanalizowania przechowywanych w kolekcji `Map` elementów `TodoItem` i wywołuje metodę `delete()` dla tych, których właściwość `complete` ma przypisaną wartość `true`. Uaktualniony kod w pliku *index.ts* pozwalający na wywołanie metody `removeComplete()` przedstawiłem na listingu 1.23.

Listing 1.23. Sprawdzanie możliwości usunięcia z kolekcji już wykonanego zadania — zmianę należy wprowadzić w pliku `src/index.ts`

```
import { TodoItem } from "./todoItem";
import { TodoCollection } from "./todoCollection";

let todos: TodoItem[] = [
    new TodoItem(1, "Kupić kwiaty"), new TodoItem(2, "Odebrać buty"),
    new TodoItem(3, "Zamówić bilety"), new TodoItem(4, "Zadzwoń do Janka", true)];

let collection: TodoCollection = new TodoCollection("Adam", todos);

console.clear();
console.log(`Lista ${collection.userName}a`);

//collection.addTo(todoItem);
collection.removeComplete();
collection.getItems(true).forEach(item => item.printDetails());
```

W katalogu *todo* wydaj polecenia przedstawione na listingu 1.24, aby w ten sposób skompilować i uruchomić kod.

Listing 1.24. Kompilowanie i wykonywanie kodu przykładowej aplikacji

```
$ tsc
$ node dist/index.js
```

Po wykonaniu kodu zostaną wygenerowane następujące dane wyjściowe potwierdzające usunięcie wykonanych zadań z kolekcji:

```
Lista Adama
1      Kupić kwiaty
2      Odebrać buty
3      Zamówić bilety
```

Obsługa licznika elementów

Ostatnią funkcjonalnością, którą chcę dodać do klasy `TodoCollection`, jest obsługa licznika całkowitej liczby elementów `TodoItem`, liczby zadań już wykonanych i liczby zadań wciąż oczekujących na wykonanie.

W kodzie przedstawionym na listingach we wcześniejszej części rozdziału skoncentrowałem się na klasach, ponieważ w taki właśnie sposób większość programistów tworzy typy danych. Obiekt JavaScriptu może zostać zdefiniowany za pomocą składni literału, a TypeScript ma możliwość sprawdzenia typu i wymuszenia typu statycznego, podobnie jak w przypadku tworzenia obiektu na podstawie klasy. Gdy pracujesz z literałami, kompilator TypeScriptu koncentruje się na połączeniu nazw właściwości i typów ich wartości, co jest określane mianem *kształtu* obiektu. Konkretnie połączenie nazw i typów nosi nazwę *kształtu typu*. Na listingu 1.25 pokazałem dodanie do klasy `TodoCollection` metody zwracającej obiekt opisujący liczbę elementów znajdujących się w kolekcji.

Listing 1.25. Pozwalająca na używanie kształtu typu modyfikacja kodu w pliku *todoCollection.ts* znajdującym się w katalogu *src*

```
import { TodoItem } from "../todoItem";

type ItemCounts = {
  total: number,
  incomplete: number
}

export class TodoCollection {
  private nextId: number = 1;
  private itemMap = new Map<number, TodoItem>();

  constructor(public userName: string, todoItems: TodoItem[] = []) {
    todoItems.forEach(item => this.itemMap.set(item.id, item));
  }

  addTodo(task: string): number {
    while (this.getTodoById(this.nextId)) {
      this.nextId++;
    }
    this.itemMap.set(this.nextId, new TodoItem(this.nextId, task));
    return this.nextId;
  }

  getTodoById(id: number) : TodoItem {
    return this.itemMap.get(id);
  }

  getTodoItems(includeComplete: boolean): TodoItem[] {
    return [...this.itemMap.values()]
      .filter(item => includeComplete || !item.complete);
  }

  markComplete(id: number, complete: boolean) {
    const todoItem = this.getTodoById(id);
    if (todoItem) {
      todoItem.complete = complete;
    }
  }

  removeComplete() {
    this.itemMap.forEach(item => {
      if (item.complete) {
        this.itemMap.delete(item.id);
      }
    })
  }

  getItemCounts(): ItemCounts {
    return {
      total: this.itemMap.size,
      incomplete: this.getTodoItems(false).length
    };
  }
}
```

Słowo kluczowe `type` zostało użyte do utworzenia tzw. *aliasu typu*, czyli wygodnego rozwiązania pozwalającego na przypisanie nazwy kształtowi typu. Alias typu na listingu 1.25 opisuje obiekty zawierające dwie właściwości typu `number`: o nazwach `total` i `incomplete`. Alias typu jest używany jako wynik działania metody `getItemCounts()`, która z kolei wykorzystuje składnię literału obiektu JavaScriptu do utworzenia obiektu o kształcie typu dopasowanym do aliasu typu. Na listingu 1.26 przedstawiłem zmodyfikowaną wersję pliku `index.ts`, która teraz wyświetla użytkownikowi liczbę jeszcze niewykonanych zadań.

Listing 1.26. Modyfikacja pliku `index.ts` w katalogu `src` pozwalająca na wyświetlanie liczby elementów na liście rzeczy do zrobienia

```
import { TodoItem } from "../todoItem";
import { TodoCollection } from "../todoCollection";

let todos: TodoItem[] = [
  new TodoItem(1, "Kupić kwiaty"), new TodoItem(2, "Odebrać buty"),
  new TodoItem(3, "Zamówić bilety"), new TodoItem(4, "Zadzwoń do Janka", true)];

let collection: TodoCollection = new TodoCollection("Adam", todos);

console.clear();
console.log(`Lista ${collection.userName}a `
  + `(liczba zadań pozostałych do zrobienia: ${ collection.getItemCounts().incomplete })`);
collection.getTodoItems(true).forEach(item => item.printDetails());
```

W katalogu *todo* wydaj polecenia przedstawione na listingu 1.27, aby w ten sposób skompilować i uruchomić kod.

Listing 1.27. Kompilowanie i wykonywanie kodu przykładowej aplikacji

```
$ tsc
$ node dist/index.js
```

Oto dane wyjściowe po wykonaniu przedstawionego kodu:

```
Lista Adama (liczba zadań pozostałych do zrobienia: 3)
1      Kupić kwiaty
2      Odebrać buty
3      Zamówić bilety
4      Zadzwoń do Janka          (wykonane)
```

Używanie pakietu zewnętrznego

Jedną z zalet tworzenia kodu JavaScriptu jest istnienie ekosystemu pakietów możliwych do wykorzystania w projektach. TypeScript pozwala na używanie pakietów JavaScriptu, ale po zapewnieniu obsługi typowania statycznego. W tym podrozdziale zamierzam skorzystać z doskonałego pakietu `Inquirer.js` (<https://github.com/SBoudrias/Inquirer.js>) przeznaczonego do pobierania informacji od użytkownika i przetwarzania odpowiedzi. Aby dodać ten pakiet do projektu, przejdź do katalogu *todo* i wydaj polecenie przedstawione na listingu 1.28.

Listing 1.28. Dodawanie pakietu do projektu

```
$ npm install inquirer@7.3.3
```

Dodawanie pakietów do projektów TypeScriptu odbywa się w dokładnie taki sam sposób, jak w przypadku zwykłego kodu JavaScriptu, czyli za pomocą polecenia `npm install`. Na listingu 1.29 przedstawiłem zmodyfikowaną wersję kodu w pliku *index.ts*, który teraz korzysta z nowo dodanego pakietu.

Listing 1.29. Zmodyfikowana wersja kodu znajdującego się w pliku *index.ts* w katalogu *src* i wykorzystującego nowy pakiet

```
import { TodoItem } from "./todoItem";
import { TodoCollection } from "./todoCollection";
import * as inquirer from 'inquirer';

let todos: TodoItem[] = [
    new TodoItem(1, "Kupić kwiaty"), new TodoItem(2, "Odebrać buty"),
    new TodoItem(3, "Zamówić bilety"), new TodoItem(4, "Zadzwonić do Janka", true)];

let collection: TodoCollection = new TodoCollection("Adam", todos);

function displayTodoList(): void {
    console.log(`Lista ${collection.userName}a `
        + `~(liczba zadań pozostałych do zrobienia: ${
collection.getItemCounts().incomplete })~`);
    collection.getTodoItems(true).forEach(item => item.printDetails());
}

enum Commands {
    Quit = "Koniec"
}

function promptUser(): void {
    console.clear();
    displayTodoList();
    inquirer.prompt({
        type: "list",
        name: "command",
        message: "Wybierz opcję",
        choices: Object.values(Commands)
    }).then(answers => {
        if (answers["command"] !== Commands.Quit) {
            promptUser();
        }
    })
}

promptUser();
```

TypeScript nie przeszkadza w korzystaniu z kodu JavaScriptu, a zmiany wprowadzone na listingu 1.29 pozwalają na używanie pakietu *Inquirer.js* do zaproponowania użytkownikowi zestawu dostępnych poleceń. Obecnie obsługiwane jest tylko jedno polecenie, *Koniec*, przy czym obsługa kolejnych zostanie wkrótce dodana.

■ **Wskazówka** W książce nie zamierzam szczegółowo omawiać API pakietu `Inquirer.js`, ponieważ nie ma on bezpośredniego związku z językiem TypeScript. Jeżeli chcesz z tego pakietu korzystać we własnych projektach, więcej informacji na jego temat znajdziesz na stronie <https://github.com/SBoudrias/Inquirer.js>.

Metoda `inquirer.prompt()` jest używana do pobierania danych od użytkownika i została skonfigurowana do pracy z obiektem JavaScriptu. Wybrane przeze mnie opcje konfiguracyjne zostaną przedstawione użytkownikowi w postaci listy, po której może się poruszać za pomocą klawiszy kursora, a wybór opcji następuje po naciśnięciu klawisza *Enter*. Gdy użytkownik dokona wyboru, wywołana zostanie funkcja przekazana metodzie `then()`, a nazwa tej funkcji będzie dostępna za pomocą właściwości `answers.command`.

Na listingu 1.29 pokazałem, jak używać kodu TypeScriptu i JavaScriptu pakietu `Inquirer.js`. Słowo kluczowe `enum` to funkcjonalność oferowana przez TypeScript i pozwalająca na nadawanie nazw wartościom, jak to dokładnie omówię w rozdziale 9. W tym przypadku `enum` pozwala zdefiniować polecenia i odwoływać się do nich bez konieczności powielania w aplikacji wartości w postaci ciągów tekstowych. Wartości pochodzące z konstrukcji `enum` są używane razem z funkcjami `Inquirer.js`, np.:

```
...
if (answers["command"] !== Commands.Quit) {
...

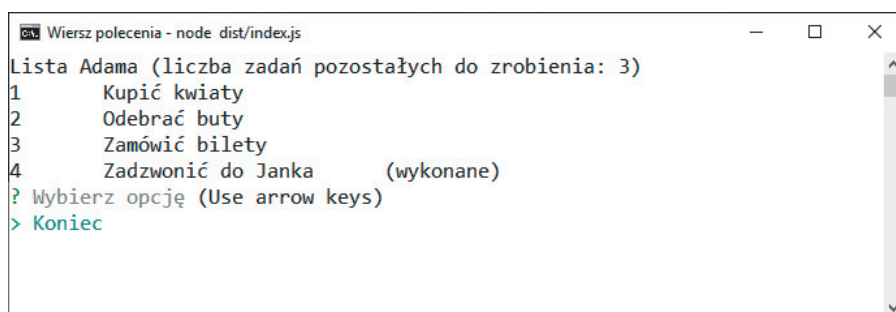
```

W katalogu *todo* wydaj polecenia przedstawione na listingu 1.30, aby w ten sposób skompilować i uruchomić kod.

Listing 1.30. Kompilowanie i wykonywanie kodu przykładowej aplikacji

```
$ tsc
$ node dist/index.js
```

Po wykonaniu tego kodu zostanie wyświetlona lista rzeczy do zrobienia, a także lista dostępnych opcji, jak pokazałem na rysunku 1.2, choć w tym momencie to tylko jedna opcja.



```
Wiersz polecenia - node dist/index.js
Lista Adama (liczba zadań pozostałych do zrobienia: 3)
1      Kupić kwiaty
2      Odebrać buty
3      Zamówić bilety
4      Zadzwońić do Janka      (wykonane)
? Wybierz opcję (Use arrow keys)
> Koniec
```

Rysunek 1.2. Wyświetlenie użytkownikowi dostępnych opcji

Jeżeli naciśniesz klawisz *Enter*, zostanie wybrana opcja *Koniec* i program zakończy działanie.

Dodawanie deklaracji typu dla pakietu JavaScriptu

TypeScript pozwala na używanie kodu JavaScriptu, choć jednocześnie nie zapewnia żadnej pomocy w tym zakresie. Kompilator nie ma informacji o typach danych używanych przez pakiet `Inquirer.js` i musi polegać na tym, że podczas wyświetlenia opcji użytkownikowi zostały zastosowane odpowiednie typy argumentów, a otrzymane w odpowiedzi obiekty są przetwarzane w sposób bezpieczny.

Istnieją dwa sposoby na dostarczenie językowi TypeScript informacji wymaganych przez ten język w celu zastosowania statycznego typowania. Pierwszy polega na samodzielnym opisywaniu typów. Dostarczane przez TypeScript funkcje przeznaczone do opisywania kodu JavaScriptu przedstawię w rozdziale 14. Samodzielne opisywanie kodu JavaScriptu nie jest trudne, choć wymaga nieco czasu i dobrej znajomości opisywanego kodu.

Drugi sposób polega na wykorzystaniu przygotowanych przez kogoś innego definicji typów. Projekt `Definitely Typed` to repozytorium deklaracji typu JavaScriptu dla tysięcy pakietów JavaScriptu, w tym także dla `Inquirer.js`. Aby zainstalować deklaracje typu, należy w katalogu *todo* wydać polecenie przedstawione na listingu 1.31.

Listing 1.31. Instalowanie deklaracji typów

```
$ npm install --save-dev @types/inquirer
```

Deklaracje typów są instalowane za pomocą polecenia `npm install`, podobnie jak pakiety JavaScriptu. Argument `save-dev` jest stosowany w przypadku pakietów używanych podczas pracy nad aplikacją, ale niebędących jej częścią. Nazwa pakietu rozpoczyna się prefiksem `@types/`, a następnie znajduje się nazwa pakietu, dla którego są wymagane deklaracje typów. W przypadku pakietu `Inquirer.js` pakietem deklaracji typów jest `@types/inquirer`, ponieważ `inquirer` to nazwa użyta do zainstalowania pakietu JavaScriptu.

■ **Uwaga** Więcej informacji o projekcie `Definitely Typed` oraz o dostępnych deklaracjach typów znajdziesz na stronie <https://github.com/DefinitelyTyped/DefinitelyTyped>.

Kompilator TypeScriptu automatycznie wykrywa deklaracje typów, więc polecenie przedstawione na listingu 1.31 pozwala kompilatorowi na sprawdzanie typów danych używanych przez API `Inquirer.js`. Aby pokazać efekt zastosowania deklaracji typów, na listingu 1.32 przedstawiłem konfigurację właściwości nieobsługiwanej przez pakiet `Inquirer.js`.

Listing 1.32. Dodawanie właściwości w pliku `index.ts` znajdującym się w katalogu `src`

```
...
function promptUser(): void {
  console.clear();
  inquirer.prompt({
    type: "list",
    name: "command",
    message: "Wybierz opcję",
    choices: Object.values(Commands),
  });
}
```

```

        badProperty: true
    }).then(answers => {
        // Nie jest wymagane żadne działanie.
        if (answers["command"] !== Commands.Quit) {
            promptUser();
        }
    })
}
...

```

API Inquirer.js nie zawiera właściwości konfiguracyjnej o nazwie `badProperty`. W katalogu *todo* wydaj polecenie przedstawione na listingu 1.33, aby w ten sposób skompilować kod.

Listing 1.33. Kompilowanie kodu przykładowej aplikacji

```
$ tsc
```

Kompilator wykorzysta informacje o typach zainstalowane po wykonaniu polecenia przedstawionego na listingu 1.31 i wygeneruje następujący komunikat błędu:

```

src/index.ts:25:13 - error TS2322: Type '"list"' is not assignable to type '"number"'.
25             type: "list",
               ~~~~
Found 1 error.

```

Deklaracja typu pozwala TypeScriptowi na dostarczanie tego samego zestawu funkcjonalności w całej aplikacji, nawet mimo że pakiet Inquirer.js został utworzony w czystym kodzie JavaScriptu, a nie TypeScriptu. Jednak ta funkcjonalność ma pewne ograniczenia, jak możesz zobaczyć w omawianym przykładzie. Dodanie nieobsługiwanej właściwości spowodowało wygenerowanie błędu dotyczącego wartości przypisywanej właściwości `type`. Ten komunikat wynika stąd, że czasami jest po prostu trudno opisywać typy za pomocą czystego kodu JavaScriptu, w jeszcze innych sytuacjach zaś komunikaty błędów mogą być ogólnymi informacjami o pojawieniu się problemu.

Dodawanie poleceń

Przykładowa aplikacja na razie nie działa zbyt dobrze i wymaga zdefiniowania kolejnych poleceń. W tym podrozdziale zajmę się przygotowaniem nowych poleceń i dostarczeniem ich implementacji.

Filtrowanie elementów

Pierwsze dodawane polecenie pozwoli użytkownikowi na włączenie lub wyłączenie filtrowania wykonanych zadań. Modyfikacje konieczne do wprowadzenia w kodzie przedstawiłem na listingu 1.34.

Listing 1.34. Implementacja w kodzie pliku *index.ts* w katalogu *src* operacji filtrowania elementów listy rzeczy do zrobienia

```
import { TodoItem } from "./todoItem";
import { TodoCollection } from "./todoCollection";
import * as inquirer from 'inquirer';

let todos: TodoItem[] = [
  new TodoItem(1, "Kupić kwiaty"), new TodoItem(2, "Odebrać buty"),
  new TodoItem(3, "Zamówić bilety"), new TodoItem(4, "Zadzwonić do Janka", true)];

let collection: TodoCollection = new TodoCollection("Adam", todos);
let showCompleted = true;

function displayTodoList(): void {
  console.log(`Lista ${collection.userName}a ~
    + ~(liczba zadań pozostałych do zrobienia: ${
collection.getItemCounts().incomplete })`);
  collection.getTodoItems(showCompleted).forEach(item => item.printDetails());
}

enum Commands {
  Toggle = "Pokaż lub ukryj wykonane",
  Quit = "Koniec"
}

function promptUser(): void {
  console.clear();
  displayTodoList();
  inquirer.prompt({
    type: "list",
    name: "command",
    message: "Wybierz opcję",
    choices: Object.values(Commands),
    //badProperty: true
  }).then(answers => {
    switch (answers["command"]) {
      case Commands.Toggle:
        showCompleted = !showCompleted;
        promptUser();
        break;
    }
  })
}

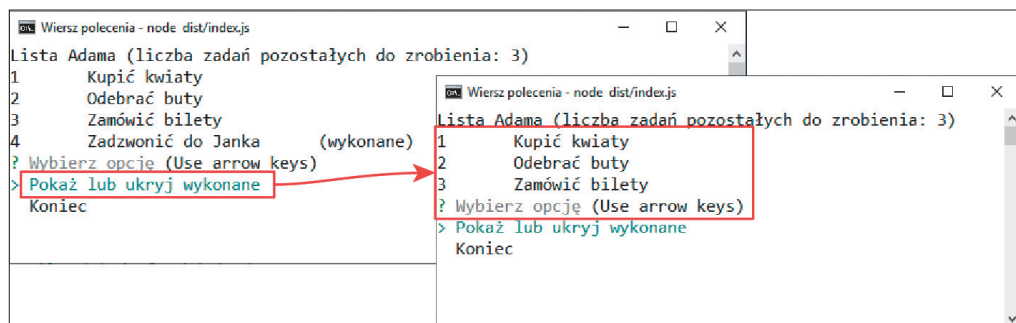
promptUser();
```

Proces dodawania opcji sprowadza się do zdefiniowania nowej wartości dla wyliczenia `Commands` i określenia poleceń wykonywanych po wybraniu danej opcji. W omawianym przykładzie nową wartością jest `Toggle`, a po jej wybraniu wartość zmiennej `showCompleted` zostanie zmieniona, aby funkcja `displayTodoList()` uwzględniła (lub nie) wykonane zadania. W katalogu *todo* wydaj polecenia przedstawione na listingu 1.35, aby w ten sposób skompilować i uruchomić kod.

Listing 1.35. Kompilowanie i wykonywanie kodu przykładowej aplikacji

```
$ tsc
$ node dist/index.js
```

Wybierz opcję *Pokaż lub ukryj wykonane* i naciśnij klawisz *Enter*, aby w ten sposób włączyć lub wyłączyć wyświetlanie na liście wykonanych zadań, jak pokazałem na rysunku 1.3.



Rysunek 1.3. Pokazywanie lub ukrywanie wykonanych zadań

Dodawanie zadań

Przykładowa aplikacja nie będzie za bardzo użyteczna, jeśli użytkownik nie otrzyma możliwości tworzenia nowych elementów na liście. Na listingu 1.36 przedstawiłem zmiany w kodzie pozwalające na tworzenie nowych obiektów typu `TodoItem`.

Listing 1.36. Zmiany w kodzie pliku `index.ts` w katalogu `src` pozwalające na dodawanie nowych zadań na liście

```
import { TodoItem } from './todoItem';
import { TodoCollection } from './todoCollection';
import * as inquirer from 'inquirer';

let todos: TodoItem[] = [
    new TodoItem(1, "Kupić kwiaty"), new TodoItem(2, "Odebrać buty"),
    new TodoItem(3, "Zamówić bilety"), new TodoItem(4, "Zadzwoń do Janka", true)];

let collection: TodoCollection = new TodoCollection("Adam", todos);
let showCompleted = true;

function displayTodoList(): void {
    console.log(`Lista ${collection.userName}a ~
        + ~(liczba zadań pozostałych do zrobienia: ${
collection.getItemCounts().incomplete })`);
    collection.getTodoItems(showCompleted).forEach(item => item.printDetails());
}

enum Commands {
    Add = "Dodaj nowe zadanie",
```

```

    Toggle = "Pokaż lub ukryj wykonane",
    Quit = "Koniec"
  }

  function promptAdd(): void {
    console.clear();
    inquirer.prompt({ type: "input", name: "add", message: "Podaj zadanie:" })
      .then(answers => {if (answers["add"] !== "") {
        collection.addToDo(answers["add"]);
      }
      promptUser();
    })
  }

  function promptUser(): void {
    console.clear();
    displayToDoList();
    inquirer.prompt({
      type: "list",
      name: "command",
      message: "Wybierz opcję",
      choices: Object.values(Commands),
    }).then(answers => {
      switch (answers["command"]) {
        case Commands.Toggle:
          showCompleted = !showCompleted;
          promptUser();
          break;
        case Commands.Add:
          promptAdd();
          break;
      }
    })
  }

  promptUser();

```

Pakiet Inquirer.js może wyświetlać użytkownikowi różne rodzaje pytań. Gdy użytkownik wybierze polecenie Add, wówczas typ `input` będzie użyty w celu pobrania zadania, które następnie stanie się argumentem wywołania metody `TodoCollection.addToDo()`. W katalogu *todo* wydaj polecenia przedstawione na listingu 1.37, aby w ten sposób skompilować i uruchomić kod.

Listing 1.37. *Kompilowanie i wykonywanie kodu przykładowej aplikacji*

```

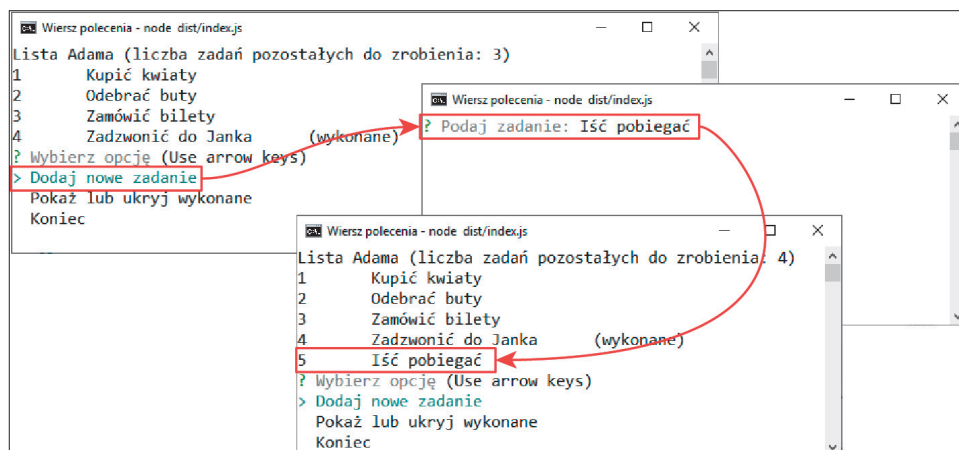
$ tsc
$ node dist/index.js

```

Wybierz teraz opcję *Dodaj nowe zadanie*, wpisz treść zadania i naciśnij klawisz *Enter*, aby dodać nowe zadanie, jak pokazałem na rysunku 1.4.

Oznaczanie zadania jako wykonanego

Wykonanie zadania to proces składający się z dwóch etapów i wymagający od użytkownika wskazania elementu listy, który ma być oznaczony jako wykonany. Na listingu 1.38 przedstawiłem zdefiniowanie dodatkowego polecenia i pytania, co pozwoli użytkownikowi na oznaczenie zadania jako wykonanego oraz na usuwanie już wykonanych zadań.



Rysunek 1.4. Dodawanie nowego zadania na liście rzeczy do zrobienia

Listing 1.38. Modyfikacje w kodzie pliku `index.ts` w katalogu `src` pozwalające na oznaczanie zadania jako już wykonanego

```
import { TodoItem } from './todoItem';
import { TodoCollection } from './todoCollection';
import * as inquirer from 'inquirer';

let todos: TodoItem[] = [
    new TodoItem(1, "Kupić kwiaty"), new TodoItem(2, "Odebrać buty"),
    new TodoItem(3, "Zamówić bilety"), new TodoItem(4, "Zadzwońić do Janka", true)];

let collection: TodoCollection = new TodoCollection("Adam", todos);
let showCompleted = true;

function displayTodoList(): void {
    console.log(`Lista ${collection.userName}a ~
    + ~(liczba zadań pozostałych do zrobienia: ${
collection.getItemCounts().incomplete }`);
    collection.getTodoItems(showCompleted).forEach(item => item.printDetails());
}

enum Commands {
    Add = "Dodaj nowe zadanie",
    Complete = "Wykonanie zadania",
    Toggle = "Pokaż lub ukryj wykonane",
    Purge = "Usuń wykonane zadania",
    Quit = "Koniec"
}
```



```

function promptAdd(): void {
  console.clear();
  inquirer.prompt({ type: "input", name: "add", message: "Podaj zadanie:" })
    .then(answers => {if (answers["add"] !== "") {
      collection.addToDo(answers["add"]);
    }
  })
  promptUser();
})
}

```

```

function promptComplete(): void {
  console.clear();
  inquirer.prompt({ type: "checkbox", name: "complete",
    message: "Oznaczenie zadań jako wykonanych",
    choices: collection.getToDoItems(showCompleted).map(item =>
      ({name: item.task, value: item.id, checked: item.complete}))
  }).then(answers => {
    let completedTasks = answers["complete"] as number[];
    collection.getToDoItems(true).forEach(item =>
      collection.markComplete(item.id,
        completedTasks.find(id => id === item.id) != undefined));
    promptUser();
  })
}

```

```

function promptUser(): void {
  console.clear();
  displayToDoList();
  inquirer.prompt({
    type: "list",
    name: "command",
    message: "Wybierz opcję",
    choices: Object.values(Commands),
  }).then(answers => {
    switch (answers["command"]) {
      case Commands.Toggle:
        showCompleted = !showCompleted;
        promptUser();
        break;
      case Commands.Add:
        promptAdd();
        break;
      case Commands.Complete:
        if (collection.getItemCounts().incomplete > 0) {
          promptComplete();
        } else {
          promptUser();
        }
        break;
      case Commands.Purge:
        collection.removeComplete();
        promptUser();
        break;
    }
  })
}

```

```
}
promptUser();
```

Wprowadzone zmiany powodują dodanie nowego polecenia wyświetlanego użytkownikowi z listą zadań i pozwalającego na zmianę ich stanu. Zmienna `showCompleted` jest używana do ustalenia, czy wykonane zadania mają się pojawiać na liście, oraz tworzy połączenie między poleceniami `Toggle` i `Complete`.

Jedyna nowa funkcjonalność TypeScriptu znalazła się w następującym poleceniu:

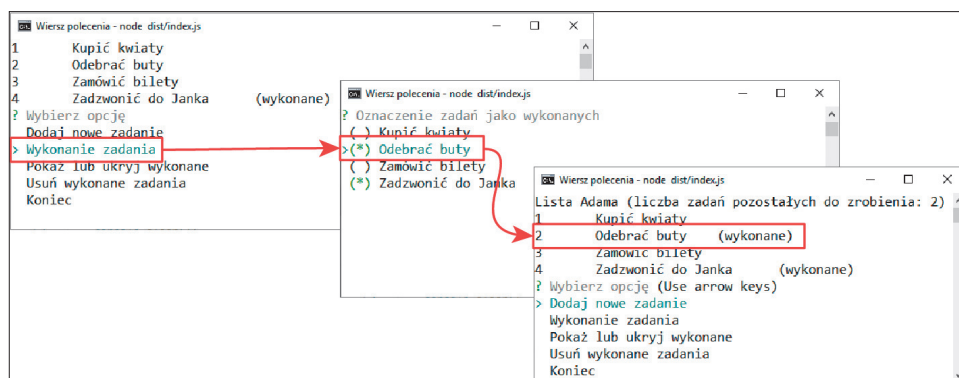
```
...
let completedTasks = answers["complete"] as number[];
...
```

Nawet pomimo istnienia definicji typów zdarzają się sytuacje, w których TypeScript nie potrafi prawidłowo ustalić używanego typu. W takim przypadku pakiet `Inquirer.js` pozwala na użycie dowolnego typu danych w pytaniu wyświetlonym użytkownikowi, a kompilator nie będzie w stanie ustalić, że wykorzystane są tylko wartości typu `number`, co oznacza możliwość otrzymania odpowiedzi jedynie w postaci wartości typu `number`. Do rozwiązania tego problemu wykorzystałem *asercję typu*, która pozwala wskazać kompilatorowi, że powinien używać podanego typu, nawet jeśli zidentyfikował inny typ danych (lub nie ustalił żadnego). Podczas stosowania asercji zachowania kompilatora są nadpisywane, co oznacza, że programista staje się odpowiedzialny za zagwarantowanie użycia prawidłowego typu. W katalogu *todo* wydaj polecenia przedstawione na listingu 1.39, aby w ten sposób skompilować i uruchomić kod.

Listing 1.39. Kompilowanie i wykonywanie kodu przykładowej aplikacji

```
$ tsc
$ node dist/index.js
```

Wybierz opcję *Wykonanie zadania*, zaznacz jedno lub więcej zadań, których stan ma się zmienić, i naciśnij klawisz `Enter`. Stan wybranych zadań zostanie zmieniony, co będzie odzwierciedlone na liście, jak pokazałem na rysunku 1.5.



Rysunek 1.5. Oznaczenie zadania jako wykonanego

Trwałe przechowywanie danych

Aby zapewnić możliwość trwałego przechowywania danych, zamierzam skorzystać z kolejnego pakietu typu *open source*, ponieważ nie ma żadnego sensu tworzenie funkcjonalności, gdy dostępne są już gotowe i doskonale przetestowane alternatywy. W katalogu *todo* wydaj polecenia przedstawione na listingu 1.40, aby w ten sposób zainstalować pakiet Lowdb i definicje typów dostarczające TypeScriptowi opis API tego pakietu.

Listing 1.40. Dodawanie pakietu i definicji typu

```
$ npm install lowdb@1.0.0
$ npm install --save-dev @types/lowdb
```

Lowdb to doskonały pakiet bazy danych przechowujący dane w pliku JSON i używany jako komponent magazynu danych w pakiecie json-server, który w trzeciej części książki będzie wykorzystany podczas tworzenia usług sieciowych HTTP.

■ **Wskazówka** W książce nie zamierzam szczegółowo omawiać API pakietu Lowdb, ponieważ nie ma on bezpośredniego związku z językiem TypeScript. Jeżeli chcesz z tego pakietu korzystać we własnych projektach, więcej informacji na jego temat znajdziesz na stronie <https://github.com/typecode/lowdb>.

Implementacja trwałego magazynu danych będzie się odbywała poprzez klasę `TodoCollection`. Na etapie przygotowań trzeba zmienić używane przez tę klasę słowo kluczowe kontroli dostępu, aby jej podklasy mogły uzyskiwać dostęp do kolekcji `Map` zawierającej obiekty `TodoItem`. Zmianę konieczną do wprowadzenia przedstawiłem na listingu 1.41.

Listing 1.41. Zmiana ustawień kontroli dostępu w pliku `todoCollection.ts` znajdującym się w katalogu `src`

```
import { TodoItem } from "../todoItem";

type ItemCounts = {
  total: number,
  incomplete: number
}

export class TodoCollection {
  private nextId: number = 1;
  protected itemMap = new Map<number, TodoItem>();

  constructor(public userName: string, todoItems: TodoItem[] = []) {
    todoItems.forEach(item => this.itemMap.set(item.id, item));
  }

  // Pozostałe metody zostały pominięte.
}
```

Słowo kluczowe `protected` wskazuje kompilatorowi TypeScriptu, że właściwość jest dostępna jedynie dla klasy i jej podklas. W celu utworzenia podklasy należy dodać do katalogu `src` plik o nazwie `jsonTodoCollection.ts` z kodem przedstawionym na listingu 1.42.

Listing 1.42. Zawartość pliku *jsonTodoCollection.ts* w katalogu *src*

```
import { TodoItem } from "./todoItem";
import { TodoCollection } from "./todoCollection";
import * as lowdb from "lowdb";
import * as FileSync from "lowdb/adapters/FileSync";

type schemaType = {
  tasks: { id: number; task: string; complete: boolean; }[]
};

export class JsonTodoCollection extends TodoCollection {
  private database: lowdb.LowdbSync<schemaType>;

  constructor(public userName: string, todoItems: TodoItem[] = []) {
    super(userName, []);
    this.database = lowdb(new FileSync("Todos.json"));
    if (this.database.has("tasks").value()) {
      let dbItems = this.database.get("tasks").value();
      dbItems.forEach(item => this.itemMap.set(item.id,
        new TodoItem(item.id, item.task, item.complete)));
    } else {
      this.database.set("tasks", todoItems).write();
      todoItems.forEach(item => this.itemMap.set(item.id, item));
    }
  }

  addTodo(task: string): number {
    let result = super.addTodo(task);
    this.storeTasks();
    return result;
  }

  markComplete(id: number, complete: boolean): void {
    super.markComplete(id, complete);
    this.storeTasks();
  }

  removeComplete(): void {
    super.removeComplete();
    this.storeTasks();
  }

  private storeTasks() {
    this.database.set("tasks", [...this.itemMap.values()]).write();
  }
}
```

Definicje typu dla pakietu Lowdb używają schematu do opisanie struktury przechowywanych danych. Ten schemat jest stosowany za pomocą argumentów typu generycznego, aby kompilator TypeScriptu mógł sprawdzić używane typy danych. W omawianej tutaj aplikacji trzeba przechowywać dane tylko jednego typu, który zostanie opisany za pomocą aliasu typu.

```
...
type schemaType = {
  tasks: { id: number; task: string; complete: boolean; }[]
```

```
};
...
```

Typ schematu jest stosowany podczas tworzenia bazy danych Lowdb, a kompilator ma możliwość sprawdzenia sposobu używania danych w trakcie ich odczytywania z bazy danych, jak w przedstawionym tutaj poleceniu:

```
...
let dbItems = this.database.get("tasks").value();
...
```

Kompilator wie, że argument `tasks` odpowiada właściwości `tasks` w typie schematu, więc wynikiem operacji `get()` jest tablica obiektów z właściwościami `id`, `task` i `complete`.

Stosowanie klasy trwałego magazynu danych

Na listingu 1.43 przedstawiłem przykład użycia klasy `JsonTodoCollection` w pliku `index.ts`, aby umożliwić przykładowej aplikacji trwałe przechowywanie danych.

Listing 1.43. *Używanie trwałej kolekcji danych w pliku `index.ts` w katalogu `src`*

```
...
import { TodoItem } from "./todoItem";
import { TodoCollection } from "./todoCollection";
import * as inquirer from 'inquirer';
import { JsonTodoCollection } from "./jsonTodoCollection";

let todos: TodoItem[] = [
  new TodoItem(1, "Kupić kwiaty"), new TodoItem(2, "Odebrać buty"),
  new TodoItem(3, "Zamówić bilety"), new TodoItem(4, "Zadzwoń do Janka", true)];

let collection: TodoCollection = new JsonTodoCollection("Adam", todos);
let showCompleted = true;
...
```

W katalogu `todo` wydaj polecenia przedstawione na listingu 1.44, aby w ten sposób po raz ostatni w rozdziale skompilować i uruchomić kod.

Listing 1.44. *Kompilowanie i wykonywanie kodu przykładowej aplikacji*

```
$ tsc
$ node dist/index.js
```

Po uruchomieniu aplikacji w katalogu `todo` zostanie utworzony plik o nazwie `Todos.json`, który będzie zawierał reprezentację JSON obiektów `TodoItem`. Dzięki temu zmiany wprowadzone w aplikacji nie zostaną utracone po zakończeniu jej działania.

Podsumowanie

W tym rozdziale utworzyłem prostą aplikację, aby przedstawić wprowadzenie do programowania w języku TypeScript oraz zaprezentować kilka najważniejszych koncepcji tego języka. Przekonałeś się, że TypeScript dostarcza funkcje uzupełniające język JavaScript, skoncentrowane na zapewnieniu bezpieczeństwa typu i pomagające w unikaniu błędów najczęściej popełnianych przez programistów, zwłaszcza tych, którzy wcześniej mieli doświadczenie w pracy z językami takimi jak C# i Java.

Dowiedziałeś się, że TypeScript nie jest używany oddzielnie, a środowisko uruchomieniowe JavaScriptu jest niezbędne do wykonania kodu JavaScriptu wygenerowanego przez kompilator TypeScriptu. Zaletą przedstawionego podejścia jest to, że projekty utworzone za pomocą TypeScriptu mają pełny dostęp do szerokiej gamy dostępnych pakietów JavaScriptu, z których wiele zawiera definicje typów ułatwiające wykorzystanie tych pakietów w języku TypeScript.

Aplikacja utworzona w rozdziale stosuje kilka najważniejszych funkcji TypeScriptu. Jak możesz się domyślać na podstawie wielkości książki, tych funkcji jest znacznie więcej. W następnym rozdziale dowiesz się nieco więcej na temat powiązania języków TypeScript i JavaScript oraz poznasz strukturę i treść książki.

ROZDZIAŁ 2.



Poznajemy TypeScript

TypeScript to nadzbiór języka JavaScript koncentrujący się na generowaniu przewidywalnego i zapewniającego bezpieczeństwo kodu, który może być wykonywany przez dowolne środowisko uruchomieniowe JavaScriptu. Jako najważniejszą cechę TypeScriptu podaje się typowanie statyczne, które pozwala na pracę z JavaScriptem w znacznie bardziej przewidywalny sposób programistom mającym doświadczenie w programowaniu z użyciem języków takich jak C# i Java. W książce zamierzam wyjaśnić, czym jest TypeScript, i przedstawić możliwości oferowane przez ten język.

Niniejsza książka i harmonogram wydań języka TypeScript

Język TypeScript jest dość często uaktualniany, co oznacza nieustanny strumień poprawek i nowych funkcji. Oczekiwanie od czytelników zakupu co kilka miesięcy nowego wydania książki nie wydaje się ani w porządku, ani rozsądne — istnieje prawdopodobieństwo, że większość funkcji TypeScriptu pozostanie niezmienną nawet w nowej wersji języka. Dlatego też zdecydowałem się na publikowanie odpowiednich aktualizacji w repozytorium GitHub, które znajdziesz pod adresem <https://github.com/Apress/essential-typescript-4>.

To jest eksperyment zarówno dla mnie, jak i dla wydawnictwa. W tym momencie nawet jeszcze nie wiem, jaką postać te aktualizacje przyjmą, m.in. dlatego, że nie znam zmian, które będą wprowadzane w nowych wydaniach języka TypeScript. Moim celem jest maksymalne wydłużenie czasu, przez który ta książka pozostanie użyteczna, co chcę osiągnąć, uaktualniając zaprezentowane w niej przykłady.

W tym miejscu nie składam żadnych obietnic dotyczących wspomnianych aktualizacji, ich postaci ani czasu, w którym będą się pojawiały, zanim zdecyduję się na umieszczenie ich w kolejnym wydaniu książki. Pamiętaj o tych uaktualnieniach i po pojawieniu się nowej wersji TypeScriptu sprawdzaj repozytorium przygotowane dla tej książki. Jeżeli masz propozycje dotyczące usprawnienia publikowanych przeze mnie uaktualnień, napisz mi o tym, wysyłając wiadomość na adres adam@adam-freeman.com.

Dlaczego powinieneś używać języka TypeScript?

TypeScript nie jest rozwiązaniem każdego problemu i dlatego bardzo ważna jest wiedza, kiedy należy go stosować, a kiedy nie. W tym podrozdziale przedstawię oferowane przez TypeScript funkcje wysokiego poziomu i sytuacje, w których okazują się one użyteczne.

Funkcje języka TypeScript zwiększające produktywność programisty

Główną i często podkreślaną cechą języka TypeScript związaną szczególnie z produktywnością programisty jest typowanie statyczne, które ułatwia pracę z systemem typów JavaScriptu. Inne funkcje dotyczące produktywności, takie jak słowa kluczowe kontroli dostępu i zwięzła składnia konstruktora klasy, pomagają w unikaniu najczęściej popełnianych błędów podczas tworzenia kodu.

Związane z produktywnością funkcje TypeScriptu są stosowane dla kodu JavaScriptu. Jak się dowiedziałeś w rozdziale 1., pakiet języka TypeScript zawiera kompilator przetwarzający pliki TypeScriptu i generujący czysty kod JavaScriptu, który może być później wykonywany przez środowisko uruchomieniowe JavaScriptu, takie jak Node.js lub przeglądarki WWW, co schematycznie pokazałem na rysunku 2.1.



Rysunek 2.1. Konwersja kodu TypeScriptu na kod JavaScriptu

Połączenie funkcji JavaScriptu i TypeScriptu pozwala na zachowanie większości elastyczności i dynamicznej natury języka JavaScript przy jednoczesnym ograniczeniu używania typów danych, aby pozostały one znane i bardziej przewidywalne dla większości programistów. Oznacza to również, że w projekcie używającym języka TypeScript można wykorzystać szeroką gamę dostępnych pakietów JavaScriptu w celu zapewnienia konkretnej funkcjonalności (np. opcje powłoki, jak to zaprezentowałem w rozdziale 1.) lub też zastosować całe frameworki przeznaczone do tworzenia aplikacji (np. React, Angular i Vue.js — takie rozwiązania dokładnie omówię w trzeciej części książki).

Funkcje języka TypeScript mogą być stosowane selektywnie, co pozwala na wykorzystanie tylko tych, które okazują się użyteczne w danym projekcie. Jeżeli dopiero zaczynasz programowanie z zastosowaniem języków TypeScript i JavaScript, wówczas prawdopodobnie będziesz używać wszystkich funkcji oferowanych przez TypeScript. Gdy zdobędziesz większe doświadczenie i pogłębisz wiedzę, przekonasz się, że używasz funkcji TypeScriptu z większą rozważą i stosujesz je względem tych fragmentów tworzonego kodu, które są szczególnie skomplikowane lub mogą powodować duże problemy.

Poznajemy ograniczenia funkcji związanych z produktywnością

Część funkcji TypeScriptu jest implementowana całkowicie przez kompilator i nie pozostawiają one żadnych śladów w kodzie JavaScriptu wykonywanym po uruchomieniu aplikacji. Kolejne funkcje są implementowane przez utworzenie standardowego kodu JavaScriptu i przeprowadzenie na nim operacji dodatkowych podczas kompilacji. Często oznacza to konieczność poznania sposobu działania danej funkcjonalności *oraz* sposobu jej implementacji, ponieważ tylko wtedy można osiągnąć najlepszy wynik. To wyraźnie pokazuje, że funkcje TypeScriptu wydają się niespójne i nie do końca jasne.

Ogólnie rzecz biorąc, TypeScript stanowi usprawnienie dla JavaScriptu, choć wynikiem działania kompilatora TypeScriptu jest kod w języku JavaScript, natomiast proces opracowywania projektu TypeScriptu to w dużej mierze tworzenie kodu JavaScriptu. Część programistów stosuje język TypeScript, ponieważ chcą oni tworzyć aplikacje internetowe bez konieczności poznawania sposobu działania JavaScriptu. Dostrzegają fakt, że TypeScript został opracowany przez firmę Microsoft, i postrzegają TypeScript jako odmianę C# lub Javy przeznaczoną do tworzenia aplikacji internetowych, co jest założeniem błędnym i prowadzącym do zakłopotania oraz wielu frustracji.

Efektywne używanie TypeScriptu wymaga pogłębionej znajomości JavaScriptu i istnieją ku temu dobre powody. W rozdziałach 3. i 4. przedstawię te funkcje JavaScriptu, które musisz poznać, aby móc jak najlepiej wykorzystać TypeScript oraz zdobyć solidne podstawy pozwalające dostrzec, dlaczego TypeScript jest narzędziem o naprawdę potężnych możliwościach.

Jeżeli poznasz system typów JavaScriptu, przekonasz się, że używanie języka TypeScript jest przyjemnością. Z kolei jeśli nie zamierzasz poświęcić czasu na poznanie JavaScriptu, w ogóle nie powinieneś zabierać się za programowanie z użyciem TypeScriptu. Dodanie TypeScriptu do projektu, gdy nie masz wiedzy z zakresu JavaScriptu, znacznie utrudnia programowanie, ponieważ musisz zmagać się z dwoma językami, z których żaden nie działa dokładnie w oczekiwany przez Ciebie sposób.

Poznanie wersji JavaScriptu

JavaScript miał burzliwą historię, choć ostatnio jego twórcy skoncentrowali się na standaryzacji i modernizacji, a także wprowadzili nowe funkcje, dzięki którym JavaScript stał się językiem łatwiejszym w użyciu. Problem polega na tym, że wciąż istnieje wiele środowisk uruchomieniowych JavaScriptu pozbawionych obsługi nowoczesnych funkcji — mam tutaj na myśli przede wszystkim starsze przeglądarki WWW, co ogranicza możliwości programowania w JavaScriptcie do zaledwie niewielkiego zbioru wszechstronnie obsługiwanych funkcji. Dokładne poznanie JavaScriptu może być trudnym zadaniem, a sytuację jeszcze bardziej komplikuje to, że nawet po opanowaniu tego języka zdarzają się przypadki, gdy nie można korzystać z funkcji ułatwiających programowanie.

Kompilator TypeScriptu potrafi skonwertować kod JavaScriptu utworzony z wykorzystaniem nowoczesnych funkcji na kod zgodny ze starszymi wersjami JavaScriptu. Dzięki temu możliwe jest korzystanie z najnowszych funkcji JavaScriptu podczas programowania w języku TypeScript, a wynikiem będzie wygenerowany kod, który działa również w starszych wersjach środowiska uruchomieniowego JavaScriptu.

Poznanawanie ograniczeń wersji JavaScriptu

Kompilator TypeScriptu całkiem dobrze radzi sobie z większością funkcji języka, choć części funkcji nie można efektywnie skonwertować na postać zgodną ze starszymi wersjami środowiska uruchomieniowego. Jeżeli gotowa aplikacja ma działać także w najstarszych wersjach JavaScriptu, przekonasz się, że nie wszystkie nowoczesne funkcje JavaScriptu mogą być używane podczas tworzenia aplikacji, ponieważ kompilator TypeScriptu nie będzie potrafił ich skonwertować na kod wyrażony za pomocą starych wersji JavaScriptu.

Mając to na uwadze, konieczność wygenerowania przestarzałego kodu JavaScriptu nie ma ważnego znaczenia we wszystkich projektach, ponieważ kompilator TypeScriptu to tylko jeden z elementów dość obszernego łańcucha narzędzi. Wprawdzie kompilator TypeScriptu jest odpowiedzialny za stosowanie funkcji TypeScriptu, ale wynikiem jest nowoczesny kod JavaScriptu, który następnie będzie przetwarzany przez jeszcze inne narzędzia. Takie podejście jest powszechnie stosowane podczas tworzenia aplikacji internetowych, a przykłady jego wykorzystania zobaczysz w trzeciej części książki.

Co powinieneś wiedzieć?

Jeżeli uznałeś TypeScript za odpowiedni język programowania dla swojego projektu, powinieneś zdobyć ogólne umiejętności w zakresie używania typów danych i znajomości podstawowych funkcji JavaScriptu. Jednak nie przejmuj się, jeśli dokładnie nie rozumiesz, jak JavaScript radzi sobie z typami danych, ponieważ w rozdziałach 3. i 4. zamieściłem wprowadzenie do wszystkich funkcji JavaScriptu, które pomagają w zrozumieniu sposobu działania TypeScriptu. Natomiast w trzeciej części książki pokażę, jak połączyć TypeScript z popularnymi frameworkami tworzenia aplikacji internetowych, więc te przykłady wymagają znajomości technologii HTML-a i CSS.

Jak skonfigurować środowisko programistyczne?

Jedynie narzędzia programistyczne potrzebne do tworzenia aplikacji za pomocą języka TypeScript to te, które zainstalowałeś w rozdziale 1. podczas pracy nad pierwszą aplikacją. W dalszej części książki będzie zachodziła potrzeba instalowania dodatkowych pakietów i wówczas przedstawię odpowiednie procedury. Jeżeli udało Ci się utworzyć aplikację w rozdziale 1., jesteś gotów na poznawanie materiału zaprezentowanego w pozostałych rozdziałach książki.

Jaka jest struktura książki?

Książka została podzielona na trzy części, z których każda zawiera omówienie powiązanych ze sobą tematów.

- Część I, „Zaczynamy”, zawiera informacje niezbędne do rozpoczęcia programowania z użyciem języka TypeScript. Obejmuje rozdziały 1., bieżący, 3. i 4. zawierające wprowadzenie do oferowanych przez JavaScript funkcji typów danych oraz 5. i 6., w których przedstawiłem narzędzia programistyczne używane podczas pracy z TypeScriptem.

- Część II, „Praca z językiem TypeScript”, zawiera omówienie funkcji TypeScriptu związanych z produktywnością programisty, czyli m.in. typowanie statyczne. TypeScript oferuje wiele różnych funkcji związanych z typami — omówię je dokładnie i zaprezentuję ich działanie na przykładach.
- Część III, „Tworzenie aplikacji internetowych” — TypeScript nie jest używany w osamotnieniu, więc w trzeciej części dowiesz się, jak go używać do tworzenia aplikacji internetowych za pomocą trzech popularnych frameworków: Angular, React i Vue.js. Rozdziały tej części przedstawią funkcje TypeScriptu użyteczne w poszczególnych frameworkach, a także pokażą sposoby wykonywania zadań często pojawiających się podczas tworzenia aplikacji internetowych. Aby zapewnić podstawy dotyczące wymienionych wcześniej frameworków, pokażę również, jak utworzyć samodzielną aplikację internetową, której działanie nie będzie się opierało na żadnym frameworku.

Czy w książce znajdziesz wiele przykładów?

Ta książka jest wręcz *wypełniona* przykładami. Według mnie najlepszy sposób poznania języka TypeScript to analiza przykładów, których tutaj znajdziesz naprawdę wiele. W celu zmaksymalizowania liczby przykładów zdecydowałem się na użycie prostej konwencji pozwalającej uniknąć nieustannego podawania zawartości plików. Podczas pierwszego wykorzystania danego pliku w rozdziale prezentuję jego pełną zawartość, jak przedstawiłem na listingu 2.1. Podaję nazwę pliku oraz katalog, w którym powinien zostać utworzony. Po wprowadzeniu zmian w kodzie zmodyfikowane polecenia są sformatowane pogrubioną czcionką.

Listing 2.1. *Asercja nieznannej wartości w kodzie pliku `index.ts` znajdującego się w katalogu `src`*

```
function calculateTax(amount: number, format: boolean): string | number {
  const calcAmount = amount * 1.2;
  return format ? `${calcAmount.toFixed(2)} zł` : calcAmount;
}

let taxValue = calculateTax(100, false);

switch (typeof taxValue) {
  case "number":
    console.log(`Wartość typu number: ${taxValue.toFixed(2)}`);
    break;
  case "string":
    console.log(`Wartość typu string: ${taxValue.charAt(0)}`);
    break;
  default:
    let value: never = taxValue;
    console.log(`Nieoczekiwany typ dla wartości: ${value}`);
}

let newResult: unknown = calculateTax(200, false);
let myNumber: number = newResult as number;
console.log(`Wartość liczbowa: ${myNumber.toFixed(2)}`);
```

Ten listing pochodzi z rozdziału 7. i przedstawia zawartość pliku o nazwie *index.ts* utworzonego w katalogu *src*. W tym momencie nie przejmuj się zawartością listingu lub jego przeznaczeniem. Pamiętaj tylko, że taki listing przedstawia pełną zawartość pliku, a zmiany konieczne do wprowadzenia są oznaczone pogrubioną czcionką.

Niektóre pliki mogą być naprawdę długie, a omawiana przeze mnie funkcjonalność wymaga tylko niewielkiej zmiany. W takich przypadkach zamiast prezentować pełną zawartość pliku, używam wielokropka do wskazania jedynie fragmentu listingu, jak pokazałem na listingu 2.2.

Listing 2.2. Konfigurowanie narzędzi w pliku *package.json* w katalogu *reactapp*

```
...
"scripts": {
  "json": "json-server data.js -p 4600",
  "serve": "react-scripts start",
  "start": "npm-run-all -p serve json",
  "build": "react-scripts build",
  "test": "react-scripts test",
  "eject": "react-scripts eject"
},
...
```

Ten listing pochodzi z rozdziału 19. i pokazuje zestaw zmian koniecznych do wprowadzenia w pewnej części większego pliku. Gdy zobaczysz listing częściowy, możesz mieć pewność, że pozostała część pliku nie uległa zmianie, a zmodyfikowane zostały jedynie sekcje oznaczone pogrubioną czcionką.

W jeszcze innych sytuacjach konieczne jest wprowadzenie zmian w różnych fragmentach pliku, co utrudnia przedstawienie listingu częściowego. Wówczas będę pomijał niezmodyfikowany fragment listingu, jak pokazałem na listingu 2.3.

Listing 2.3. Stosowanie dekoratora w kodzie pliku *abstractDataSource.ts* w katalogu *src*

```
import { Product, Order } from "../entities";
import { minimumValue } from "../decorators";

export type ProductProp = keyof Product;

export abstract class AbstractDataSource {
  private _products: Product[];
  private _categories: Set<string>;
  public order: Order;
  public loading: Promise<void>;

  constructor() {
    this._products = [];
    this._categories = new Set<string>();
    this.order = new Order();
    this.loading = this.getData();
  }

  @minimumValue("price", 30)
  async getProducts(sortProp: ProductProp = "id",
    category? : string): Promise<Product[]> {
```

```

    await this.loading;
    return this.selectProducts(this._products, sortProp, category);
  }

  // Pozostałe metody zostały pominięte.
}

```

Ten listing pochodzi z rozdziału 16., zmiany również są przedstawione pogrubioną czcionką, a niezmienione fragmenty listingu zostały pominięte.

Gdzie znajdziesz przykładowe fragmenty kodu?

Wszystkie przykładowe projekty zostały umieszczone w archiwum dostępnym pod adresem <https://ftp.helion.pl/przyklady/troyaID.zip>. Archiwum plików jest dostępne bezpłatnie i zawiera wszystkie zasoby niezbędne do odtworzenia przykładów przedstawionych w książce bez konieczności ich samodzielnego wpisywania.

Co zrobić w przypadku problemów podczas wykonywania przykładów?

Przede wszystkim powróć na początek rozdziału i od nowa rozpocznij wykonywanie danego przykładu. Większość problemów jest spowodowana przez pominięcie któregoś kroku lub fragmentu listingu. Zwróć uwagę na zaznaczone fragmenty listingów, w których zostały wskazane zmiany, jakie należy wprowadzić.

Następnie zajrzyj do erraty. Książki informatyczne bywają skomplikowane i błędy są nieuniknione pomimo wysiłków autora i wydawcy. Sprawdź erratę pod kątem błędów i wykonaj zamieszczone w niej polecenia.

Jeżeli nadal będziesz mieć problemy, zajrzyj do projektu dla danego rozdziału. (Wszystkie przykładowe projekty zostały umieszczone w archiwum dostępnym pod adresem <https://ftp.helion.pl/przyklady/tys4o2.zip>). To pozwoli porównać tworzony projekt z moim. Wspomniane projekty utworzyłem, pracując nad poszczególnymi rozdziałami, więc powinny zawierać dokładnie te same pliki i tę samą treść, która znajduje się w Twoim projekcie.

Jeżeli wciąż nie będziesz w stanie otrzymać działającego projektu, możesz się ze mną skontaktować, pisząc na adres adam@adam-freeman.com. Dokładnie napisz, którą książkę czytasz i z którym przykładem masz problem. Pamiętaj, że otrzymuję naprawdę wiele wiadomości e-mail, więc może minąć nieco czasu, nim otrzymasz odpowiedź.

Co zrobić w sytuacji, gdy znajdę błąd w książce?

Wprawdzie wszelkie błędy możesz zgłaszać, pisząc do mnie na adres adam@adam-freeman.com, ale najpierw proszę o sprawdzenie erraty, którą znajdziesz w repozytorium GitHub przygotowanym dla książki i dostępnym pod adresem <https://github.com/Apress/essential-typescript-4>. Być może dany błąd został już wcześniej zgłoszony.

W repozytorium tym dodaję znalezione błędy, które prawdopodobnie będą dezorientowały czytelników (w szczególności są to problemy związane z przykładowymi fragmentami kodu). Dziękuję przy tym pierwszemu czytelnikowi zgłaszającemu dany błąd. Mniejsze błędy, najczęściej w tekście dotyczącym przykładów, zapisuję na oddzielnej liście, której później używam podczas przygotowywania nowego wydania danej książki.

Jak mogę skontaktować się z autorem?

Napisz do mnie na adres adam@adam-freeman.com. Minęło już kilka lat, odkąd zacząłem umieszczać swój adres e-mail w książkach. Na początku nie byłem pewien, czy to jest dobry pomysł, ale cieszę się, że to zrobiłem. Otrzymałem wiele wiadomości z całego świata, od czytelników dopiero wchodzących do świata programowania, a także od zajmujących się tym zawodowo. W większości przypadków te wiadomości były pozytywne, uprzejme i naprawdę miło było je otrzymać.

Staram się odpowiadać na e-maile, choć otrzymuję ich naprawdę wiele. Czasami jestem bardzo zajęty, zwłaszcza gdy kończę pracę nad kolejną książką. Zawsze staram się pomóc czytelnikom, którzy mają problemy z przykładami zamieszczonymi w książce. Jednak zanim się ze mną skontaktujesz w tej sprawie, spróbuj najpierw wykonać kroki przedstawione nieco wcześniej w tym rozdziale.

Wprawdzie bardzo lubię wiadomości od czytelników, ale otrzymuję sporo często powtarzających się pytań, na które odpowiedź zawsze brzmi „nie”. Bardzo mi przykro, ale nie utworzę kodu dla nowego startupu, nie pomogę Ci w pracy domowej z programowania, nie zaangażuję się w spór projektowy w Twoim zespole, a także nie nauczę Cię podstaw programowania.

Co zrobić, jeśli książka mi się podobała?

Napisz do mnie na adres adam@adam-freeman.com. Zawsze z przyjemnością czytam wiadomości od zadowolonych czytelników oraz doceniam czas poświęcony na ich napisanie i wysłanie. Pisanie książek informatycznych nie należy do łatwych zadań, a dzięki otrzymanym wiadomościom zachowuję motywację do dalszej pracy i podejmowania kolejnych wyzwań, które czasami wydają się niemożliwe do zrealizowania.

Co zrobić, jeśli książka mi się nie podobała?

Napisz do mnie na adres adam@adam-freeman.com, a ja spróbuję Ci pomóc. Pamiętaj, proszę, że mogę pomóc tylko wtedy, gdy dokładnie przedstawisz problem i wyjaśnisz, czego ode mnie oczekujesz. Może się okazać, że mój styl pisania nie trafi do Ciebie. W takich przypadkach rozwiązaniem jest zwrot książki i jej wymiana na inną. Bardzo chcę poznać powody Twojego niezadowolenia, choć po 25 latach pisania książek mam świadomość, że nie każdy czerpie radość z ich czytania.

Podsumowanie

W tym rozdziale wyjaśniłem, kiedy TypeScript będzie dobrym wyborem dla projektu. Przedstawiłem także zawartość i strukturę książki, wyjaśniłem, gdzie znajdziesz przykładowe fragmenty kodu, oraz podałem adres, za pomocą którego możesz się ze mną skontaktować, gdy napotkasz problemy podczas tworzenia projektów zaprezentowanych w książce. W następnym rozdziale przedstawię krótkie wprowadzenie do systemu typów JavaScriptu, który stanowi podstawę dla funkcji oferowanych przez język TypeScript.

ROZDZIAŁ 3.



Wprowadzenie do języka JavaScript — część I

Efektywne programowanie w języku TypeScript wymaga poznania sposobów, w jakie JavaScript radzi sobie z typami danych. Może to być rozczarowujące dla programistów dopiero poznających język TypeScript, którzy będą uważać JavaScript za dezorientujący. Jednak poznanie JavaScriptu pomaga w łatwiejszym uczeniu się TypeScriptu i dostarcza wręcz nieocenione informacje o tym, co oferuje i jak działa TypeScript. W tym rozdziale zamierzam wprowadzić podstawowe funkcje JavaScriptu, a te bardziej zaawansowane poznasz w rozdziale 4.

Przygotowanie projektu

Aby przygotować się do utworzenia projektu omawianego w rozdziale, w dogodnym miejscu utwórz katalog o nazwie *primer*. Następnie przejdź do powłoki, wejdź do katalogu *primer* i wydaj w nim polecenie przedstawione na listingu 3.1.

■ **Wskazówka** Przykładowy projekt dla tego rozdziału, a także projekty dla pozostałych rozdziałów książki zostały umieszczone w archiwum dostępnym pod adresem <https://ftp.helion.pl/przyklady/tys4o2.zip>.

Listing 3.1. Przygotowywanie katalogu projektu

```
$ npm init --yes
```

Aby zainstalować pakiet automatycznie uruchamiający plik JavaScriptu po zmianie jego zawartości, z poziomu katalogu *primer* wydaj polecenie przedstawione na listingu 3.2.

Listing 3.2. Instalowanie niezbędnego pakietu

```
$ npm install nodemon@2.0.7
```

Pakiet o nazwie nodemon zostanie pobrany i zainstalowany. Po zakończeniu instalacji w katalogu *primer* utwórz nowy plik o nazwie *index.js* i zawartości przedstawionej na listingu 3.3.

Listing 3.3. Zawartość pliku *index.js* w katalogu *primer*

```
let hatPrice = 100;
console.log(`Cena czapki: ${hatPrice}`);
```

Wyдай polecenie przedstawione na listingu 3.4, aby rozpocząć monitorowanie zawartości pliku JavaScriptu.

Listing 3.4. Uruchamianie monitora zawartości pliku JavaScriptu

```
$ npx nodemon index.js
```

Pakiet nodemon spowoduje wykonanie poleceń zdefiniowanych w pliku *index.js* i wygenerowanie następujących danych wyjściowych:

```
[nodemon] 2.0.7
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: *.*
[nodemon] starting `node index.js`
Cena czapki: 100
[nodemon] clean exit - waiting for changes before restart
```

Pogrubioną czcionką oznaczyłem tę część danych wyjściowych, która pochodzi z pliku *index.js*. Aby mieć pewność o prawidłowym wykrywaniu zmian, zawartość pliku *index.js* zmodyfikuj w sposób przedstawiony na listingu 3.5.

Listing 3.5. Wprowadzanie zmian w kodzie pliku *index.js* w katalogu *primer*

```
let hatPrice = 100;
console.log(`Cena czapki: ${hatPrice}`);
let bootsPrice = "100";
console.log(`Cena butów: ${bootsPrice}`);
```

Po zapisaniu zmian pakiet nodemon powinien wykryć modyfikację pliku *index.js* i wykonać znajdujący się w nim kod. Na listingu 3.5 przedstawiłem wygenerowane dane wyjściowe bez tych pochodzących z pakietu nodemon.

```
Cena czapki: 100
Cena butów: 100
```

Zagmatwany JavaScript

JavaScript ma wiele funkcji podobnych do istniejących w innych językach programowania, a programiści mają tendencję do tworzenia kodu przypominającego polecenia przedstawione na listingu 3.5. Nawet jeśli dopiero zaczynasz programowanie z użyciem JavaScriptu, polecenia z listingu 3.5 powinny być Ci znajome.

Elementami konstrukcyjnymi kodu JavaScriptu są polecenia wykonywane w kolejności ich zdefiniowania. Słowo kluczowe `let` jest używane do definiowania zmiennej (w przeciwieństwie do słowa kluczowego `const` pozwalającego na zdefiniowanie stałej), a po nim znajduje się nazwa tworzonej zmiennej. Wartość zmiennej jest definiowana za pomocą operatora przypisania (znak równości), po którym umieszcza się żądaną wartość.

JavaScript oferuje wbudowane obiekty przeznaczone do przeprowadzania zadań takich jak wyświetlanie ciągów tekstowych w powłoce za pomocą metody `console.log()`. Ciąg tekstowy może być zdefiniowany na różne sposoby: jako literał, za pomocą znaków apostrofu lub cudzysłowu, na podstawie szablonu ciągu tekstowego, za pomocą znaków odwrotnego apostrofu oraz przez wstawienie wyrażenia do szablonu przy użyciu znaku dolara i nawiasu klamrowego.

Jednak w pewnym momencie pojawi się nieoczekiwany wynik. Może to być źródłem zakłopotania związanego ze sposobem, w jaki JavaScript radzi sobie z typami. Na listingu 3.6 przedstawiłem typowy problem.

Listing 3.6. Dodawanie poleceń w kodzie pliku `index.ts` w katalogu `primer`

```
let hatPrice = 100;
console.log(`Cena czapki: ${hatPrice}`);
let bootsPrice = "100";
console.log(`Cena butów: ${bootsPrice}`);

if (hatPrice == bootsPrice) {
    console.log("Ceny są takie same");
} else {
    console.log("Ceny są różne");
}

let totalPrice = hatPrice + bootsPrice;
console.log(`Kwota całkowita: ${totalPrice}`);
```

Nowe polecenia mają za zadanie porównać wartości zmiennych `hatPrice` i `bootsPrice` oraz przypisać ich sumę nowej zmiennej o nazwie `totalPrice`. Metoda `console.log()` jest używana do wyświetlania komunikatów w powłoce, a po wykonaniu omawianego kodu spowoduje wygenerowanie następujących danych:

```
Cena czapki: 100
Cena butów: 100
Ceny są takie same
Kwota całkowita: 100100
```

Większość programistów zauważy, że wartość `hatPrice` została wyrażona w postaci liczby, a wartość `bootsPrice` w postaci znaków ciągu tekstowego ujętego w znaki cudzysłowu. W większości języków programowania przeprowadzanie operacji na różnych typach danych prowadzi do błędu. Pod tym względem JavaScript jest inny — porównanie wartości liczbowej i tekstowej kończy się sukcesem, natomiast ich suma powstaje na skutek konkatenacji wartości. Zrozumienie wyniku działania kodu przedstawionego na listingu 3.6 — i powodu wygenerowania takich danych — ujawnia szczegóły związane z podejściem JavaScriptu do danych i jednocześnie pokazuje, dlaczego język TypeScript może być bardzo użyteczny.

Typy języka JavaScript

Można odnieść wrażenie, że JavaScript nie zawiera typów danych bądź są one używane niespójnie, ale to nieprawda. JavaScript po prostu działa odmiennie niż większość popularnych języków programowania i tylko wydaje się, że działa niespójnie, jeśli nie wiesz, czego się spodziewać. Podstawą dla języka JavaScript jest zestaw wbudowanych typów, które wymienilem w tabeli 3.1.

Tabela 3.1. Wbudowane typy w JavaScriptcie

Nazwa	Opis
numer	Ten typ jest używany do przedstawiania wartości liczbowej. W przeciwieństwie do innych języków programowania JavaScript nie odróżnia wartości całkowitej od zmiennoprzecinkowej i dlatego obydwie te wartości mogą być używane w tym typie
string	Ten typ jest używany do przedstawiania danych tekstowych
boolean	Ten typ wykorzystuje wartości <code>true</code> i <code>false</code>
symbol	Ten typ wykorzystuje unikatowe wartości stałe, takie jak klucze w kolekcji
null	Ten typ może mieć przypisaną tylko wartość <code>null</code> i służy do wskazania nieistniejącego lub nieprawidłowego odwołania
undefined	Ten typ jest używany w przypadku, gdy zmienna została zdefiniowana, ale nie przypisano jej wartości
object	Ten typ służy do przedstawiania wartości złożonych, utworzonych z poszczególnych właściwości i wartości

Sześć pierwszych typów wymienionych w tabeli 3.1 zalicza się do podstawowych typów danych w JavaScriptcie. Są one zawsze dostępne, a każda wartość w aplikacji JavaScriptu jest typu podstawowego lub składa się z typów podstawowych. Siódmym typem jest `object` i służy do przedstawiania obiektów.

Praca z podstawowymi typami danych

Jeżeli raz jeszcze spojrzysz na listing 3.6, to zauważysz, że w kodzie nie ma żadnych deklaracji typów. W innych językach programowania konieczne jest zadeklarowanie typu danych zmiennej przed jej użyciem. Przedstawiony tutaj fragment kodu pochodzi z jednej z moich książek dotyczących programowania w języku C#:

```
...
string name = "Adam";
...
```

To polecenie określa, że typem danych zmiennej `name` jest `string`, i przypisuje tej zmiennej wartość `Adam`. W języku JavaScript to *wartość* ma typ, a nie zmienna. W celu zdefiniowania zmiennej przechowującej ciąg tekstowy należy przypisać wartość w postaci ciągu tekstowego, jak pokazałem na listingu 3.7.

Listing 3.7. Utworzenie w kodzie pliku *index.js* w katalogu *primer* zmiennej w postaci ciągu tekstowego

```
let hatPrice = 100;
console.log(`Cena czapki: ${hatPrice}`);
let bootsPrice = "100";
console.log(`Cena butów: ${bootsPrice}`);

if (hatPrice == bootsPrice) {
  console.log("Ceny są takie same");
} else {
  console.log("Ceny są różne");
}

let totalPrice = hatPrice + bootsPrice;
console.log(`Kwota całkowita: ${totalPrice}`);

let myVariable = "Adam";
```

Środowisko uruchomieniowe ma tylko ustalić, który z typów podanych w tabeli 3.1 powinien zostać użyty dla wartości przypisanej zmiennej *myVariable*. Mały zestaw typów obsługiwanych przez JavaScript znacznie upraszcza ten proces, a środowisko uruchomieniowe, że każda wartość ujęta w znaki cytowania jest ciągiem tekstowym (string). Do potwierdzenia typu wartości można użyć słowa kluczowego *typeof*, jak pokazałem na listingu 3.8.

Listing 3.8. Pobieranie typu wartości w kodzie pliku *index.js* w katalogu *primer*

```
let hatPrice = 100;
console.log(`Cena czapki: ${hatPrice}`);
let bootsPrice = "100";
console.log(`Cena butów: ${bootsPrice}`);

if (hatPrice == bootsPrice) {
  console.log("Ceny są takie same");
} else {
  console.log("Ceny są różne");
}

let totalPrice = hatPrice + bootsPrice;
console.log(`Kwota całkowita: ${totalPrice}`);

let myVariable = "Adam";
console.log(`Typ: ${typeof myVariable}`);
```

Słowo kluczowe *typeof* identyfikuje typ wartości, a następnie powoduje wygenerowanie następujących danych wyjściowych po uruchomieniu omawianego fragmentu kodu:

```
Cena czapki: 100
Cena butów: 100
Ceny są takie same
Kwota całkowita: 100100
Typ: string
```

Na listingu 3.9 pokazałem przypisanie nowej wartości zmiennej *myVariable* i ponowne wyświetlenie typu.

Listing 3.9. Przypisywanie nowej wartości w kodzie pliku *index.js* w katalogu *primer*

```

let hatPrice = 100;
console.log(`Cena czapki: ${hatPrice}`);
let bootsPrice = "100";
console.log(`Cena butów: ${bootsPrice}`);

if (hatPrice == bootsPrice) {
  console.log("Ceny są takie same");
} else {
  console.log("Ceny są różne");
}

let totalPrice = hatPrice + bootsPrice;
console.log(`Kwota całkowita: ${totalPrice}`);

let myVariable = "Adam";
console.log(`Typ: ${typeof myVariable}`);
myVariable = 100;
console.log(`Typ: ${typeof myVariable}`);

```

Po zapisaniu wprowadzonych zmian ten kod spowoduje wygenerowanie następujących danych wyjściowych:

```

Cena czapki: 100
Cena butów: 100
Ceny są takie same
Kwota całkowita: 100100
Typ: string
Typ: numer

```

Zmiana wartości przypisanej zmiennej powoduje zmianę typu podawanego przez słowo kluczowe `typeof`, ponieważ to wartość ma zdefiniowany typ. W omawianym przykładzie typem wartości początkowo przypisanej zmiennej `myVariable` był `string`, a następnie, po przypisaniu jej wartości `100`, typ zmienił się na `number`. Takie dynamiczne podejście do typów znacznie ułatwia pracę w przypadku ograniczonej liczby typów obsługiwanych przez JavaScript, co zdecydowanie ułatwia ustalenie, który z wbudowanych typów jest używany. Na przykład wszystkie liczby są przedstawiane za pomocą typu `number`, co byłoby niemożliwe w przypadku bardziej złożonego zestawu typów.

Dziwactwa typu null

Gdy słowo kluczowe `typeof` zostanie użyte z wartością `null`, wynikiem będzie typ `object`. Jest to odwieczne zachowanie JavaScriptu od jego początków i nie zostało zmienione, ponieważ ogromna ilość kodu została utworzona w sposób oczekujący właśnie takiego zachowania.

Koercja typu

W przypadku zastosowania operatora z wartościami różnych typów środowisko uruchomieniowe JavaScriptu skonwertuje wartość jednego typu na jej odpowiednik w drugim typie. Ten proces jest określany mianem *koercji typu*. Ta funkcjonalność koercji typu — nazywana również *konwersją typu* — powoduje otrzymanie niespójnych wyników po wykonaniu kodu przedstawionego na listingu 3.6, choć jak się wkrótce dowiesz po zrozumieniu sposobu działania koercji, ten wynik wcale nie jest niespójny. W kodzie na listingu 3.6 mamy dwa miejsca, w których przeprowadzana jest koercja typu.

```
...
let hatPrice = 100;
console.log(`Cena czapki: ${hatPrice}`);
let bootsPrice = "100";
console.log(`Cena butów: ${bootsPrice}`);

if (hatPrice == bootsPrice) {
  ...
}
```

Dwa znaki równości oznaczają porównanie za pomocą koercji typu, więc JavaScript spróbuje skonwertować wartości w celu wygenerowania użytecznego wyniku. Ta operacja jest w JavaScriptcie znana pod nazwą *abstrakcji sprawdzenia równości* i gdy wartość typu number jest porównywana z wartością typu string, następuje skonwertowanie wartości typu string na typ number, a następnie przeprowadzenie porównania. Dlatego też podczas porównywania wartości 100 typu number z wartością 100 typu string druga z wymienionych jest konwertowana na typ number i to jest powód, dla którego wyrażenie `if` przyjmuje wartość `true`.

■ **Wskazówka** W specyfikacji JavaScriptu zamieszczonej na stronie <https://www.ecma-international.org/ecma-262/7.0/#sec-abstract-equality-comparison> istnieje możliwość zapoznania się z sekwencją kroków wykonywanych przez JavaScript podczas wspomnianej operacji abstrakcji sprawdzenia równości. Ta specyfikacja jest doskonale przygotowana i zaskakująco interesująca. Zanim jednak poświęcisz dzień na zagłębienie się w szczegóły implementacji, powinieneś wiedzieć, że TypeScript ogranicza używanie niektórych z rzadziej stosowanych i egzotycznych funkcji JavaScriptu.

Po raz drugi koercja została użyta w kodzie na listingu 3.6 podczas obliczania kwoty całkowitej:

```
...
let totalPrice = hatPrice + bootsPrice;
...
```

Gdy próbujesz użyć operatora `+` z wartościami typu `number` i `string`, wówczas jedna z nich zostaje skonwertowana. Zaskakujące może być to, że konwersja nie jest taka sama jak w przypadku operacji porównania. Jeżeli którakolwiek z wartości jest typu `string`, wówczas ta druga również będzie skonwertowana na postać ciągu tekstowego, a następnie obie wartości będą poddane konkatenaacji. Dlatego też gdy wartość 100 typu `number` zostaje dodana do wartości 100 typu `string`, następuje konwersja wartości typu `number` na `string` i wygenerowanie wyniku w postaci ciągu tekstowego `100100`.

Unikanie niechcianej koercji typu

Wprawdzie koercja typu może być użyteczną funkcją, ale zdobyła złą sławę tylko dlatego, że jest stosowana mimowolnie — to nie jest trudne, zwłaszcza gdy przetwarzane typy ulegają zmianie po przypisaniu nowych wartości. Jak się dowiesz w dalszej części książki, TypeScript oferuje funkcje pomagające w zarządzaniu niechcianą koercją i gwarantujące jej stosowanie tylko wtedy, gdy będzie wyraźnie żądana. Jednak JavaScript oferuje także funkcje pomagające w unikaniu koercji, co zaprezentowałem na listingu 3.10.

Listing 3.10. Zapobieganie koercji w kodzie pliku *index.js* w katalogu *primer*

```
let hatPrice = 100;
console.log(`Cena czapki: ${hatPrice}`);
let bootsPrice = "100";
console.log(`Cena butów: ${bootsPrice}`);

if (hatPrice === bootsPrice) {
  console.log("Ceny są takie same");
} else {
  console.log("Ceny są różne");
}

let totalPrice = Number(hatPrice) + Number(bootsPrice);
console.log(`Kwota całkowita: ${totalPrice}`);

let myVariable = "Adam";
console.log(`Typ: ${typeof myVariable}`);
myVariable = 100;
console.log(`Typ: ${typeof myVariable}`);
```

Dwa znaki równości przeprowadzają operację porównania, w trakcie której jest stosowana koercja. Z kolei trzy znaki równości powodują zastosowanie ścisłego porównania, którego wynikiem będzie wartość `true` tylko wtedy, gdy porównywane wartości są tego samego typu i mają tę samą wartość.

Aby uniknąć konkatencji ciągu tekstowego, wartości mogą być wyraźnie skonwertowane na postać liczb jeszcze przed zastosowaniem operatora dodawania — do tego celu służy wbudowana funkcja `Number()`. Po konwersji wartości na liczby przeprowadzane jest arytmetyczne dodawanie, a kod przedstawiony na listingu 3.10 powoduje wygenerowanie następujących danych wyjściowych:

```
Cena czapki: 100
Cena butów: 100
Ceny są różne
Kwota całkowita: 200
Typ: string
Typ: numer
```

Wartość jawnie stosowanej koercji typu

Koercja typu może być użyteczną funkcją, gdy jest stosowana jawnie. Jednym z użytecznych aspektów koercji jest sposób, w jaki wartości są poddawane koercji na typ `boolean` przez operator logiczny lub (`||`). Wartość typu `null` lub `undefined` zostaje skonwertowana na `false`, co powoduje, że koercja staje się efektywnym narzędziem zapewniającym wartości awaryjne, jak pokazałem na listingu 3.11.

Listing 3.11. Obsługa wartości null w kodzie pliku index.js w katalogu primer

```
let hatPrice = 100;
console.log(`Cena czapki: ${hatPrice}`);
let bootsPrice = "100";
console.log(`Cena butów: ${bootsPrice}`);

if (hatPrice === bootsPrice) {
  console.log("Ceny są takie same");
} else {
  console.log("Ceny są różne");
}

let totalPrice = Number(hatPrice) + Number(bootsPrice);
console.log(`Kwota całkowita: ${totalPrice}`);

let myVariable = "Adam";
console.log(`Typ: ${typeof myVariable}`);
myVariable = 100;
console.log(`Typ: ${typeof myVariable}`);

let firstCity;
let secondCity = firstCity || "Londyn";
console.log(`Miasto: ${ secondCity }`);
```

Wartość zmiennej o nazwie secondCity zostaje zdefiniowana za pomocą wyrażenia sprawdzającego wartość firstCity: jeżeli firstCity będzie skonwertowana na wartość boolowską true, wówczas wartością secondCity będzie wartość firstCity.

Typ undefined jest używany podczas definiowania zmiennych, którym jeszcze nie jest przypisywana wartość. Z taką sytuacją mamy do czynienia w przypadku zmiennej firstCity, a użycie operatora || gwarantuje zastosowanie wartości awaryjnej dla secondCity, gdy wartością firstCity jest undefined lub null.

Operator null coalescing

Problem z operatorem logicznym OR polega na tym, że wartości null i undefined nie są konwertowane na false, co może doprowadzić do wygenerowania nieoczekiwanych wyników. Przykład takiej sytuacji pokazałem na listingu 3.12.

Listing 3.12. Niezamierzony efekt działania koercji typu w kodzie pliku index.js w katalogu primer

```
let hatPrice = 100;
console.log(`Cena czapki: ${hatPrice}`);
let bootsPrice = "100";
console.log(`Cena butów: ${bootsPrice}`);

let taxRate; // Stawka podatkowa nie została zdefiniowana.
console.log(`Stawka podatkowa: ${taxRate || 10}%`);
taxRate = 0; // Wysokość stawki podatkowej wynosi 0.
console.log(`Stawka podatkowa: ${taxRate || 10}%`);
```


Oprócz wartości `null` i `undefined` koercję na `false` operator `OR` stosuje również dla wartości liczbowej `0` (zero), pustego ciągu tekstowego (`""`) i wartości specjalne `NaN`. Wymienione wartości, a także `false`, są w JavaScriptcie określane mianem „falsy” i mogą wywoływać dużą dezorientację u programistów. W kodzie zamieszczonym na listingu 3.12 operator `OR` użył wartości awaryjnej, gdy zmiennej `taxRate` została przypisana wartość zero. W efekcie otrzymujemy następujące dane wyjściowe:

```
Cena czapki: 100
Cena butów: 100
Stawka podatkowa: 10%
Stawka podatkowa: 10%
```

Kod nie potrafi odróżnić wartości nieprzypisanej i zerowej, co może stanowić problem, gdy wymagane jest użycie wartości. W omawianym przykładzie nie ma możliwości określenia stawki podatkowej w wysokości `0`, pomimo że to jest poprawna wartość dla tego typu danych. Aby rozwiązać ten problem, JavaScript obsługuje operator *null coalescing*, `??`, odpowiedzialny za koercję jedynie wartości `undefined` i `null`. Nie jest on stosowany z żadnymi innymi wartościami `false`. Przykład użycia operatora `??` zamieściłem na listingu 3.13.

Listing 3.13. Przykład użycia operatora *null coalescing* w kodzie pliku `index.js` w katalogu `primer`

```
let hatPrice = 100;
console.log(`Cena czapki: ${hatPrice}`);
let bootsPrice = "100";
console.log(`Cena butów: ${bootsPrice}`);

let taxRate; // Stawka podatkowa nie została zdefiniowana.
console.log(`Stawka podatkowa: ${taxRate ?? 10}%`);
taxRate = 0; // Wysokość stawki podatkowej wynosi 0.
console.log(`Stawka podatkowa: ${taxRate ?? 10}%`);
```

W pierwszym poleceniu zostanie użyta wartość rezerwowa, ponieważ wartością `taxRate` jest `undefined`. Z kolei w drugim poleceniu wartość rezerwowa nie będzie wykorzystana, gdyż operator `??` nie jest stosowany razem z zerem. Omawiany kod spowoduje wygenerowanie następujących danych wyjściowych:

```
Cena czapki: 100
Cena butów: 100
Stopa podatkowa: 10%
Stawka podatkowa: 0%
```

Praca z funkcją

Elastyczne podejście JavaScriptu do typów ma odzwierciedlenie także w innych częściach języka, np. funkcjach. Na listingu 3.14 pokazałem dodanie funkcji do przykładowego pliku JavaScriptu. W celu zachowania zwięzłości usunąłem część wcześniej dodanych poleceń.

Listing 3.14. Definiowanie funkcji w kodzie pliku *index.js* w katalogu *primer*

```
let hatPrice = 100;
console.log(`Cena czapki: ${hatPrice}`);
let bootsPrice = "100";
console.log(`Cena butów: ${bootsPrice}`);

function sumPrices(first, second, third) {
    return first + second + third;
}

let totalPrice = sumPrices(hatPrice, bootsPrice);
console.log(`Kwota całkowita: ${totalPrice}`);
```

Typy parametrów funkcji są określane na podstawie wartości użytych do wywołania funkcji. Funkcja może zakładać otrzymanie wartości np. typu `number`, choć nic nie uniemożliwia wywołania jej z argumentami typów `string`, `boolean` lub `object`. Wygenerowany będzie nieoczekiwany wynik, jeśli funkcja nie weryfikuje założeń z powodu przeprowadzanej przez środowisko uruchomieniowe JavaScriptu koercji wartości lub na skutek użycia funkcjonalności charakterystycznej tylko dla jednego typu.

Przedstawiona na listingu 3.14 funkcja `sumPrices()` wykorzystuje operator `+` w celu obliczenia sumy parametrów typu `number`, ale jedna z wartości użytych do wywołania funkcji ma postać ciągu tekstowego. Jak już wcześniej wspomniałem, operator `+` użyty z wartością typu `string` przeprowadza konkatencję. Dlatego też omawiany tutaj kod powoduje wygenerowanie następujących danych wyjściowych:

```
Cena czapki: 100
Cena butów: 100
Kwota całkowita: 100100undefined
```

JavaScript nie wymusza dopasowania między parametrami definiowanymi przez funkcję, a także nie sprawdza ich liczby. Wartością każdego parametru, dla którego nie została podana wartość, będzie `undefined`. W omawianym przykładzie brakuje wartości parametru `third`, więc wartość `undefined` zostaje skonwertowana na postać ciągu tekstowego i dołączona w danych wyjściowych utworzonych za pomocą konkatencji.

```
Kwota całkowita: 100100undefined
```

Praca z wynikiem działania funkcji

Różnice między typami JavaScriptu a typami w innych językach programowania są potęgowane przez funkcje. W efekcie typ JavaScriptu ma tę cechę, że argumenty użyte w wywołaniu funkcji mogą określić typ wyniku, jak pokazałem na listingu 3.15.

Listing 3.15. Przykład wywołania funkcji w kodzie pliku *index.js* w katalogu *primer*

```
let hatPrice = 100;
console.log(`Cena czapki: ${hatPrice}`);
let bootsPrice = "100";
console.log(`Cena butów: ${bootsPrice}`);
```

```
function sumPrices(first, second, third) {
    return first + second + third;
}

let totalPrice = sumPrices(hatPrice, bootsPrice);
console.log(`Wartość całkowita: ${totalPrice} ${typeof totalPrice}`);

totalPrice = sumPrices(100, 200, 300);
console.log(`Wartość całkowita: ${totalPrice} ${typeof totalPrice}`);

totalPrice = sumPrices(100, 200);
console.log(`Wartość całkowita: ${totalPrice} ${typeof totalPrice}`);
```

Wartość zmiennej `totalPrice` jest definiowana trzykrotnie przez wywołanie funkcji `sumPrices()`. Po każdym wywołaniu słowo kluczowe `typeof` pozwala na ustalenie typu wartości zwrótej funkcji. Kod przedstawiony na listingu 3.15 powoduje wygenerowanie następujących danych wyjściowych:

```
Cena czapki: 100
Cena butów: 100
Wartość całkowita: 100100undefined string
Wartość całkowita: 600 number
Wartość całkowita: NaN number
```

Pierwsze wywołanie funkcji `sumPrices()` zawiera argument typu `string`, co powoduje skonwertowanie wszystkich argumentów na postać wartości typu `string`, które następnie są poddawane konkatencji. W efekcie funkcja zwraca wartość w postaci ciągu tekstowego `100100undefined`.

Drugie wywołanie funkcji używa trzech wartości typu `number`, które po dodaniu do siebie dają liczbę 600. Ostatnie wywołanie używa argumentów typu `number`, ale jednocześnie nie zawiera wartości dla trzeciego parametru, który w ten sposób otrzymuje wartość `undefined`. JavaScript przeprowadza koalescencję wartości `undefined` na specjalną wartość `NaN` typu `number` (ang. *not a number*). Wynikiem dodawania zawierającego element `NaN` jest `NaN`. Dokładnie taki jest typ wyniku, choć ta wartość nie jest użyteczna i mało prawdopodobne, aby była oczekiwana.

Unikanie problemów z błędnym dopasowaniem argumentu

Wprawdzie wyniki otrzymane w poprzednim punkcie mogą wprawiać w zakłopotanie, ale są zgodne z regułami przedstawionymi w specyfikacji JavaScriptu. Problem nie polega na nieprzewidywalności JavaScriptu, ale na tym, że takie podejście jest odmienne od stosowanego w innych językach programowania.

W tej sekcji zaprezentuję oferowane przez JavaScript rozwiązania, które mogą być używane w celu uniknięcia przedstawionych wcześniej problemów. Pierwsze to wartości domyślne parametrów używane, gdy funkcja zostanie wywołana bez danego parametru, jak pokazałem na listingu 3.16.

Listing 3.16. *Używanie wartości domyślnych parametru w kodzie pliku `index.js` w katalogu `primer`*

```
let hatPrice = 100;
console.log(`Cena czapki: ${hatPrice}`);
let bootsPrice = "100";
```

```
console.log(`Cena butów: ${bootsPrice}`);

function sumPrices(first, second, third = 0) {
  return first + second + third;
}

let totalPrice = sumPrices(hatPrice, bootsPrice);
console.log(`Wartość całkowita: ${totalPrice} ${typeof totalPrice}`);

totalPrice = sumPrices(100, 200, 300);
console.log(`Wartość całkowita: ${totalPrice} ${typeof totalPrice}`);

totalPrice = sumPrices(100, 200);
console.log(`Wartość całkowita: ${totalPrice} ${typeof totalPrice}`);
```

Po nazwie parametru `third` znajduje się znak równości i wartość, która powinna zostać użyta w sytuacji, gdy funkcja będzie wywołana bez odpowiadającej mu wartości. Dlatego też polecenie wywołujące funkcję `sumPrices()` tylko z dwoma parametrami nie będzie już generowało wyniku typu `NaN`, jak pokazałem w przykładowych danych wyjściowych.

```
Cena czapki: 100
Cena butów: 100
Wartość całkowita: 1001000 string
Wartość całkowita: 600 number
Wartość całkowita: 300 number
```

Znacznie elastyczniejsze podejście polega na wykorzystaniu parametru resztowego, który jest poprzedzony prefiksem w postaci trzech kropek i musi być ostatnim parametrem definiowanym przez funkcję. Przykład jego użycia zaprezentowałem na listingu 3.17.

Listing 3.17. *Używanie parametru resztowego w kodzie pliku `index.js` w katalogu `primer`*

```
let hatPrice = 100;
console.log(`Cena czapki: ${hatPrice}`);
let bootsPrice = "100";
console.log(`Cena butów: ${bootsPrice}`);

function sumPrices(...numbers) {
  return numbers.reduce(function(total, val) {
    return total + val
  }, 0);
}

let totalPrice = sumPrices(hatPrice, bootsPrice);
console.log(`Wartość całkowita: ${totalPrice} ${typeof totalPrice}`);

totalPrice = sumPrices(100, 200, 300);
console.log(`Wartość całkowita: ${totalPrice} ${typeof totalPrice}`);

totalPrice = sumPrices(100, 200);
console.log(`Wartość całkowita: ${totalPrice} ${typeof totalPrice}`);
```

Parametr resztowy to tablica zawierająca wszystkie argumenty, dla których nie zostały zdefiniowane parametry. Funkcja w kodzie na listingu 3.17 definiuje tylko jeden parametr resztowy, co oznacza, że jej wartością jest tablica zawierająca wszystkie argumenty użyte do wywołania funkcji. Zawartość tablicy jest sumowana za pomocą wbudowanej metody tablicy o nazwie `reduce()`. Tablice JavaScriptu dokładniej omówię w dalszej części rozdziału. Metoda `reduce()` jest używana do wywołania funkcji dla każdego obiektu tablicy i wygenerowania pojedynczej wartości zwrotnej. Takie podejście gwarantuje, że liczba argumentów nie będzie miała wpływu na wynik, choć funkcja wywoływana przez metodę `reduce()` używa operatora dodawania, co oznacza, że wartości typu `string` nadal będą poddawane konkatenaacji. Kod na omawianym listingu powoduje wygenerowanie następujących danych wyjściowych:

```
Cena czapki: 100
Cena butów: 100
Wartość całkowita: 100100 string
Wartość całkowita: 600 number
Wartość całkowita: 300 number
```

Aby mieć pewność, że funkcja generuje użyteczną sumę wartości jej parametrów, można je skonwertować na liczby i poprzez filtrowanie pozbyć się wszystkich wartości `NaN`. Przykład takiego rozwiązania przedstawiłem na listingu 3.18.

Listing 3.18. Konwertowanie i filtrowanie wartości parametru przez kod pliku `index.js` w katalogu `primer`

```
let hatPrice = 100;
console.log(`Cena czapki: ${hatPrice}`);
let bootsPrice = "100";
console.log(`Cena butów: ${bootsPrice}`);

function sumPrices(...numbers) {
  return numbers.reduce(function(total, val) {
    return total + (Number.isNaN(Number(val)) ? 0 : Number(val));
  }, 0);
}

let totalPrice = sumPrices(hatPrice, bootsPrice);
console.log(`Wartość całkowita: ${totalPrice} ${typeof totalPrice}`);

totalPrice = sumPrices(100, 200, 300);
console.log(`Wartość całkowita: ${totalPrice} ${typeof totalPrice}`);

totalPrice = sumPrices(100, 200, undefined, false, "Witaj");
console.log(`Wartość całkowita: ${totalPrice} ${typeof totalPrice}`);
```

Metoda `Number.isNaN()` jest używana do sprawdzenia, czy wartość to `NaN`. Kod przedstawiony na listingu 3.18 jawnie konwertuje wszystkie parametry na typ `number` i zastępuje zerem każdy, którego typ to `NaN`. Przetworzone będą tylko te wartości parametrów, które mogą być uznane za liczby. Natomiast argumenty typu `undefined`, `boolean` i `string` znajdujące się w ostatecznym wywołaniu funkcji nie będą miały wpływu na wynik jej działania.

```
Cena czapki: 100
Cena butów: 100
Wartość całkowita: 200 number
Wartość całkowita: 600 number
Wartość całkowita: 300 number
```

Używanie funkcji strzałki

Funkcje strzałki — nazywane również *funkcjami grubej strzałki* lub *wyrażeniami lambda* — stanowią alternatywny sposób na zwięźle zdefiniowanie funkcji i są często używane do definiowania funkcji będących argumentami innych funkcji. Na listingu 3.19 pokazałem zmodyfikowaną wersję kodu, w której funkcja standardowa użyta z metodą `reduce()` została zastąpiona przez funkcję strzałki.

Listing 3.19. *Używanie funkcji strzałki w kodzie pliku `index.js` w katalogu `primer`*

```
let hatPrice = 100;
console.log(`Cena czapki: ${hatPrice}`);
let bootsPrice = "100";
console.log(`Cena butów: ${bootsPrice}`);

function sumPrices(...numbers) {
    return numbers.reduce((total, val) =>
        total + (Number.isNaN(Number(val)) ? 0 : Number(val)));
}

let totalPrice = sumPrices(hatPrice, bootsPrice);
console.log(`Wartość całkowita: ${totalPrice} ${typeof totalPrice}`);

totalPrice = sumPrices(100, 200, 300);
console.log(`Wartość całkowita: ${totalPrice} ${typeof totalPrice}`);

totalPrice = sumPrices(100, 200, undefined, false, "Witaj");
console.log(`Wartość całkowita: ${totalPrice} ${typeof totalPrice}`);
```

Funkcja strzałki składa się z trzech elementów: parametrów wejściowych, znaków równości i większości (tzw. strzałki) oraz wartości wynikowej. Słowo kluczowe `return` i nawias klamrowy są wymagane wyłącznie wtedy, gdy funkcja strzałki ma wykonać więcej niż tylko jedno polecenie.

Funkcja strzałki może być używana wszędzie tam, gdzie wymagana jest zwykła funkcja. Wybór między tymi funkcjami to tylko kwestia preferencji, z wyjątkiem sytuacji, którą wyjaśnię w sekcji zatytułowanej „Słowo kluczowe `this`” w dalszej części rozdziału. Na listingu 3.20 przedstawiłem zmodyfikowaną wersję funkcji `sumPrices()`, która teraz stosuje składnię funkcji strzałki.

Listing 3.20. *Zastępowanie funkcji w kodzie pliku `index.js` w katalogu `primer`*

```
let hatPrice = 100;
console.log(`Cena czapki: ${hatPrice}`);
let bootsPrice = "100";
console.log(`Cena butów: ${bootsPrice}`);

let sumPrices = (...numbers) => numbers.reduce((total, val) =>
```

```

    total + (Number.isNaN(Number(val)) ? 0 : Number(val)));

let totalPrice = sumPrices(hatPrice, bootsPrice);
console.log(`Wartość całkowita: ${totalPrice} ${typeof totalPrice}`);

totalPrice = sumPrices(100, 200, 300);
console.log(`Wartość całkowita: ${totalPrice} ${typeof totalPrice}`);

totalPrice = sumPrices(100, 200, undefined, false, "Witaj");
console.log(`Wartość całkowita: ${totalPrice} ${typeof totalPrice}`);

```

Funkcje — niezależnie od używanej składni — również są wartościami. Jest to specjalna kategoria typu `object` omówionej w podrozdziale „Praca z obiektem” w dalszej części rozdziału. Funkcje mogą być przypisywane zmiennym przekazywanym jako argumenty innych funkcji, a także używane w dokładnie taki sam sposób jak każda inna wartość.

W kodzie przedstawionym na listingu 3.20 składnia funkcji strzałki została użyta do zdefiniowania funkcji przypisywanej zmiennej o nazwie `sumPrices`. Funkcje są specjalne, ponieważ mogą być wywoływane. Możliwość traktowania funkcji jako wartości pozwala na zwięzłe wyrażanie skomplikowanych funkcji, choć to łatwo prowadzi do utworzenia kodu trudnego w odczycie. W książce znajdziesz znacznie więcej przykładów funkcji strzałki oraz używania funkcji jako wartości.

Praca z tablicą

Tablica JavaScriptu stosuje podejście znane z większości języków programowania, z wyjątkiem tego, że jej wielkość można zmieniać dynamicznie, a jej zawartością może być dowolne połączenie wartości, a tym samym dowolne połączenie typów. Kod na listingu 3.21 pokazuje przykład zdefiniowania i używania tablicy.

Listing 3.21. Definiowanie i używanie tablicy w kodzie pliku `index.js` w katalogu `primer`

```

let names = ["czapka", "buty", "rękawiczki"];
let prices = [];

prices.push(100);
prices.push("100");
prices.push(50.25);

console.log(`Pierwszy element: ${names[0]}: ${prices[0]}`);

let sumPrices = (...numbers) => numbers.reduce((total, val) =>
    total + (Number.isNaN(Number(val)) ? 0 : Number(val)));

let totalPrice = sumPrices(...prices);
console.log(`Wartość całkowita: ${totalPrice} ${typeof totalPrice}`);

```

Wielkość tablicy nie jest określana podczas jej tworzenia i będzie alokowana automatycznie w trakcie dodawania lub usuwania elementów. Indeksy tablic w JavaScriptcie zaczynają się od zera i są definiowane za pomocą nawiasu kwadratowego, opcjonalnie z zawartością początkową w postaci

elementów rozdzielonych przecinkami. W omawianym przykładzie tablica `names` została utworzona z wartościami tekstowymi. Z kolei tablica `prices` została utworzona jako pusta, a metoda `push()` umieszczała elementy na końcu tablicy.

Elementy tablicy mogą być odczytywane lub definiowane za pomocą nawiasu kwadratowego, a także przetwarzane za użyciem metod wymienionych w tabeli 3.2.

Tabela 3.2. *Użyteczne metody tablicy*

Metoda	Opis
<code>concat(innaTablica)</code>	Metoda zwraca nową tablicę powstałą w wyniku połączenia tablicy bieżącej z podaną w argumencie. Istnieje możliwość podania wielu tablic
<code>join(separator)</code>	Metoda łączy wszystkie elementy w tablicy w ciąg tekstowy. Argument określa znak używany do rozgraniczenia elementów
<code>pop()</code>	Metoda usuwa i zwraca ostatni element w tablicy
<code>shift()</code>	Metoda usuwa i zwraca pierwszy element w tablicy
<code>push(item)</code>	Metoda dołącza określony element na końcu tablicy
<code>unshift(item)</code>	Metoda wstawia nowy element na początku tablicy
<code>reverse()</code>	Metoda zwraca nową tablicę zawierającą elementy umieszczone w odwrotnej kolejności
<code>slice(początek, koniec)</code>	Metoda zwraca fragment tablicy
<code>sort()</code>	Metoda sortuje tablicę. W celu wykonania niestandardowego porównania można użyć opcjonalnej funkcji porównania. Jeżeli taka funkcja nie została zdefiniowana, przeprowadzane jest sortowanie w kolejności alfabetycznej
<code>splice(indeks, liczba)</code>	Metoda usuwa podaną liczbę elementów, począwszy od elementu o wskazanym indeksie. Usunięte elementy są zwracane jako wynik działania metody
<code>every(test)</code>	Metoda wywołuje funkcję <code>test()</code> dla każdego elementu w tablicy i zwraca wartość <code>true</code> , jeżeli wynikiem wszystkich wywołań funkcji <code>test()</code> również była wartość <code>true</code> . W przeciwnym razie metoda zwraca wartość <code>false</code>
<code>some(test)</code>	Metoda zwraca wartość <code>true</code> , gdy wywołanie funkcji <code>test()</code> dla każdego elementu tablicy przynajmniej raz zwróciło wartość <code>true</code>
<code>filter(test)</code>	Metoda zwraca nową tablicę zawierającą elementy, dla których funkcja <code>test()</code> zwróciła wartość <code>true</code>
<code>find(test)</code>	Metoda zwraca pierwszy element tablicy, dla którego funkcja <code>test()</code> zwróciła wartość <code>true</code>

Tabela 3.2. *Użyteczne metody tablicy (ciąg dalszy)*

Metoda	Opis
<code>findIndex(test)</code>	Metoda zwraca indeks pierwszego elementu tablicy, dla którego funkcja <code>test()</code> zwróciła wartość <code>true</code>
<code>forEach(funkcjaWywołaniaZwrotnego)</code>	Metoda wywołuje podaną w argumencie funkcję dla każdego elementu tablicy, zgodnie z opisem przedstawionym we wcześniejszej części rozdziału
<code>includes(value)</code>	Metoda zwraca wartość <code>true</code> , gdy tablica zawiera podaną wartość
<code>map(funkcjaWywołaniaZwrotnego)</code>	Metoda zwraca nową tablicę zawierającą wynik wywołania podanej w argumencie <code>map()</code> funkcji wywołania zwrótnego dla każdego elementu tablicy
<code>reduce(funkcjaWywołaniaZwrotnego)</code>	Metoda zwraca akumulowaną wartość wygenerowaną przez wykonanie funkcji wywołania zwrótnego dla każdego elementu tablicy

Używanie operatora rozwinienia w tablicy

Operator rozwinienia (ang. *spread operator*) może być używany do rozszerzenia zawartości tablicy w taki sposób, aby jej elementy były stosowane jako argumenty funkcji. Operator rozwinienia ma postać trzech kropek. W kodzie przedstawionym na listingu 3.21 został użyty w celu przekazania zawartości tablicy do funkcji `sumPrices()`.

```
...
let totalPrice = sumPrices(...prices);
...
```

Operator jest używany przed nazwą tablicy. Operator rozwinienia może być używany do rozszerzenia zawartości tablicy w celu łatwiejszej konkatencji, jak pokazałem na listingu 3.22.

Listing 3.22. *Używanie operatora rozwinienia w kodzie pliku `index.js` w katalogu `primer`*

```
let names = ["czapka", "buty", "rękawiczki"];
let prices = [];

prices.push(100);
prices.push("100");
prices.push(50.25);

console.log(`Pierwszy element: ${names[0]}: ${prices[0]}`);

let sumPrices = (...numbers) => numbers.reduce((total, val) =>
  total + (Number.isNaN(Number(val)) ? 0 : Number(val)));

let totalPrice = sumPrices(...prices);
console.log(`Wartość całkowita: ${totalPrice} ${typeof totalPrice}`);

let combinedArray = [...names, ...prices];
combinedArray.forEach(element => console.log(`Połączony element tablicy: ${element}`));
```

Operator rozwinięcia jest używany do utworzenia tablicy zawierającej elementy pochodzące z tablic `names` i `prices`. Kod przedstawiony na listingu 3.22 powoduje wygenerowanie następujących danych wyjściowych:

```
Pierwszy element: czapka: 100
Wartość całkowita: 250.25 number
Połączony element tablicy: czapka
Połączony element tablicy: buty
Połączony element tablicy: rękawiczki
Połączony element tablicy: 100
Połączony element tablicy: 100
Połączony element tablicy: 50.25
```

Destrukturyzacja tablicy

Wartości tablicy mogą zostać rozpakowane za pomocą składni przypisania destrukturyzującego, która powoduje przypisanie zmiennym wybranych wartości, jak pokazałem na listingu 3.23.

Listing 3.23. Przykład destrukturyzacji tablicy w kodzie pliku `index.js` w katalogu `primer`

```
let names = ["czapka", "buty", "rękawiczki"];
```

```
let [one, two] = names;
console.log(`Jeden: ${one}, dwa: ${two}`);
```

Lewa strona wyrażenia została użyta do określenia zmiennych, którym będą przypisane wartości. W omawianym przykładzie pierwsza wartość tablicy `names` będzie przypisana zmiennej o nazwie `one`, druga zaś zmiennej `two`. Liczba zmiennych nie musi odpowiadać liczbie elementów znajdujących się w tablicy. Elementy, dla których nie zostały podane zmienne w składni przypisania destrukturyzującego, zostaną po prostu zignorowane. Natomiast zmienne w składni przypisania destrukturyzującego, dla których nie będą istniały elementy w tablicy, otrzymają wartość `undefined`. Kod zamieszczony na listingu 3.23 spowoduje wygenerowanie następujących danych wyjściowych:

```
Jeden: czapka, dwa: buty
```

Ignorowanie elementów podczas destrukturyzacji tablicy

Istnieje możliwość zignorowania elementów przez pominięcie ich nazw w składni przypisania destrukturyzującego, jak pokazałem na listingu 3.24.

Listing 3.24. Ignorowanie elementów w kodzie pliku `index.js` w katalogu `primer`

```
let names = ["czapka", "buty", "rękawiczki"];
```

```
let [, , three] = names;
console.log(`Trzy: ${three}`);
```

W pierwszych dwóch położeniach przypisania destrukturyzującego nie zostały podane nazwy zmiennych, więc dwa pierwsze elementy tablicy zostaną zignorowane. Natomiast trzeci element

zostanie przypisany zmiennej `three`, a kod listingu spowoduje wygenerowanie następujących danych wyjściowych:

Trzy: rękawiczki

Przypisywanie pozostałych elementów do tablicy

Ostatnia nazwa zmiennej w składni przypisania destrukuryzującego może być poprzedzona trzema kropkami, dzięki którym otrzymujemy tzw. *wyrażenie resztowe* lub *wzorzec resztowy*, używany następnie do przypisania pozostałych elementów do tablicy. Przykład takiego rozwiązania pokazałem na listingu 3.25. (Dla zachowania spójności wyrażenie resztowe jest często określane mianem *operatora rozwinięcia*, ponieważ w obu przypadkach są używane trzy kropki, a sposób działania wyrażenia i operatora jest podobny).

Listing 3.25. Przykład przypisania pozostałych elementów do tablicy w kodzie pliku `index.js` w katalogu `primer`

```
let names = ["czapka", "buty", "rękawiczki"];

let [, , three] = names;
console.log(`Trzy: ${three}`);

let prices = [100, 120, 50.25];
let [, ...highest] = prices.sort((a, b) => a - b);
highest.forEach(price => console.log(`Najwyższa cena: ${price}`));
```

Tablica `prices` zostaje posortowana, pierwszy element jest odrzucany, a pozostałe są przypisywane tablicy o nazwie `highest`. Wartości tej tablicy są iterowane w celu ich wyświetlenia w konsoli i tym samym wygenerowania następujących danych wyjściowych:

Trzy: rękawiczki
Najwyższa cena: 100
Najwyższa cena: 120

Praca z obiektem

Obiekt JavaScriptu to kolekcja właściwości, z których każda ma nazwę i wartość. Najprostszy sposób na zdefiniowanie obiektu polega na użyciu składni literału, jak pokazałem na listingu 3.26.

Listing 3.26. Tworzenie obiektu w kodzie pliku `index.js` w katalogu `primer`

```
let hat = {
  name: "czapka",
  price: 100
};

let boots = {
  name: "buty",
  price: "100"
```

```

}

let sumPrices = (...numbers) => numbers.reduce((total, val) =>
  total + (Number.isNaN(Number(val)) ? 0 : Number(val)));

let totalPrice = sumPrices(hat.price, boots.price);
console.log(`Wartość całkowita: ${totalPrice} ${typeof totalPrice}`);

```

Składnia literału wykorzystuje nawias klamrowy, w którym znajduje się lista nazw właściwości i ich wartości. Nazwa jest oddzielona dwukropkiem od wartości, natomiast poszczególne pary nazw i wartości są rozdzielone przecinkami. Obiekt można przypisać zmiennej, użyć go jako argumentu funkcji, a także przechowywać w tablicy. Dwa obiekty utworzone w kodzie na listingu 3.26 zostały przypisane zmiennym o nazwach `hat` i `boots`. Dostęp do właściwości zdefiniowanych w obiekcie odbywa się za pomocą nazw zmiennych, jak pokazałem w kolejnym poleceniu, którego działanie polega na pobraniu wartości właściwości `price` zdefiniowanej w obu wymienionych obiektach.

```

...
let totalPrice = sumPrices(hat.price, boots.price);
...

```

Kod przedstawiony na listingu 3.26 powoduje wygenerowanie następujących danych wyjściowych:

```
Wartość całkowita: 200 number
```

Dodawanie, modyfikowanie i usuwanie właściwości obiektu

Podobnie jak inne konstrukcje JavaScriptu, także obiekty są dynamiczne. Właściwości mogą być dodawane i usuwane, a wartości dowolnego typu można przypisywać właściwościom, jak pokazałem na listingu 3.27.

Listing 3.27. *Przeprowadzanie operacji na obiekcie w kodzie pliku `index.js` w katalogu `primer`*

```

let hat = {
  name: "czapka",
  price: 100
};

let boots = {
  name: "buty",
  price: "100"
}

let gloves = {
  productName: "rękawiczki",
  price: "40"
}

gloves.name = gloves.productName;
delete gloves.productName;
gloves.price = 20;

```

```
let sumPrices = (...numbers) => numbers.reduce((total, val) =>
  total + (Number.isNaN(Number(val)) ? 0 : Number(val)));
```

```
let totalPrice = sumPrices(hat.price, boots.price, gloves.price);
console.log(`Wartość całkowita: ${totalPrice} ${typeof totalPrice}`);
```

Obiekt `gloves` został utworzony z właściwościami `productName` i `price`. W kolejnych poleceniach mamy utworzenie właściwości `name`, użycie słowa kluczowego `delete` do usunięcia właściwości, przypisanie właściwości `price` wartości typu `number` oraz zastąpienie poprzedniej wartości typu `string`. Kod przedstawiony na listingu 3.27 powoduje wygenerowanie następujących danych wyjściowych:

```
Wartość całkowita: 220 number
```

Zabezpieczenie przed niezdefiniowanymi obiektami i właściwościami

Należy zachować ostrożność podczas używania obiektów, ponieważ nie muszą one mieć takiego samego *kształtu* (to pojęcie służy do określenia połączenia właściwości i wartości), którego można by się spodziewać, lub tego, który został użyty początkowo podczas tworzenia obiektu.

Skoro kształt obiektu może ulec zmianie, przypisanie lub pobranie wartości niezdefiniowanej właściwości nie jest błędem. Jeżeli przypiszesz wartość nieistniejącej właściwości, zostanie ona utworzona w obiekcie i otrzyma podaną wartość. Natomiast wynikiem próby odczytania nieistniejącej właściwości jest wartość `undefined`. Użytecznym sposobem zagwarantowania, że kod zawsze ma wartość, jest poleganie na funkcji koercji typu oraz na operatorze logicznego LUB, jak pokazałem na listingu 3.28.

Listing 3.28. Zabezpieczenie przed niezdefiniowanymi wartościami zastosowane w kodzie pliku `index.js` w katalogu `primer`

```
let hat = {
  name: "czapka",
  price: 100
};

let boots = {
  name: "buty",
  price: "100"
}

let gloves = {
  productName: "rękawiczki",
  price: "40"
}

gloves.name = gloves.productName;
delete gloves.productName;
gloves.price = 20;

let propertyCheck = hat.price ?? 0;
let objectAndPropertyCheck = (hat ?? {}).price ?? 0;
console.log(`Sprawdzenie: ${propertyCheck}, ${objectAndPropertyCheck}`);
```

Kod może być trudny w odczycie, ale operator `??` przeprowadza koercję wartości `undefined` i `null` na `false`, a wszystkich pozostałych na `true`. Takie sprawdzenie można wykorzystać w celu dostarczenia wartości awaryjnej dla pojedynczej właściwości, obiektu lub połączenia właściwości i obiektu.

W pierwszej operacji sprawdzenia w kodzie na listingu 3.28 przyjęto założenie, że zmienna `hat` ma przypisaną wartość, a celem sprawdzenia jest ustalenie, czy zmienna `hat.price` istnieje i ma przypisaną wartość. Druga operacja sprawdzenia jest ostrożniejsza — i jednocześnie trudniejsza do odczytu — i ma na celu ustalenie przed sprawdzeniem właściwości `price`, czy zmiennej `hat` została przypisana wartość. Kod przedstawiony na listingu 3.28 powoduje wygenerowanie następujących danych wyjściowych:

Sprawdzenie: 100, 100

Istnieje możliwość uproszczenia drugiej operacji sprawdzenia na listingu 3.28 przez użycie łączenia opcjonalnego, jak pokazałem na listingu 3.29.

Listing 3.29. Przykład użycia łączenia opcjonalnego w kodzie pliku `index.js` w katalogu `primer`

```
let hat = {
  name: "czapka",
  price: 100
};

let boots = {
  name: "buty",
  price: "100"
}

let gloves = {
  productName: "rękawiczki",
  price: "40"
}

gloves.name = gloves.productName;
delete gloves.productName;
gloves.price = 20;

let propertyCheck = hat.price ?? 0;
let objectAndPropertyCheck = hat?.price ?? 0;
console.log(`Sprawdzenie: ${propertyCheck}, ${objectAndPropertyCheck}`);
```

Operator łączenia opcjonalnego (znak `?`) zakończy analizę wyrażenia, gdy przypisaną mu wartością jest `null` lub `undefined`. W omawianym przykładzie ten operator został użyty razem z właściwością `hat`, więc wyrażenie nie będzie próbowało odczytywać wartości właściwości `price`, gdy wartością właściwości `hat` jest `undefined` lub `null`. Jeżeli wartością `hat` lub `hat.price` jest `undefined` bądź `null`, wówczas zostanie użyta wartość awaryjna.

Używanie operatorów rozwinęcia i resztowego w obiekcie

Operator rozwinęcia może być używany w celu rozszerzania właściwości i wartości zdefiniowanych w obiekcie, co znacznie ułatwia utworzenie obiektu na podstawie właściwości zdefiniowanych w innym obiekcie, jak pokazałem na listingu 3.30.

Listing 3.30. *Używanie w obiekcie operatora rozwinęcia w kodzie pliku index.js w katalogu primer*

```
let hat = {
  name: "czapka",
  price: 100
};

let boots = {
  name: "buty",
  price: "100"
}

let otherHat = { ...hat };
console.log(`Rozwinięcie: ${otherHat.name}, ${otherHat.price}`);
```

Operator rozwinęcia jest używany w celu dołączenia właściwości obiektu `hat` jako części składni literału obiektu. W omawianym przykładzie operator rozwinęcia powoduje skopiowanie właściwości z obiektu `hat` do nowego `otherHat` i następuje wygenerowanie pokazanych tutaj danych wyjściowych:

```
Rozwinięcie: czapka, 100
```

Operator rozwinęcia może być łączony również z innymi właściwościami w celu dodawania, zastępowania lub wykorzystywania właściwości z obiektu źródłowego, jak pokazałem na listingu 3.31.

Listing 3.31. *Dodawanie, zastępowanie i używanie właściwości w kodzie pliku index.js w katalogu primer*

```
let hat = {
  name: "czapka",
  price: 100
};

let boots = {
  name: "buty",
  price: "100"
}

let additionalProperties = { ...hat, discounted: true};
console.log(`Dodatkowe: ${JSON.stringify(additionalProperties)}`);

let replacedProperties = { ...hat, price: 10};
console.log(`Zastąpione: ${JSON.stringify(replacedProperties)}`);

let { price, ...someProperties } = hat;
console.log(`Wybrane: ${JSON.stringify(someProperties)}`);
```

Nazwy i wartości właściwości rozwinięte za pomocą operatora rozwinięcia są traktowane tak, jakby były zdefiniowane za pomocą składni literału obiektu. Oznacza to możliwość zmiany kształtu obiektu przez połączenie operatora rozwinięcia z innymi właściwościami. Na przykład polecenie:

```
...
let additionalProperties = { ...hat, discounted: true};
...
```

zostanie rozszerzone tak, aby właściwości definiowane przez obiekt `hat` były połączone z właściwością `discounted`, co jest odpowiednikiem następującego polecenia:

```
let additionalProperties = { name: "czapka", price: 100, discounted: true};
```

Jeżeli nazwa właściwości została użyta dwukrotnie w składni literału obiektu, wówczas zastosowanie będzie miała druga z tych wartości. Takiej możliwości można użyć do zmiany wartości właściwości pobranej za pomocą operatora rozwinięcia, więc polecenie:

```
...
let replacedProperties = { ...hat, price: 10};
...
```

zostanie rozszerzone i stanie się odpowiednikiem następującego:

```
let replacedProperties = { name: "czapka", price: 100, price: 10};
```

Efektem jest obiekt z właściwością `name` i wartością pochodzącą z obiektu `hat`, natomiast wartością właściwości `price` będzie 10. Operator resztowy (jego nazwa to również trzy kropki, podobnie jak w przypadku operatora rozwinięcia) może być używany do pobierania właściwości lub ich wykluczania, gdy jest stosowana składnia literału obiektu. Przedstawione tutaj polecenie definiuje zmienne o nazwach `price` i `someProperties`.

```
...
let { price, ...someProperties } = hat;
...
```

Właściwości zdefiniowane przez obiekt `hat` są rozkładane. Właściwość `hat.price` zostaje przypisana nowej właściwości `price`, a wszystkie pozostałe właściwości są przypisywane obiektowi `someProperties`.

Wbudowana metoda `JSON.stringify()` powoduje utworzenie w postaci ciągu tekstowego reprezentacji obiektu przy użyciu formatu danych JSON. Takie rozwiązanie jest użyteczne jedynie podczas przedstawiania prostych obiektów — nie sprawdza się w trakcie pracy z np. funkcjami — i pomagają w zrozumieniu tego, jak są tworzone obiekty. Kod przedstawiony na listingu 3.31 powoduje wygenerowanie następujących danych wyjściowych.

```
Dodatkowe: {"name":"czapka","price":100,"discounted":true}
Zastąpione: {"name":"czapka","price":10}
Wybrane: {"name":"czapka"}
```

Definiowanie funkcji typu getter i setter

Getter i setter to funkcje wywoływane podczas odczytywania lub definiowania wartości właściwości, jak pokazałem na listingu 3.32.

Listing 3.32. *Używanie funkcji typu getter i setter w kodzie pliku index.js w katalogu primer*

```
let hat = {
  name: "czapka",
  _price: 100,
  priceIncTax: 100 * 1.2,

  set price(newPrice) {
    this._price = newPrice;
    this.priceIncTax = this._price * 1.2;
  },

  get price() {
    return this._price;
  }
};

let boots = {
  name: "buty",
  price: "100",

  get priceIncTax() {
    return Number(this.price) * 1.2;
  }
}

console.log(`czapka: ${hat.price}, ${hat.priceIncTax}`);
hat.price = 120;
console.log(`czapka: ${hat.price}, ${hat.priceIncTax}`);

console.log(`buty: ${boots.price}, ${boots.priceIncTax}`);
boots.price = "120";
console.log(`buty: ${boots.price}, ${boots.priceIncTax}`);
```

W omawianym przykładzie wprowadziłem właściwość `priceIncTax`, której wartość jest uaktualniana automatycznie podczas definiowania właściwości `price`. Obiekt `hat` pozwala na to przy użyciu funkcji typu getter i setter dla właściwości `price` w celu uaktualnienia właściwości pomocniczej o nazwie `_price`. Po przypisaniu nowej wartości właściwości `price` funkcja typu setter powoduje uaktualnienie właściwości `_price` i `priceIncTax`. Natomiast podczas odczytywania właściwości `price` funkcja typu getter przekazuje wartość właściwości `_price`. (Właściwość pomocnicza jest wymagana, ponieważ funkcje typu getter i setter są traktowane jako właściwości i nie mogą mieć takich samych nazw jak wszelkie pozostałe właściwości konwencjonalne zdefiniowane w obiekcie).

Właściwości prywatne w języku JavaScript

JavaScript nie oferuje żadnego wbudowanego mechanizmu przeznaczonego do obsługi właściwości prywatnych, czyli takich, które są dostępne jedynie dla metod obiektu oraz jego funkcji typu getter i setter. Wprawdzie istnieją techniki pozwalające na osiągnięcie podobnego w skutkach efektu, ale są one skomplikowane. Najczęściej stosowane podejście polega na wykorzystaniu konwencji nazw do oznaczenia właściwości nieprzeznaczonych do publicznego używania. Nie zabrania to dostępu do tych właściwości, ale przynajmniej pokazuje, że jest to niepożądane. Najczęściej stosowaną konwencją jest poprzedzenie nazwy właściwości znakiem podkreślenia, jak to pokazałem w przypadku właściwości `_price` na listingu 3.32.

W trakcie powstawania tej książki była zgłoszona propozycja, aby w ramach procesu standaryzacji dodać do języka JavaScript obsługę właściwości prywatnych. Zgodnie z tą propozycją nazwy właściwości prywatnych mają być poprzedzane znakiem `#`. Upłynie jednak nieco czasu, zanim ta funkcja stanie się częścią standardu JavaScriptu. Z kolei TypeScript obsługuje właściwości prywatne, co dokładnie omówię w rozdziale 11.

Obiekt `boots` definiuje takie samo zachowanie, jakie ma obiekt `hat`, ale odbywa się to przez utworzenie funkcji typu getter, która nie ma odpowiadającej jej funkcji typu setter. Dlatego też wartość może być pobierana, ale już nie modyfikowana — to pokazuje wyraźnie, że funkcje typu getter i setter nie muszą być używane razem. Kod przedstawiony na listingu 3.32 powoduje wygenerowanie następujących danych wyjściowych:

```
czapka: 100, 120
czapka: 120, 144
buty: 100, 120
buty: 120, 144
```

Definiowanie metod

Na pierwszy rzut oka JavaScript może być dezorientujący, ale po zagłębieniu się w szczegóły odkrywamy spójność, która nie zawsze jest oczywista. Jednym z przykładów są metody, na podstawie których zostały opracowane funkcjonalności omówione nieco wcześniej w rozdziale. Spójrz na przykład przedstawiony na listingu 3.33.

Listing 3.33. Definiowanie metod w kodzie pliku `index.js` w katalogu `primer`

```
let hat = {
  name: "czapka",
  _price: 100,
  priceIncTax: 100 * 1.2,

  set price(newPrice) {
    this._price = newPrice;
    this.priceIncTax = this._price * 1.2;
  },

  get price() {
    return this._price;
  },
}
```

```

    writeDetails: function() {
        console.log(`${this.name}: ${this.price}, ${this.priceIncTax}`);
    }
};

let boots = {
    name: "buty",
    price: "100",

    get priceIncTax() {
        return Number(this.price) * 1.2;
    }
}

hat.writeDetails();
hat.price = 120;
hat.writeDetails();

console.log(`buty: ${boots.price}, ${boots.priceIncTax}`);
boots.price = "120";
console.log(`buty: ${boots.price}, ${boots.priceIncTax}`);

```

Metoda jest właściwością, której wartością jest funkcja. Oznacza to, że cała funkcjonalność i zachowanie oferowane przez funkcję, czyli np. parametry domyślne i resztowe, mogą być wykorzystane także w metodach. Metoda na listingu 3.33 została zdefiniowana za pomocą słowa kluczowego `function`, choć istnieje znacznie zwięźlejsza składnia, jak pokazałem na listingu 3.34.

Listing 3.34. *Używanie zwięzłej składni metod w kodzie pliku `index.js` w katalogu `primer`*

```

...
writeDetails() {
    console.log(`${this.name}: ${this.price}, ${this.priceIncTax}`);
}
...

```

Pominięte zostało słowo kluczowe `function` i dwukropek oddzielający nazwę właściwości od jej wartości, co pozwala na definiowanie metody w stylu uważanym przez wielu programistów za naturalny. Oto dane wyjściowe wygenerowane przez listing 3.34:

```

czapka: 100, 120
czapka: 120, 144
buty: 100, 120
buty: 120, 144

```

Słowo kluczowe `this`

Słowo kluczowe `this` może być dezorientujące nawet dla doświadczonych programistów JavaScriptu. W innych językach programowania słowo kluczowe `this` pozwala na odwołanie się do bieżącego egzemplarza obiektu utworzonego na podstawie klasy. W JavaScriptcie wydaje się, że słowo kluczowe `this` często działa w taki właśnie sposób — przynajmniej do chwili, gdy wprowadzona w kodzie zmiana nie spowoduje uszkodzenia aplikacji i wygenerowania wartości `undefined`.

Aby to pokazać, posłużę się składnią grubej strzałki do zdefiniowania metody w obiekcie `hat`. Spójrz na fragment kodu przedstawiony na listingu 3.35.

Listing 3.35. *Używanie składni grubej strzałki w kodzie pliku `index.js` w katalogu `primer`*

```
let hat = {
  name: "czapka",
  _price: 100,
  priceIncTax: 100 * 1.2,

  set price(newPrice) {
    this._price = newPrice;
    this.priceIncTax = this._price * 1.2;
  },

  get price() {
    return this._price;
  },

  writeDetails: () =>
    console.log(`${this.name}: ${this.price}, ${this.priceIncTax}`)
};

let boots = {
  name: "buty",
  price: "100",

  get priceIncTax() {
    return Number(this.price) * 1.2;
  }
}

hat.writeDetails();
hat.price = 120;
hat.writeDetails();

console.log(`buty: ${boots.price}, ${boots.priceIncTax}`);
boots.price = "120";
console.log(`buty: ${boots.price}, ${boots.priceIncTax}`);
```

Ta metoda używa dokładnie tego samego polecenia `console.log()` co na listingu 3.34, ale po zapisaniu zmian i wykonaniu kodu w danych wyjściowych zaczynają się pojawiać wartości `undefined`:

```
undefined: undefined, undefined
undefined: undefined, undefined
buty: 100, 120
buty: 120, 144
```

Aby zrozumieć, co tak naprawdę się dzieje, i mieć możliwość rozwiązania problemu, najpierw trzeba wykonać krok wstecz i przeanalizować faktyczny sposób działania słowa kluczowego `this` w JavaScriptcie.

Słowo kluczowe `this` w oddzielnych funkcjach

Słowo kluczowe `this` może być stosowane w dowolnej funkcji, nawet takiej nieużywanej jako metoda, co pokazałem na listingu 3.36.

Listing 3.36. Wywołanie funkcji w kodzie pliku `index.js` w katalogu `primer`

```
function writeMessage(message) {
    console.log(`${this.greeting}, ${message}`);
}

greeting = "Witaj";

writeMessage("dzisiaj mamy słoneczny dzień");
```

Działanie funkcji `writeMessage()` polega na odczytaniu właściwości o nazwie `greeting` z `this` w jednym z wyrażeń w szablonie ciągu tekstowego przekazanego metodzie `console.log()`. Słowo kluczowe `this` nie pojawi się ponownie na listingu, ale po zapisaniu i wykonaniu kodu zostaną wygenerowane następujące dane wyjściowe:

```
Witaj, dzisiaj mamy słoneczny dzień
```

JavaScript definiuje obiekt globalny, któremu można przypisywać wartości dostępne w całej aplikacji. Ten obiekt globalny jest używany w celu zapewnienia dostępu do podstawowych funkcji środowiska uruchomieniowego, np. obiektu `document` w przeglądarce WWW pozwalającego na współdziałanie z API modelu DOM.

Wartości przypisywane nazwom bez używania słów kluczowych `let`, `const` lub `var` są przypisywane obiektowi globalnemu. Polecenie przypisujące wartość w postaci ciągu tekstowego `Witaj` powoduje utworzenie zmiennej w przestrzeni globalnej. Gdy funkcja zostanie wykonana, wartością `this` będzie obiekt globalny, więc wywołanie `this.greeting` zwraca ciąg tekstowy `Witaj`, co wyjaśnia w ten sposób dane wyjściowe wygenerowane przez aplikację.

Wprawdzie standardowy sposób wywołania funkcji polega na użyciu nawiasu zawierającego argumenty, ale JavaScript oferuje wygodną składnię, która zostaje przekształcona na polecenie, jak pokazałem na listingu 3.37.

Listing 3.37. Wywołanie funkcji w kodzie pliku `index.js` w katalogu `primer`

```
function writeMessage(message) {
    console.log(`${this.greeting}, ${message}`);
}

greeting = "Witaj";

writeMessage("dzisiaj mamy słoneczny dzień");
writeMessage.call(global, "dzisiaj mamy słoneczny dzień");
```

Jak wcześniej wyjaśniłem, funkcje są obiektami, co oznacza, że definiują metody, w tym także metodę `call()`. Ta metoda jest używana do wywołania funkcji w tle. Pierwszym argumentem metody `call()` jest wartość dla `this`, którą w omawianym przykładzie jest obiekt globalny. Z tego powodu słowo kluczowe `this` może być używane w dowolnej funkcji i dlatego też domyślnie zwraca obiekt globalny.

Nowe polecenie przedstawione na listingu 3.37 bezpośrednio używa metody `call()` i przypisuje `this` obiekt globalny. Wynik jest dokładnie taki sam jak w przypadku wcześniejszego wywołania funkcji konwencjonalnej, co widać na wygenerowanych danych wyjściowych:

```
Witaj, dzisiaj mamy słoneczny dzień
Witaj, dzisiaj mamy słoneczny dzień
```

Nazwa obiektu globalnego zmienia się na podstawie środowiska uruchomieniowego. W przypadku kodu wykonywanego przez środowisko uruchomieniowe Node.js używany jest obiekt `global`, natomiast w przeglądarkach WWW będzie to obiekt `window` lub `self`. W chwili powstawania książki była zgłoszona propozycja standaryzowania nazwy `global`, ale nie została ona jeszcze wprowadzona do specyfikacji JavaScriptu.

Efekt działania trybu ścisłego

JavaScript obsługuje „tryb ścisły” — powoduje wyłączenie lub ograniczenie funkcji, które wcześniej przyczyniały się do powstawania oprogramowania niskiej jakości lub uniemożliwiały środowisku uruchomieniowemu efektywne wykonywanie kodu. Po włączeniu trybu ścisłego wartością domyślną dla `this` jest `undefined`, co ma na celu ochronę przed przypadkowym użyciem obiektu globalnego, natomiast wartości w zasięgu globalnym muszą być wyraźnie zdefiniowane jako właściwości obiektu globalnego. Więcej informacji na temat trybu ścisłego znajdziesz na stronie https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode. Kompilator TypeScript oferuje funkcję automatycznego włączania trybu ścisłego w generowanym kodzie JavaScriptu, co dokładnie przedstawię w rozdziale 5.

Słowo kluczowe `this` w metodach

Gdy funkcja jest wywoływana jako metoda obiektu, wówczas `this` wskazuje obiekt, jak pokazałem w przykładzie na listingu 3.38.

Listing 3.38. Wywołanie funkcji jako metody w kodzie pliku `index.js` w katalogu `primer`

```
let myObject = {
  greeting: "Witaj, przyjacielu",

  writeMessage(message) {
    console.log(`${this.greeting}, ${message}`);
  }
}

greeting = "Witaj";

myObject.writeMessage("dzisiaj mamy słoneczny dzień");
```

Gdy funkcja jest wywoływana za pomocą obiektu, polecenie wywołujące tę funkcję jest odpowiednikiem użycia metody `call()` z pierwszym argumentem w postaci obiektu:

```
...
myObject.writeMessage.call(myObject, "dzisiaj mamy słoneczny dzień");
...
```

Trzeba zachować ostrożność, ponieważ słowo kluczowe `this` jest definiowane inaczej, gdy funkcja jest dostępna na zewnątrz obiektu, co może się zdarzyć w przypadku przypisania funkcji zmiennej, jak pokazałem na listingu 3.39.

Listing 3.39. Wywołanie funkcji poza obiektem w kodzie pliku `index.js` w katalogu `primer`

```
let myObject = {
  greeting: "Witaj, przyjacielu",

  writeMessage(message) {
    console.log(`${this.greeting}, ${message}`);
  }
}

greeting = "Witaj";

myObject.writeMessage("dzisiaj mamy słoneczny dzień");

let myFunction = myObject.writeMessage;
myFunction("dzisiaj mamy słoneczny dzień");
```

Funkcja może być używana podobnie jak każda inna wartość, co oznacza również możliwość przypisania funkcji zmiennej poza obiektem, w którym została zdefiniowana — takie rozwiązanie mogłeś zobaczyć w przykładzie na listingu 3.39. Jeżeli funkcja jest wywoływana za pomocą zmiennej, wówczas `this` prowadzi do obiektu globalnego. To często powoduje problemy, gdy funkcja jest używana jako argument innej metody lub jako wywołanie zwrotne przeznaczone do obsługi zdarzeń. W efekcie ta sama funkcja będzie działała odmiennie w zależności od sposobu jej wywołania, jak widać w danych wyjściowych wygenerowanych przez kod przedstawiony na listingu 3.39:

```
Witaj, przyjacielu, dzisiaj mamy słoneczny dzień
Witaj, dzisiaj mamy słoneczny dzień
```

Zmiana zachowania słowa kluczowego `this`

Jednym ze sposobów na kontrolowanie wartości `this` jest wywoływanie funkcji za pomocą metody `call()`, ale jest to rozwiązanie niewygodne i musi być stosowane w trakcie każdego wywołania funkcji. Znacznie bardziej niezawodnym rozwiązaniem jest używanie metody `bind()` funkcji, która jest wykorzystywana do zdefiniowania wartości `this` niezależnie od sposobu wywołania funkcji, jak pokazałem na listingu 3.40.

Listing 3.40. Definiowanie wartości w kodzie pliku `index.js` w katalogu `primer`

```
let myObject = {
  greeting: "Witaj, przyjacielu",

  writeMessage(message) {
```

```
        console.log(`${this.greeting}, ${message}`);
    }
}
```

```
myObject.writeMessage = myObject.writeMessage.bind(myObject);
```

```
greeting = "Witaj";
```

```
myObject.writeMessage("dzisiaj mamy słoneczny dzień");
```

```
let myFunction = myObject.writeMessage;
myFunction("dzisiaj mamy słoneczny dzień");
```

Metoda `bind()` zwraca nową funkcję ze stałą wartością dla `this` stosowaną po wywołaniu funkcji. Funkcja zwrócona przez metodę `bind()` jest używana do zastąpienia metody początkowej i gwarantuje spójność podczas wywoływania metody `writeMessage()`. Używanie metody `bind()` jest niewygodne, ponieważ odwołanie do obiektu jest niedostępne aż do chwili utworzenia tego obiektu, co prowadzi do dwuetapowego procesu tworzenia obiektu i wywołania metody `bind()` w celu zastąpienia wszystkich metod, w których jest wymagana spójna wartość `this`. Kod przedstawiony na listingu 3.40 powoduje wygenerowanie następujących danych wyjściowych:

```
Witaj, przyjacielu, dzisiaj mamy słoneczny dzień
Witaj, przyjacielu, dzisiaj mamy słoneczny dzień
```

Wartością `this` zawsze jest `myObject`, nawet jeśli `writeMessage()` będzie wywołana jako samodzielna funkcja.

Słowo kluczowe `this` w funkcji strzałki

Sytuację związaną z obsługą `this` jeszcze bardziej komplikuje to, że funkcja strzałki nie działa w dokładnie taki sam sposób jak zwykła. Funkcja strzałki nie ma własnej wartości `this` i dziedziczy najbliższą taką wartość, którą może znaleźć w trakcie wykonywania swojego kodu. Aby pokazać, jak to działa, w kodzie na listingu 3.41 dodałem funkcję strzałki do omawianego przykładu.

Listing 3.41. *Użycie funkcji strzałki w kodzie pliku `index.js` w katalogu `primer`*

```
let myObject = {
    greeting: "Witaj, przyjacielu",

    getWriter() {
        return (message) => console.log(`${this.greeting}, ${message}`);
    }
}

greeting = "Witaj";

let writer = myObject.getWriter();
writer("dzisiaj mamy deszczowy dzień");

let standAlone = myObject.getWriter;
```



```
let standAloneWriter = standAlone();
standAloneWriter("dzisiaj mamy słoneczny dzień");
```

Przedstawiona na listingu 3.41 funkcja `getWriter()` to zwykła funkcja zwracająca wynik w postaci funkcji strzałki. Po wywołaniu funkcji strzałki zwróconej przez `getWriter()` będzie poruszała się w górę zasięgu aż do znalezienia wartości dla `this`. W efekcie sposób wywołania `getWriter()` ma wpływ na wartość `this` dla funkcji strzałki. Spójrz na dwa polecenia wywołujące funkcję `getWriter()`:

```
...
let writer = myObject.getWriter();
writer("dzisiaj mamy deszczowy dzień");
...
```

Można je połączyć w jedno następujące polecenie:

```
...
myObject.getWriter()("dzisiaj mamy deszczowy dzień");
...
```

To połączone polecenie jest nieco trudniejsze w odczycie, ale pomaga w podkreśleniu tego, że wartość `this` zależy od sposobu wywołania funkcji. Metoda `getWriter()` jest wywoływana za pomocą `myObject` i wówczas wartością `this` będzie `myObject`. Natomiast gdy wywoływana jest funkcja strzałki, znajduje ona wartość `this` pochodzącą z funkcji `getWriter()`. Dlatego też jeśli metoda `getWriter()` jest wywoływana za pomocą `myObject`, wartością `this` w funkcji strzałki będzie `myObject`, a wyrażenie `this.greeting` w szablonie ciągu tekstowego będzie miało wartość Witaj, przyjacielu.

Polecenia w drugim zestawie traktują `getWriter()` jako samodzielną funkcję, więc wartością `this` będzie obiekt globalny. Po wywołaniu funkcji strzałki wartością wyrażenia `this.greeting` będzie Witaj. Kod przedstawiony na listingu 3.41 powoduje wygenerowanie następujących danych wyjściowych, które potwierdzają wartość `this` w omówionych tutaj poszczególnych przypadkach:

```
Witaj, przyjacielu, dzisiaj mamy deszczowy dzień
Witaj, dzisiaj mamy słoneczny dzień
```

Powrót do problemu początkowego

Zacząłem ten podrozdział od modyfikacji funkcji w składni strzałki i pokazałem, że zachowuje się odmiennie, co prowadzi do wartości `undefined` w danych wyjściowych. Spójrz na obiekt i jego funkcję:

```
...
let hat = {
  name: "czapka",
  _price: 100,
  priceIncTax: 100 * 1.2,

  set price(newPrice) {
    this._price = newPrice;
    this.priceIncTax = this._price * 1.2;
```

```

    },

    get price() {
        return this._price;
    },

    writeDetails: () =>
        console.log(`${this.name}: ${this.price}, ${this.priceIncTax}`)
};
...

```

Zachowanie uległo zmianie, ponieważ funkcja strzałki nie ma własnej wartości `this` i nie znajduje się wewnątrz zwykłej funkcji, która mogłaby dostarczyć tę wartość. Aby rozwiązać problem i mieć pewność otrzymania spójnych wyników, trzeba powrócić do zwykłej funkcji i użyć metody `bind()` w celu poprawienia wartości `this`, jak pokazałem na listingu 3.42.

Listing 3.42. Rozwiązywanie problemów funkcji w kodzie pliku `index.js` w katalogu `primer`

```

let hat = {
    name: "czapka",
    _price: 100,
    priceIncTax: 100 * 1.2,

    set price(newPrice) {
        this._price = newPrice;
        this.priceIncTax = this._price * 1.2;
    },

    get price() {
        return this._price;
    },

    writeDetails() {
        console.log(`${this.name}: ${this.price}, ${this.priceIncTax}`);
    }
};

let boots = {
    name: "buty",
    price: "100",

    get priceIncTax() {
        return Number(this.price) * 1.2;
    }
}

hat.writeDetails = hat.writeDetails.bind(hat);
hat.writeDetails();
hat.price = 120;
hat.writeDetails();

console.log(`buty: ${boots.price}, ${boots.priceIncTax}`);
boots.price = "120";
console.log(`buty: ${boots.price}, ${boots.priceIncTax}`);

```

Po wprowadzeniu przedstawionych zmian wartością `this` dla metody `writeDetails()` będzie zawierający ją obiekt, niezależnie od sposobu wywołania funkcji.

Podsumowanie

W tym rozdziale przedstawiłem wprowadzenie do podstawowych funkcji systemu typów w JavaScriptcie. Te funkcje są często powodem dezorientacji, ponieważ działają odmiennie niż w innych językach programowania. Poznanie tych funkcji znacznie ułatwia pracę z TypeScriptem, ponieważ pokazują one dokładnie problemy rozwiązywane przez TypeScript. W następnym rozdziale przedstawię kolejne funkcje typów JavaScriptu, które okazują się użyteczne podczas poznawania TypeScriptu.

ROZDZIAŁ 4.



Wprowadzenie do języka JavaScript — część II

W tym rozdziale będę kontynuował omawianie związanych z typami JavaScriptu funkcji, które są ważne podczas programowania w języku TypeScript. Skoncentruję się przede wszystkim na oferowanej przez JavaScript obsługę obiektów, różnych sposobach ich definiowania, a także powiązaniu z klasami JavaScriptu. Zaprezentuję również funkcje przeznaczone do obsługi sekwencji wartości, kolekcje JavaScriptu oraz funkcje modułów pozwalających podzielić projekt na wiele plików JavaScriptu.

Przygotowanie projektu

W rozdziale będę kontynuował używanie projektu *primer* utworzonego w rozdziale 3. Zawartość pliku *index.js* w katalogu *primer* należy zastąpić kodem przedstawionym na listingu 4.1.

■ **Wskazówka** Przykładowy projekt dla tego rozdziału, a także projekty dla pozostałych rozdziałów książki zostały umieszczone w archiwum dostępnym pod adresem <https://ftp.helion.pl/przyklady/tys4o2.zip>.

Listing 4.1. Nowa zawartość pliku *index.js* w katalogu *primer*

```
let hat = {
  name: "czapka",
  price: 100,
  getPriceIncTax() {
    return Number(this.price) * 1.2;
  }
};

console.log(`czapka: ${hat.price}, ${hat.getPriceIncTax()}`);
```

W powłoce przejdź do katalogu *primer*, a następnie wydaj polecenie przedstawione na listingu 4.2, aby w ten sposób zacząć monitorowanie i uruchomić kod JavaScriptu.

Listing 4.2. Uruchamianie narzędzi programistycznych

```
$ npx nodemon index.js
```

Pakiet nodemon rozpocznie wykonywanie zawartości pliku *index.js* i wygeneruje następujące dane wyjściowe:

```
[nodemon] 2.0.7
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: *.*
[nodemon] starting `node index.js`
czapka: 100, 120
[nodemon] clean exit - waiting for changes before restart
```

Dziedziczenie obiektu JavaScriptu

Obiekt JavaScriptu ma łącze do innego obiektu nazywanego *prototypem*, po którym dziedziczy właściwości i metody. Skoro prototyp jest obiektem i ma własny prototyp, powstaje tym samym łańcuch dziedziczenia pozwalający na definiowanie skomplikowanych funkcji i ich spójne używanie.

Gdy obiekt jest tworzony za pomocą składni literału, np. jak obiekt *hat* na listingu 4.1, jego prototypem jest *Object*, czyli wbudowany obiekt dostarczany przez JavaScript. Zawiera on podstawowe funkcje dziedziczone przez wszystkie obiekty, w tym również metodę o nazwie *toString()*, której działanie polega na zwróceniu tekstowej reprezentacji obiektu, jak pokazałem na listingu 4.3.

Listing 4.3. Używanie obiektu w kodzie pliku *index.js* w katalogu *primer*

```
let hat = {
  name: "czapka",
  price: 100,
  getPriceIncTax() {
    return Number(this.price) * 1.2;
  }
};

console.log(`czapka: ${hat.price}, ${hat.getPriceIncTax()} `);
console.log(`toString: ${hat.toString()}`);
```

Pierwsze polecenie `console.log()` otrzymuje szablon ciągu tekstowego z właściwością `price` będącą jedną z właściwości obiektu *hat*. Z kolei drugie polecenie `console.log()` wywołuje metodę `toString()`. Żadna z właściwości obiektu *hat* nie ma nazwy `toString`, więc środowisko uruchomieniowe zwraca się ku prototypowi obiektu *hat*, czyli *Object*, który zawiera właściwość `toString`. W efekcie zostają wygenerowane następujące dane wyjściowe:

```
czapka: 100, 120
toString: [object Object]
```

Wynik działania metody `toString()` nie jest tutaj szczególnie użyteczny, ale ilustruje związek zachodzący między obiektem `hat` a jego prototypem, jak pokazałem na rysunku 4.1.



Rysunek 4.1. Obiekt i jego prototyp

Analizowanie i modyfikowanie prototypu obiektu

`Object` to prototyp większości obiektów, choć dostarcza również pewne metody używane bezpośrednio, a nie poprzez interfejs, i pozwalające na zebranie informacji o prototypach. W tabeli 4.1 wymieniałem najbardziej użyteczne z tych metod.

Tabela 4.1. Użyteczne metody obiektu

Metoda	Opis
<code>getPrototypeOf()</code>	Metoda zwraca prototyp obiektu
<code>setPrototypeOf()</code>	Metoda zmienia prototyp obiektu
<code>getOwnPropertyNames()</code>	Metoda zwraca nazwy właściwości obiektu

Kod przedstawiony na listingu 4.4 używa metody `getPrototypeOf()` do potwierdzenia, że dwa obiekty utworzone za pomocą składni literału współdzielą ten sam prototyp.

Listing 4.4. Porównywanie właściwości w kodzie pliku `index.js` w katalogu `primer`

```

let hat = {
  name: "czapka",
  price: 100,
  getPriceIncTax() {
    return Number(this.price) * 1.2;
  }
};

let boots = {
  name: "buty",
  price: 100,
  getPriceIncTax() {
    return Number(this.price) * 1.2;
  }
}

let hatPrototype = Object.getPrototypeOf(hat);
console.log(`Prototyp obiektu czapki: ${hatPrototype}`);

let bootsPrototype = Object.getPrototypeOf(boots);
console.log(`Prototyp obiektu butów: ${bootsPrototype}`);

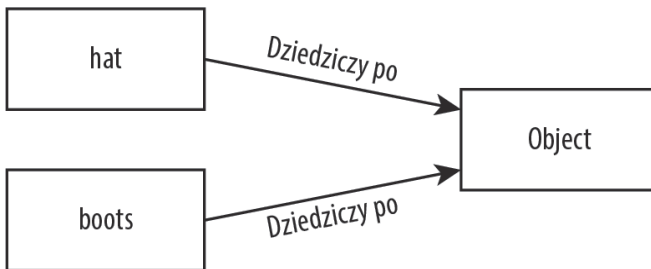
console.log(`Wspólny prototyp: ${ hatPrototype === bootsPrototype}`);
  
```

```
console.log(`czapka: ${hat.price}, ${hat.getPriceIncTax()} `);
console.log(`toString: ${hat.toString()}`);
```

Kod przedstawiony na tym listingu tworzy jeszcze inny obiekt, porównuje ich prototypy i generuje następujące dane wyjściowe:

```
Prototyp obiektu czapki: [object Object]
Prototyp obiektu butów: [object Object]
Wspólny prototyp: true
czapka: 100, 120
toString: [object Object]
```

Na podstawie tych danych wyjściowych wyraźnie widać, że obiekty `hat` i `boots` mają ten sam prototyp, jak pokazałem na rysunku 4.2.



Rysunek 4.2. Obiekty i ich wspólny prototyp

Skoro prototyp to zwykły obiekt JavaScriptu, nowe właściwości mogą być definiowane w prototypach, a nowe wartości mogą być przypisywane istniejącym właściwościom, jak pokazałem na listingu 4.5.

Listing 4.5. Zmiana właściwości prototypu w kodzie pliku `index.js` w katalogu `primer`

```
let hat = {
  name: "czapka",
  price: 100,
  getPriceIncTax() {
    return Number(this.price) * 1.2;
  }
};

let boots = {
  name: "buty",
  price: 100,
  getPriceIncTax() {
    return Number(this.price) * 1.2;
  }
}

let hatPrototype = Object.getPrototypeOf(hat);
hatPrototype.toString = function() {
  return `toString: nazwa: ${this.name}, cena: ${this.price}`;
}
```

```
console.log(hat.toString());
console.log(boots.toString());
```

Kod na listingu 4.5 przypisuje nową funkcję metodzie `toString()` za pomocą prototypu obiektu `hat`. Skoro obiekt zachowuje łącznie do swojego prototypu, nowa metoda `toString()` będzie możliwa do użycia również w obiekcie `boots`, jak widać w wygenerowanych danych wyjściowych:

```
toString: nazwa: czapka, cena: 100
toString: nazwa: buty, cena: 100
```

Tworzenie własnych właściwości

Zmiany w prototypie należy wprowadzać z zachowaniem dużej ostrożności, ponieważ mają one wpływ na wszystkie pozostałe obiekty aplikacji. Nowa funkcja `toString()` wykorzystana w kodzie na listingu 4.5 spowodowała wygenerowanie znacznie bardziej użytecznych danych wyjściowych dla obiektów `hat` i `boots`, ale przyjęte zostało założenie o istnieniu właściwości o nazwach `name` i `price`. Tak jednak nie jest w przypadku wywołania `toString()` w innych obiektach.

Znacznie lepsze podejście polega na utworzeniu prototypu przeznaczonego specjalnie dla tych obiektów, o których wiadomo, że mają właściwości `name` i `price`. Do tego celu można się posłużyć metodą `Object.setPrototypeOf()`, której przykład użycia zaprezentowałem na listingu 4.6.

Listing 4.6. *Używanie własnej właściwości w kodzie pliku `index.js` w katalogu `primer`*

```
let ProductProto = {
  toString: function() {
    return `toString: nazwa: ${this.name}, cena: ${this.price}`;
  }
}

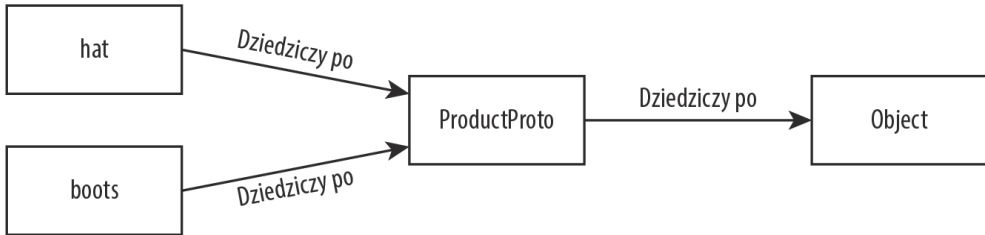
let hat = {
  name: "czapka",
  price: 100,
  getPriceIncTax() {
    return Number(this.price) * 1.2;
  }
};

let boots = {
  name: "buty",
  price: 100,
  getPriceIncTax() {
    return Number(this.price) * 1.2;
  }
}

Object.setPrototypeOf(hat, ProductProto);
Object.setPrototypeOf(boots, ProductProto);
```

```
console.log(hat.toString());
console.log(boots.toString());
```


Prototyp może być definiowany podobnie jak każdy inny obiekt. Na listingu 4.6 obiekt o nazwie `ProductProto` definiuje metodę `toString()` używaną jako prototyp dla obiektów `hat` i `boots`. Obiekt `ProductProto` jest podobny do każdego innego obiektu, więc także ma prototyp, którym jest `Object`, jak pokazałem na rysunku 4.3.



Rysunek 4.3. Łańcuch prototypów

Efektom jest powstanie łańcucha prototypów sprawdzanego przez JavaScript aż do momentu odnalezienia właściwości lub metody bądź też dotarcia do końca łańcucha. Kod na listingu 4.6 powoduje wygenerowanie następujących danych wyjściowych:

```

toString: nazwa: czapka, cena: 100
toString: nazwa: buty, cena: 100
  
```

Używanie funkcji konstruktora

Funkcja konstruktora jest używana do utworzenia nowego obiektu, skonfigurowania jego właściwości i przypisania jego prototypu — to wszystko odbywa się w pojedynczym kroku za pomocą słowa kluczowego `new`. Funkcje konstruktora mogą być używane do zagwarantowania spójnego tworzenia obiektów i stosowania dla nich prawidłowych prototypów, jak pokazałem na listingu 4.7.

Listing 4.7. Używanie funkcji konstruktora w kodzie pliku `index.js` w katalogu `primer`

```

let Product = function(name, price) {
  this.name = name;
  this.price = price;
}

Product.prototype.toString = function() {
  return `toString: nazwa: ${this.name}, cena: ${this.price}`;
}

let hat = new Product("czapka", 100);
let boots = new Product("buty", 100);

console.log(hat.toString());
console.log(boots.toString());
  
```

Funkcje konstruktora są wywoływane za pomocą słowa kluczowego `new`, po którym znajduje się funkcja lub jej nazwa zmiennej i argumenty przeznaczone do konfiguracji obiektu, np.:

```
...
let hat = new Product("czapka", 100);
...
```

Środowisko uruchomieniowe JavaScriptu tworzy nowy obiekt i używa go jako wartości `this` do wywołania funkcji konstruktora, dostarczając wartości argumentów jako parametry. Funkcja konstruktora może konfigurować właściwości obiektu za pomocą wartości `this` prowadzącej do nowo utworzonego obiektu.

```
...
let Product = function(name, price) {
  this.name = name;
  this.price = price;
}
...
```

Prototypem dla nowego obiektu jest obiekt zwrócony przez właściwość `prototype` funkcji konstruktora. Prowadzi to do zdefiniowania konstruktora przez dwa komponenty: funkcję używaną do konfiguracji właściwości obiektu oraz obiekt zwrócony przez właściwość `prototype`, wykorzystywany dla właściwości i metod, które powinny być współdzielone przez wszystkie obiekty tworzone przez konstruktor. Na listingu 4.7 właściwość `toString` została dodana do prototypu funkcji konstruktora i użyta do zdefiniowania metody:

```
...
Product.prototype.toString = function() {
  return `toString: nazwa: ${this.name}, cena: ${this.price}`;
}
...
```

Wynik działania jest dokładnie taki sam jak w poprzednim przykładzie, ale używanie funkcji konstruktora może pomóc zagwarantować spójne tworzenie obiektów i prawidłowe definiowanie ich właściwości.

Łączenie funkcji konstruktora

Używanie metody `setPrototypeOf()` do utworzenia łańcucha własnych właściwości jest łatwe, ale to samo zadanie z funkcjami konstruktora wymaga nieco więcej pracy, aby mieć pewność prawidłowej konfiguracji obiektów i umieszczenia odpowiednich właściwości w łańcuchu. Na listingu 4.8 przedstawiłem nową funkcję konstruktora przeznaczoną do utworzenia łańcucha z konstruktorem `Product`.

Listing 4.8. *Zmiana funkcji konstruktora w kodzie pliku `index.js` w katalogu `primer`*

```
let Product = function(name, price) {
  this.name = name;
  this.price = price;
}

Product.prototype.toString = function() {
  return `toString: nazwa: ${this.name}, cena: ${this.price}`;
}
```

```

let TaxedProduct = function(name, price, taxRate) {
    Product.call(this, name, price);
    this.taxRate = taxRate;
}
Object.setPrototypeOf(TaxedProduct.prototype, Product.prototype);

TaxedProduct.prototype.getPriceIncTax = function() {
    return Number(this.price) * this.taxRate;
}

TaxedProduct.prototype.toTaxString = function() {
    return `${this.toString()}, z podatkiem: ${this.getPriceIncTax()}`;
}

let hat = new TaxedProduct("czapka", 100, 1.2);
let boots = new Product("buty", 100);

console.log(hat.toTaxString());
console.log(boots.toString());

```

W celu przygotowania konstruktorów i ich prototypów w łańcuchu muszą być wykonane dwa kroki. Pierwszy krok to użycie metody `call()` do wywołania następnego konstruktora, aby nowe obiekty były tworzone prawidłowo. W omawianym przykładzie chcę, aby konstruktor `TaxedProduct()` został utworzony na podstawie konstruktora `Product`, więc użyłem metody `call()` w funkcji `Product`, co spowodowało dodanie jego właściwości do nowego obiektu.

```

...
Product.call(this, name, price);
...

```

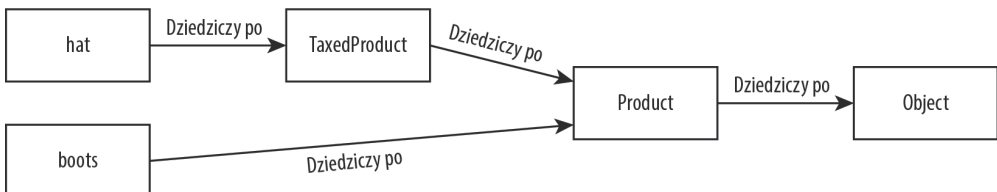
Metoda `call()` pozwala na przekazanie nowego obiektu następnemu konstruktorowi za pomocą jego wartości `this`. Drugim krokiem jest połączenie prototypów ze sobą.

```

...
Object.setPrototypeOf(TaxedProduct.prototype, Product.prototype);
...

```

Zwróć uwagę na to, że argumentami metody `setPrototypeOf()` są obiekty zwracane przez właściwości `prototype` konstruktora, a nie przez same funkcje. Połączenie prototypów gwarantuje, że środowisko uruchomieniowe JavaScriptu będzie podążało za łańcuchem podczas wyszukiwania właściwości nienależących do obiektu. Na rysunku 4.4 pokazałem nowy zestaw prototypów.



Rysunek 4.4. Znacznie bardziej złożony łańcuch prototypów

Prototyp `TaxedProduct` definiuje metodę `toTaxString()` wywołującą `toString()`, która będzie znaleziona przez środowisko uruchomieniowe JavaScriptu w prototypie `Product`, a kod przedstawiony na listingu 4.8 wygeneruje następujące dane wyjściowe:

```
toString: nazwa: czapka, cena: 100, z podatkiem: 120
toString: nazwa: buty, cena: 100
```

Uzyskiwanie dostępu do nadpisanych metod prototypu

Prototyp może nadpisać właściwość lub metodę przez używanie tej samej nazwy, która została zdefiniowana w łańcuchu. W JavaScriptcie nosi to nazwę *przesłaniania* i wykorzystuje cechę związaną ze sposobem, w jaki środowisko uruchomieniowe JavaScriptu podąża za łańcuchem.

Trzeba zachować dużą ostrożność podczas stosowania nadpisanych metod, do których dostęp musi się odbywać za pomocą definiujących je prototypów. Prototyp `TaxedProduct` może definiować metodę `toString()` nadpisującą metodę zdefiniowaną przez prototyp `Product` i wywoływać nadpisaną metodę przez bezpośrednie uzyskanie dostępu za pomocą prototypu i użycie wywołania `call()` do zdefiniowania wartości `this`.

```
...
TaxedProduct.prototype.toString = function() {
    let chainResult = Product.prototype.toString.call(this);
    return `${chainResult}, z podatkiem: ${this.getPriceIncTax()}`;
}
...
```

Ta metoda pobiera wynik z metody `toString()` prototypu `Product` i łączy go z dodatkowymi danymi w ciągu tekstowym szablonu.

Sprawdzanie typu prototypu

Operator `instanceof` jest używany do ustalenia, czy prototyp konstruktora jest częścią łańcucha określonego obiektu, jak pokazałem na listingu 4.9.

Listing 4.9. Sprawdzanie prototypów w kodzie pliku `index.js` w katalogu `primer`

```
let Product = function(name, price) {
    this.name = name;
    this.price = price;
}

Product.prototype.toString = function() {
    return `toString: nazwa: ${this.name}, cena: ${this.price}`;
}

let TaxedProduct = function(name, price, taxRate) {
    Product.call(this, name, price);
    this.taxRate = taxRate;
}

Object.setPrototypeOf(TaxedProduct.prototype, Product.prototype);
```

```

TaxedProduct.prototype.getPriceIncTax = function() {
    return Number(this.price) * this.taxRate;
}

TaxedProduct.prototype.toTaxString = function() {
    return `${this.toString()}, z podatkiem: ${this.getPriceIncTax()}`;
}

let hat = new TaxedProduct("czapka", 100, 1.2);
let boots = new Product("buty", 100);

console.log(hat.toTaxString());
console.log(boots.toString());
console.log(`hat i TaxedProduct: ${ hat instanceof TaxedProduct}`);
console.log(`hat i Product: ${ hat instanceof Product}`);
console.log(`boots i TaxedProduct: ${ boots instanceof TaxedProduct}`);
console.log(`boots i Product: ${ boots instanceof Product}`);

```

Nowe polecenia używają operatora `instanceof` do ustalenia, czy prototypy funkcji konstruktora `TaxedProduct` i `Product` znajdują się w łańcuchu obiektów `hat` i `boots`. Kod przedstawiony na listingu 4.9 powoduje wygenerowanie następujących danych wyjściowych:

```

toString: nazwa: czapka, cena: 100, z podatkiem: 120
toString: nazwa: buty, cena: 100
hat i TaxedProduct: true
hat i Product: true
boots i TaxedProduct: false
boots i Product: true

```

■ **Wskazówka** Zwróć uwagę na użycie operatora `instanceof` z funkcją konstruktora. Metoda `Object.isPrototypeOf()` jest używana bezpośrednio z prototypami, co może być użyteczne, jeśli nie korzystasz z konstruktorów.

Definiowanie statycznych właściwości i metod

Właściwości i metody definiowane w funkcji konstruktora są często określane mianem *statycznych*, co oznacza, że są dostępne za pomocą konstruktora, a nie poszczególnych obiektów utworzonych przez ten konstruktor (jest to przeciwieństwo *właściwości egzemplarza* dostępnych poprzez obiekt). Metody `Object.setPrototypeOf()` i `Object.getPrototypeOf()` są dobrymi przykładami metod statycznych. Kod przedstawiony na listingu 4.10 upraszcza przykład w celu zachowania zwięzłości i pokazuje używanie metody statycznej.

Listing 4.10. Definiowanie metody statycznej w kodzie pliku `index.js` w katalogu `primer`

```

let Product = function(name, price) {
    this.name = name;
    this.price = price;
}

```

```
Product.prototype.toString = function() {
    return `toString: nazwa: ${this.name}, cena: ${this.price}`;
}
```

```
Product.process = (...products) =>
    products.forEach(p => console.log(p.toString()));
```

```
Product.process(new Product("czapka", 100, 1.2), new Product("buty", 100));
```

Metoda statyczna `process()` została zdefiniowana przez dodanie nowej właściwości do funkcji obiektu `Product` i przypisanie jej funkcji. Nie zapominaj, że funkcje JavaScriptu to obiekty, a właściwości mogą być dowolnie dodawane do i usuwane z obiektów. Metoda `process()` definiuje parametr resztowy, używa metody `forEach()` w celu wywołania metody `toString()` dla każdego otrzymanego obiektu i wyświetla wynik w konsoli. Kod przedstawiony na listingu 4.10 powoduje wygenerowanie następujących danych wyjściowych:

```
toString: nazwa: czapka, cena: 100
toString: nazwa: buty, cena: 100
```

Używanie klas JavaScriptu

Klasy JavaScriptu zostały dodane do języka, aby ułatwić przejście z innych popularnych języków programowania. Pod maską klasy JavaScriptu są implementowane za pomocą prototypów, więc różnią się od klas stosowanych w innych językach, takich jak C# i Java. Na listingu 4.11 usunąłem z przykładu konstruktory i prototypy, a wprowadziłem klasę `Product`.

Listing 4.11. Definiowanie klasy w kodzie pliku `index.js` w katalogu `primer`

```
class Product {
    constructor(name, price) {
        this.name = name;
        this.price = price;
    }

    toString() {
        return `toString: nazwa: ${this.name}, cena: ${this.price}`;
    }
}

let hat = new Product("czapka", 100);
let boots = new Product("buty", 100);

console.log(hat.toString());
console.log(boots.toString());
```

Klasa jest definiowana za pomocą słowa kluczowego `class`, po którym znajduje się nazwa klasy. Wprawdzie składnia klas może się wydawać znajoma, ale są one konwertowane na działający pod maską system prototypów JavaScriptu, który omówiłem we wcześniejszej części rozdziału.

Utworzenie nowego obiektu na podstawie klasy odbywa się za pomocą słowa kluczowego `new`. Środowisko uruchomieniowe JavaScriptu tworzy nowy obiekt, a następnie wywołuje funkcję `constructor()` klasy otrzymującą nowy obiekt poprzez wartość `this`. Wymieniona funkcja jest odpowiedzialna za zdefiniowanie właściwości obiektu. Metody definiowane przez klasę są dodawane do prototypu przypisywanego obiektom utworzonym na podstawie danej klasy. Kod na listingu 4.11 powoduje wygenerowanie następujących danych wyjściowych:

```
toString: nazwa: czapka, cena: 100
toString: nazwa: buty, cena: 100
```

Używanie dziedziczenia w klasach

Klasa może dziedziczyć funkcje za pomocą słowa kluczowego `extends` oraz wywoływać konstruktor i metody klasy nadrzędnej za pomocą słowa kluczowego `super`, jak widać w przykładzie na listingu 4.12.

Listing 4.12. Rozszerzanie klasy w kodzie pliku `index.js` w katalogu `primer`

```
class Product {
  constructor(name, price) {
    this.name = name;
    this.price = price;
  }

  toString() {
    return `toString: nazwa: ${this.name}, cena: ${this.price}`;
  }
}

class TaxedProduct extends Product {

  constructor(name, price, taxRate = 1.2) {
    super(name, price);
    this.taxRate = taxRate;
  }

  getPriceIncTax() {
    return Number(this.price) * this.taxRate;
  }

  toString() {
    let chainResult = super.toString();
    return `${chainResult}, z podatkiem: ${this.getPriceIncTax()}`;
  }
}

let hat = new TaxedProduct("czapka", 100);
let boots = new TaxedProduct("buty", 100, 1.3);

console.log(hat.toString());
console.log(boots.toString());
```

Klasa deklaruje swoją klasę nadrzędną za pomocą słowa kluczowego `extends`. W omawianym przykładzie klasa `TaxedProduct` używa słowa kluczowego `extend` do dziedziczenia po klasie `Product`. Słowo kluczowe `super` zostało w konstruktorze użyte do wywołania konstruktora klasy nadrzędnej, co jest odpowiednikiem łączenia funkcji `constructor()`.

```
...
constructor(name, price, taxRate = 1.2) {
  super(name, price);
  this.taxRate = taxRate;
}
...
```

Słowo kluczowe `super` musi być użyte przed słowem kluczowym `this` i ogólnie rzecz biorąc, musi być pierwszym poleceniem konstruktora. Słowo kluczowe `super` można również wykorzystać w celu uzyskania dostępu do właściwości i metod, jak pokazałem w kolejnym fragmencie kodu:

```
...
toString() {
  let chainResult = super.toString();
  return `${chainResult}, z podatkiem: ${this.getPriceIncTax()}`;
}
...
```

Metoda `toString()` zdefiniowana przez klasę `TaxedProduct` wywołała metodę `toString()` klasy nadrzędnej, co jest odpowiednikiem nadpisywania metod prototypu. Kod przedstawiony na listingu 4.12 powoduje wygenerowanie następujących danych wyjściowych:

```
toString: nazwa: czapka, cena: 100, z podatkiem: 120
toString: nazwa: buty, cena: 100, z podatkiem: 130
```

Definiowanie metody statycznej

Słowo kluczowe `static` jest stosowane podczas tworzenia metody statycznej dostępnej poprzez klasę, a nie za pomocą obiektu danej klasy. Spójrz na przykładowy program przedstawiony na listingu 4.13.

Listing 4.13. Definiowanie metody statycznej w kodzie pliku `index.js` w katalogu `primer`

```
class Product {
  constructor(name, price) {
    this.name = name;
    this.price = price;
  }

  toString() {
    return `toString: nazwa: ${this.name}, cena: ${this.price}`;
  }
}

class TaxedProduct extends Product {

  constructor(name, price, taxRate = 1.2) {
    super(name, price);
```



```

    this.taxRate = taxRate;
  }

  getPriceIncTax() {
    return Number(this.price) * this.taxRate;
  }

  toString() {
    let chainResult = super.toString();
    return `${chainResult}, z podatkiem: ${this.getPriceIncTax()}`;
  }

  static process(...products) {
    products.forEach(p => console.log(p.toString()));
  }
}

```

```

TaxedProduct.process(new TaxedProduct("czapka", 100, 1.2),
  new TaxedProduct("buty", 100));

```

Słowo kluczowe `static` jest używane w metodzie `process()` zdefiniowanej przez klasę `TaxedProduct` i dostępnej jako `TaxedProduct.process()`. Kod przedstawiony na listingu 4.13 powoduje wygenerowanie następujących danych wyjściowych:

```

toString: nazwa: czapka, cena: 100, z podatkiem: 120
toString: nazwa: buty, cena: 100, z podatkiem: 120

```

Używanie iteratorów i generatorów

Iterator to obiekt zwracający sekwencję wartości. Wprawdzie iteratory są używane z kolekcjami, które będą omówione w dalszej części rozdziału, ale równie użyteczne mogą być w przypadku ich wykorzystania poza kolekcjami. Iterator definiuje funkcję o nazwie `next()` zwracającą obiekt z właściwościami `value` i `done`. Właściwość `value` zwraca następną wartość w sekwencji, a właściwość `done` otrzymuje wartość `true` po dotarciu do końca sekwencji. Na listingu 4.14 przedstawiłem przykład definicji i użycia iteratora.

Listing 4.14. *Używanie iteratora w kodzie pliku `index.js` w katalogu `primer`*

```

class Product {
  constructor(name, price) {
    this.name = name;
    this.price = price;
  }

  toString() {
    return `toString: nazwa: ${this.name}, cena: ${this.price}`;
  }
}

function createProductIterator() {
  const hat = new Product("czapka", 100);

```

```

const boots = new Product("buty", 100);
const umbrella = new Product("parasol", 23);

let lastVal;

return {
  next() {
    switch (lastVal) {
      case undefined:
        lastVal = hat;
        return { value: hat, done: false };
      case hat:
        lastVal = boots;
        return { value: boots, done: false };
      case boots:
        lastVal = umbrella;
        return { value: umbrella, done: false };
      case umbrella:
        return { value: undefined, done: true };
    }
  }
}

let iterator = createProductIterator();
let result = iterator.next();
while (!result.done) {
  console.log(result.value.toString());
  result = iterator.next();
}

```

Funkcja `createProductIterator()` zwraca obiekt definiujący funkcję `next()`. W trakcie każdego wywołania metody `next()` zwracany jest inny obiekt typu `Product`, a następnie po wyczerpaniu zbioru obiektów będzie zwrócony obiekt, którego właściwość `done` ma wartość `true` wskazującą koniec danych. Pętla `while` została użyta do przetwarzania danych iteratora i wywołuje funkcję `next()` po przetworzeniu każdego obiektu. Kod przedstawiony na listingu 4.14 powoduje wygenerowanie następujących danych wyjściowych:

```

toString: nazwa: czapka, cena: 100
toString: nazwa: buty, cena: 100
toString: nazwa: parasol, cena: 23

```

Używanie generatora

Tworzenie iteratorów może być kłopotliwe, ponieważ kod musi monitorować stan danych i śledzić bieżące położenie w sekwencji w trakcie każdego wywołania funkcji. Znacznie prostsze podejście polega na użyciu generatora, czyli funkcji wywoływanej tylko raz i wykorzystującej słowo kluczowe `yield` do generowania wartości w sekwencji, jak pokazałem na listingu 4.15.

Listing 4.15. Użycie generatora w kodzie pliku *index.js* w katalogu *primer*

```

class Product {
  constructor(name, price) {
    this.name = name;
    this.price = price;
  }

  toString() {
    return `toString: nazwa: ${this.name}, cena: ${this.price}`;
  }
}

function* createProductIterator() {
  yield new Product("czapka", 100);
  yield new Product("buty", 100);
  yield new Product("parasol", 23);
}

let iterator = createProductIterator();
let result = iterator.next();
while (!result.done) {
  console.log(result.value.toString());
  result = iterator.next();
}

```

Funkcja generatora jest oznaczana gwiazdką:

```

...
function* createProductIterator() {
...

```

Generatory są używane w dokładnie taki sam sposób jak iteratory. Środowisko uruchomieniowe JavaScriptu tworzy funkcję `next()` i wykonuje funkcję generatora aż do chwili dotarcia do słowa kluczowego `yield` dostarczającego wartość w sekwencji. Wykonywanie funkcji generatora odbywa się w trakcie każdego wywołania funkcji `next()`. Gdy nie ma już żadnego polecenia `yield` do wykonania, następuje automatyczne utworzenie obiektu, którego właściwość `done` ma wartość `true`.

Generator może być używany z operatorem rozwinęcia, co pozwala na stosowanie sekwencji jako zbioru parametrów funkcji lub w celu wypełnienia tablicy elementami, jak pokazałem na listingu 4.16.

Listing 4.16. Używanie operatora rozwinęcia w kodzie pliku *index.js* w katalogu *primer*

```

class Product {
  constructor(name, price) {
    this.name = name;
    this.price = price;
  }

  toString() {
    return `toString: nazwa: ${this.name}, cena: ${this.price}`;
  }
}

```

```
function* createProductIterator() {
  yield new Product("czapka", 100);
  yield new Product("buty", 100);
  yield new Product("parasol", 23);
}
```

```
[...createProductIterator()].forEach(p => console.log(p.toString()));
```

Nowe polecenie na listingu 4.16 używa sekwencji wartości z generatora i umieszcza je w tablicy za pomocą metody `forEach()`. Kod przedstawiony na listingu 4.16 powoduje wygenerowanie następujących danych wyjściowych:

```
toString: nazwa: czapka, cena: 100
toString: nazwa: buty, cena: 100
toString: nazwa: parasol, cena: 23
```

Definiowanie obiektów pozwalających na iterację

Samodzielne funkcje dla iteratorów i generatorów mogą być użyteczne, ale często można się spotkać z wymaganiem, aby obiekt dostarczał sekwencję jako część większej funkcjonalności. Na listingu 4.17 pokazałem zdefiniowanie obiektu grupującego powiązane ze sobą elementy danych i dostarczającego generator pozwalający na umieszczenie elementów w sekwencji.

Listing 4.17. Definiowanie obiektu z sekwencją w kodzie pliku `index.js` w katalogu `primer`

```
class Product {
  constructor(name, price) {
    this.name = name;
    this.price = price;
  }

  toString() {
    return `toString: nazwa: ${this.name}, cena: ${this.price}`;
  }
}

class GiftPack {
  constructor(name, prod1, prod2, prod3) {
    this.name = name;
    this.prod1 = prod1;
    this.prod2 = prod2;
    this.prod3 = prod3;
  }

  getTotalPrice() {
    return [this.prod1, this.prod2, this.prod3]
      .reduce((total, p) => total + p.price, 0);
  }

  *getGenerator() {
    yield this.prod1;
    yield this.prod2;
  }
}
```

```

        yield this.prod3;
    }
}

let winter = new GiftPack("zima", new Product("czapka", 100),
    new Product("buty", 80), new Product("rękawiczki", 23));

console.log(`Wartość całkowita: ${ winter.getTotalPrice() }`);

[...winter.getGenerator()].forEach(p => console.log(`Produkt: ${ p }`));

```

Klasa `GiftPack` monitoruje zbiór powiązanych ze sobą produktów. Jedną z metod zdefiniowanych przez `GiftPack` jest `getGenerator()` — generator dostarczający produkty.

■ **Wskazówka** Nazwa metody generatora ma prefiks w postaci gwiazdki.

Wprowadź takie rozwiązanie się sprawdza, ale składnia używania iteratora jest niewygodna, ponieważ metoda `getGenerator()` musi być wyraźnie wywołana, jak pokazałem w kolejnym fragmencie kodu:

```

...
[...winter.getGenerator()].forEach(p => console.log(`Produkt: ${ p }`));
...

```

Znacznie bardziej eleganckie podejście polega na użyciu specjalnej nazwy metody wskazującej środowisku uruchomieniowemu JavaScriptu, że dana metoda domyślnie obsługuje iterację w obiekcie. Spójrz na przykładowy program przedstawiony na listingu 4.18.

Listing 4.18. Definiowanie domyślnej metody iteratora w kodzie pliku `index.js` w katalogu `primer`

```

class Product {
    constructor(name, price) {
        this.name = name;
        this.price = price;
    }

    toString() {
        return `toString: nazwa: ${this.name}, cena: ${this.price}`;
    }
}

class GiftPack {
    constructor(name, prod1, prod2, prod3) {
        this.name = name;
        this.prod1 = prod1;
        this.prod2 = prod2;
        this.prod3 = prod3;
    }

    getTotalPrice() {
        return [this.prod1, this.prod2, this.prod3]
            .reduce((total, p) => total + p.price, 0);
    }
}

```

```

    }

    *[Symbol.iterator]() {
        yield this.prod1;
        yield this.prod2;
        yield this.prod3;
    }
}

let winter = new GiftPack("zima", new Product("czapka", 100),
    new Product("buty", 80), new Product("rękawiczki", 23));

console.log(`Wartość całkowita: ${ winter.getTotalPrice() }`);

[...winter].forEach(p => console.log(`Produkt: ${ p }`));

```

Właściwość `Symbol.iterator` jest używana do określenia domyślnego iteratora obiektu. (W tym momencie nie zastanawiaj się nad słowem kluczowym `Symbol` — jest używane w typach podstawowych JavaScriptu, a jego przeznaczenie poznasz w następnym podrozdziale). Używając wartości `Symbol.iterator` jako nazwy generatora, można przeprowadzić bezpośrednią iterację obiektu, np.:

```

...
[...winter].forEach(p => console.log(`Produkt: ${ p }`));
...

```

Nie trzeba już wywoływać metody w celu otrzymania generatora. Takie rozwiązanie jest bardziej przejrzyste i prowadzi do powstania bardziej eleganckiego kodu źródłowego.

Używanie kolekcji JavaScriptu

Tradycyjnie kolekcje danych w JavaScriptcie są zarządzane za pomocą obiektów i tablic, gdzie obiekty są używane do przechowywania danych według kluczy, a tablice przechowują dane według indeksów. JavaScript oferuje dedykowane obiekty kolekcji zapewniające znacznie lepszą strukturę, choć jednocześnie charakteryzujące się mniejszą elastycznością, jak to dokładnie wyjaśnię w tym podrozdziale.

Sortowanie danych według klucza przy użyciu obiektu

Obiekt może zostać użyty jako kolekcja. W takim przypadku każda właściwość jest parą klucz-wartość, przy czym kluczem jest nazwa właściwości, jak pokazałem na listingu 4.19.

Listing 4.19. *Używanie obiektu jako kolekcji w kodzie pliku `index.js` w katalogu `primer`*

```

class Product {
    constructor(name, price) {
        this.name = name;
        this.price = price;
    }
}

```

```

    toString() {
        return `toString: nazwa: ${this.name}, cena: ${this.price}`;
    }
}

let data = {
    hat: new Product("czapka", 100)
}

data.boots = new Product("buty", 100);

Object.keys(data).forEach(key => console.log(data[key].toString()));

```

W omawianym przykładzie obiekt o nazwie `data` został użyty do zebrania obiektów `Product`. Nowe wartości mogą być dodawane do kolekcji przez definiowanie nowych właściwości, np.:

```

...
data.boots = new Product("buty", 100);
...

```

Klasa `Object` oferuje użyteczne metody przeznaczone do pobierania zbioru kluczy lub wartości z obiektu. Krótkie omówienie tych metod znajdziesz w tabeli 4.2.

Tabela 4.2. Metody obiektu przeznaczone do pracy z kluczami i wartościami

Nazwa	Opis
<code>Object.keys(obiekt)</code>	Metoda zwraca tablicę zawierającą nazwy właściwości zdefiniowanych przez obiekt
<code>Object.values(obiekt)</code>	Metoda zwraca tablicę zawierającą wartości właściwości zdefiniowanych przez obiekt

W programie przedstawionym na listingu 4.19 metoda `Object.keys()` została wykorzystana do pobrania tablicy zawierającej nazwy właściwości zdefiniowanych przez obiekt `data`. Do pobrania wartości użyto metody `forEach()`. Podczas przypisywania nazwy właściwości zmiennej odpowiadającą jej wartość można pobrać za pomocą nawiasu kwadratowego, jak pokazałem w kolejnym fragmencie kodu:

```

...
Object.keys(data).forEach(key => console.log(data[key].toString()));
...

```

Zawartość nawiasu kwadratowego jest obliczana jak wyrażenie, a podanie nazwy zmiennej jako klucza powoduje zwrot jej wartości. Kod przedstawiony na listingu 4.19 powoduje wygenerowanie następujących danych wyjściowych:

```

toString: nazwa: czapka, cena: 100
toString: nazwa: buty, cena: 100

```

Sortowanie danych według klucza przy użyciu obiektu Map

Obiektu można bardzo łatwo używać w charakterze prostej kolekcji, choć wiąże się to z pewnymi ograniczeniami, np. kluczem może być jedynie wartość w postaci ciągu tekstowego. JavaScript oferuje również klasę Map, której obiekty są przeznaczone do przechowywania danych o kluczach dowolnego typu, o czym możesz się przekonać, analizując kod na listingu 4.20.

Listing 4.20. Użycie obiektu Map w kodzie pliku index.js w katalogu primer

```
class Product {
  constructor(name, price) {
    this.name = name;
    this.price = price;
  }

  toString() {
    return `toString: nazwa: ${this.name}, cena: ${this.price}`;
  }
}

let data = new Map();
data.set("hat", new Product("czapka", 100));
data.set("boots", new Product("buty", 100));

[...data.keys()].forEach(key => console.log(data.get(key).toString()));
```

API dostarczane przez klasę Map pozwala na przechowywanie i pobieranie elementów, a iteratory są dostępne dla kluczy i wartości. W tabeli 4.3 zostały wymienione najczęściej używane metody tej klasy.

Używanie wartości typu Symbol jako kluczy w obiekcie Map

Największą zaletą używania obiektu Map jest to, że dowolna wartość może być kluczem — dotyczy to również wartości typu Symbol. Każda wartość typu Symbol jest unikatowa, niemodyfikowalna i idealnie dopasowana jako identyfikator obiektu. Na listingu 4.21 przedstawiłem przykład zdefiniowania nowego obiektu typu Map używającego kluczy w postaci wartości typu Symbol.

Tabela 4.3. Użyteczne metody klasy Map

Nazwa metody	Opis
set(klucz, wartość)	Metoda przechowuje wartość z określonym kluczem
get(klucz)	Metoda wyszukuje wartości przechowywane w określonym kluczu
keys()	Metoda zwraca iterator dla kluczy w obiekcie Map
values()	Metoda zwraca iterator dla wartości w obiekcie Map
entries()	Metoda zwraca iterator dla par klucz-wartość w obiekcie Map, a każda para ma postać tablicy zawierającej klucz i wartość. Jest to iterator domyślny dla obiektu Map

-
- **Uwaga** Wartość typu `Symbol` może być użyteczna, choć jednocześnie praca z nią będzie trudna, ponieważ nie jest czytelna dla człowieka, a jej utworzenie i obsługa wymagają dużej ostrożności. Więcej informacji na ten temat znajdziesz w dokumentacji zamieszczonej na stronie https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Symbol.
-

Listing 4.21. Używanie wartości typu `Symbol` jako kluczy w kodzie pliku `index.js` w katalogu `primer`

```
class Product {
  constructor(name, price) {
    this.id = Symbol();
    this.name = name;
    this.price = price;
  }
}

class Supplier {
  constructor(name, productids) {
    this.name = name;
    this.productids = productids;
  }
}

let acmeProducts = [new Product("czapka", 100), new Product("buty", 100)];
let zoomProducts = [new Product("czapka", 100), new Product("buty", 100)];

let products = new Map();
[...acmeProducts, ...zoomProducts].forEach(p => products.set(p.id, p));
let suppliers = new Map();
suppliers.set("acme", new Supplier("Acme Co", acmeProducts.map(p => p.id)));
suppliers.set("zoom", new Supplier("Zoom Shoes", zoomProducts.map(p => p.id)));

suppliers.get("acme").productids.forEach(id =>
  console.log(`nazwa: ${products.get(id).name}`));
```

Korzyścią wynikającą z użycia wartości typu `Symbol` jako kluczy jest brak niebezpieczeństwa kolizji dwóch kluczy, co może się zdarzyć w sytuacji, gdy klucze są tworzone na podstawie pewnych cech wartości. W omawianym przykładzie kluczem jest wartość `Product.name` — przedstawia dwa obiekty przechowywane w tym samym kluczu, a jeden zastępuje drugi. Tutaj każdy obiekt `Product` ma właściwość `id`, której w konstruktorze została przypisana wartość typu `Symbol` używana do przechowywania obiektu w egzemplarzu typu `Map`. Stosowanie wartości typu `Symbol` pozwala na przechowywanie obiektów mających identyczne właściwości `name` i `price` oraz pobieranie ich bez żadnych problemów. Kod przedstawiony na listingu 4.21 powoduje wygenerowanie następujących danych wyjściowych:

```
nazwa: czapka
nazwa: buty
```

Przechowywanie danych według indeksu

W rozdziale 3. miałeś okazję zobaczyć, jak dane mogą być przechowywane w tablicy. JavaScript oferuje również zbiór (typ Set) pozwalający na przechowywanie danych według indeksu. Charakteryzuje się on optymalizacją wydajności działania i — co jest najbardziej użyteczne — przechowywaniem jedynie unikatowych wartości, jak pokazałem na listingu 4.22.

Listing 4.22. Używanie zbioru w kodzie pliku *index.js* w katalogu *primer*

```
class Product {
  constructor(name, price) {
    this.id = Symbol();
    this.name = name;
    this.price = price;
  }
}

let product = new Product("czapka", 100);

let productArray = [];
let productSet = new Set();

for (let i = 0; i < 5; i++) {
  productArray.push(product);
  productSet.add(product);
}

console.log(`Wielkość tablicy: ${productArray.length}`);
console.log(`Wielkość zbioru: ${productSet.size}`);
```

W tym przykładzie ten sam obiekt `Product` został pięciokrotnie dodany do tablicy i zbioru, a następnie kod wyświetlił wielkość tablicy i zbioru, generując następujące dane wyjściowe:

```
Wielkość tablicy: 5
Wielkość zbioru: 1
```

W moich projektach konieczność umożliwienia lub uniemożliwienia powielania wartości jest czynnikiem decydującym w wyborze między zbiorem a tablicą. API dostarczany przez zbiór oferuje funkcjonalność podobną do istniejącej w tablicy. W tabeli 4.4 wymieniłem najużyteczniejsze metody przeznaczone do pracy ze zbiorem.

Tabela 4.4. Użyteczne metody podczas pracy ze zbiorem

Nazwa metody	Opis
<code>add(wartość)</code>	Metoda dodaje wartość do zbioru
<code>entries()</code>	Wartość zwraca iterator dla elementów zbioru w kolejności ich dodawania
<code>has(wartość)</code>	Wartość zwraca wartość <code>true</code> , gdy zbiór zawiera określoną wartość
<code>forEach(wywołanieZwrotne)</code>	Metoda wywołuje funkcję dla każdej wartości w zbiorze

Używanie modułów

Większość aplikacji jest zbyt skomplikowana, aby ich kod mógł być umieszczony w pojedynczym pliku. JavaScript obsługuje tzw. *moduły* pozwalające podzielić kod na fragmenty łatwiejsze w zarządzaniu. Istnieją konkurujące ze sobą podejścia w zakresie definiowania i stosowania modułów. Skoncentruję się na podejściu zdefiniowanym w specyfikacji JavaScriptu, ponieważ jest ono szeroko stosowane w popularnych narzędziach programowania JavaScriptu oraz we frameworkach aplikacji.

Node.js oferuje obsługę dla modułów, choć w sposób nieco odmienny od stosowanego przez TypeScript i przedstawionego w dalszych rozdziałach książki. Aby obejść to ograniczenie, należy zatrzymać proces nodemon uruchomiony przez kod przedstawiony na listingu 4.2, a następnie z poziomu katalogu *primer* w powłoce wydać polecenie pokazane na listingu 4.23, które powoduje zainstalowanie pakietu o nazwie *esm* przeznaczonego do pracy z modułami.

Listing 4.23. Dodawanie pakietu esm

```
$ npm install esm@3.2.25
```

Po zainstalowaniu pakietu z poziomu katalogu *primer* w powłoce należy wydać polecenie pokazane na listingu 4.24.

Listing 4.24. Uruchamianie narzędzi programistycznych

```
$ npx nodemon --require esm index.js
```

Pakiet nodemon zostanie uruchomiony i wygeneruje następujące dane wyjściowe:

```
[nodemon] 2.0.7
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: *.*
[nodemon] starting `node --require esm index.js`
Wielkość tablicy: 5
Wielkość zbioru: 1
[nodemon] clean exit - waiting for changes before restart
```

Tworzenie modułu JavaScriptu

Każdy moduł JavaScriptu znajduje się w oddzielnym pliku. Aby zdefiniować przykładowy moduł, zacznij od utworzenia w katalogu *primer* pliku o nazwie *tax.js* z kodem przedstawionym na listingu 4.25.

Listing 4.25. Zawartość pliku tax.js w katalogu primer

```
export default function(price) {
  return Number(price) * 1.2;
}
```

Zdefiniowana w pliku *tax.js* funkcja otrzymuje wartość *price*, a następnie zwiększa ją o wysokość podatku wynoszącego w omawianym przykładzie 20%. Ta funkcja sama w sobie jest prosta, a jej słowa kluczowe `export` i `default` mają ważne znaczenie. Słowo kluczowe `export` służy do określenia funkcjonalności udostępnianej poza modułem. Domyślnie zawartość pliku JavaScriptu jest prywatna i konieczne jest wyraźne użycie słowa kluczowego `export`, zanim będzie można wykorzystać ten kod w pozostałej części aplikacji. Z kolei słowo kluczowe `default` jest używane, gdy moduł zawiera pojedynczą funkcję, np. jak na listingu 4.25. Razem słowa kluczowe `export` i `default` są stosowane w celu określenia, że jedyna funkcja w pliku *tax.js* jest dostępna do wykorzystania w pozostałej części aplikacji.

Używanie modułu JavaScriptu

Kolejnym słowem kluczowym JavaScriptu wymaganym w celu użycia modułu jest `import`. Na listingu 4.26 pokazałem przykład wykorzystania słowa kluczowego `import` w pliku *index.js*, aby można było wywołać funkcję zdefiniowaną w tym pliku.

Listing 4.26. Używanie modułu w kodzie pliku *index.js* w katalogu *primer*

```
import calcTax from "./tax";

class Product {
  constructor(name, price) {
    this.id = Symbol();
    this.name = name;
    this.price = price;
  }
}

let product = new Product("czapka", 100);
let taxedPrice = calcTax(product.price);
console.log(`nazwa: ${ product.name }, cena wraz z podatkiem: ${taxedPrice}`);
```

Słowo kluczowe `import` jest używane do zadeklarowania zależności od modułu. To słowo kluczowe może być stosowane na wiele różnych sposobów, ale przedstawiony tutaj format jest najczęściej spotykany podczas pracy z modułami utworzonymi w projekcie.

Po słowie kluczowym `import` znajduje się identyfikator, czyli nazwa, pod którą będzie znana funkcja modułu. W omawianym przykładzie tym identyfikatorem jest `calcTax`. Po słowie kluczowym `from` znajduje się identyfikator i dalej położenie modułu. Trzeba koniecznie zwracać uwagę na to położenie, ponieważ różne formaty jego wskazania mają odmienne efekty, jak to wyjaśniłem w ramce „Położenie modułu”.

W trakcie kompilacji środowisko uruchomieniowe JavaScriptu wykryje polecenie `import` i zaimportuje zawartość pliku *tax.js*. Identyfikator użyty w poleceniu `import` można wykorzystać w celu uzyskania dostępu do funkcji modułu dokładnie tak samo, jak są stosowane funkcje zdefiniowane lokalnie.

```
...
let taxedPrice = calcTax(product.price);
...
```

Po uruchomieniu omawianego kodu następuje obliczenie wartości przypisywanej zmiennej `taxPrice` z wykorzystaniem funkcji zdefiniowanej w pliku `tax.js`. W efekcie zostaną wygenerowane następujące dane wyjściowe:

```
nazwa: czapka, cena wraz z podatkiem: 120
```

Położenie modułu

Położenie modułu określa to, gdzie środowisko uruchomieniowe JavaScriptu będzie szukało plików kodu zawierającego kod danego modułu. W przypadku projektu przedstawionego w rozdziale położenie zostało zdefiniowane za pomocą względnej ścieżki dostępu rozpoczynającej się od jednej lub dwóch kropek, co wskazuje na ścieżkę względem bieżącego pliku lub katalogu nadrzędnego bieżącego pliku. Na listingu 4.26 położenie zaczyna się od kropki:

```
...
import calcTax from "./tax";
...
```

To położenie wskazuje narzędziom kompilacji istnienie zależności od modułu `tax` znajdującego się w tym samym katalogu co plik zawierający polecenie `import`. Zwróć uwagę na brak rozszerzenia pliku w lokalizacji modułu.

Jeżeli pominiesz kropkę lub kropki na początku, wówczas polecenie `import` będzie deklarowało zależność od modułu nieznajdującego się w projekcie lokalnym. Sprawdzane lokalizacje modułu będą zależały od frameworka aplikacji i używanych narzędzi kompilacji. Najczęściej stosuje się katalog `node_modules`, do którego trafiają pakiety instalowane na etapie konfigurowania projektu. Ten katalog jest używany również podczas uzyskiwania dostępu do funkcjonalności dostarczanej przez pakiety firm trzecich. Przykłady wykorzystania modułów opracowanych przez firmy zewnętrzne przedstawię w trzeciej części książki. Tutaj masz tylko przedsmak tego — polecenie `import` z rozdziału 19. poświęconego tworzeniu aplikacji z użyciem frameworka React:

```
...
import React, { Component } from "react";
...
```

W tym poleceniu `import` lokalizacja nie zaczyna się od kropki, więc jest interpretowana jako zależność od modułu `react` znajdującego się w katalogu `node_modules`. Wymieniony moduł to pakiet dostarczający podstawowe funkcje używane przez aplikację zbudowaną na bazie frameworka React.

Eksportowanie funkcji z modułu

Moduł może przypisywać nazwy eksportowanym funkcjom. To podejście, które preferuję w moich projektach. Na listingu 4.27 pokazałem nadanie nazwy funkcji eksportowanej przez moduł `tax`.

Listing 4.27. Eksportowanie nazwanej funkcji w kodzie pliku `tax.js` w katalogu `primer`

```
export function calculateTax(price) {
  return Number(price) * 1.2;
}
```

Ta funkcja oferuje dokładnie tę samą funkcjonalność, ale jest wyeksportowana za pomocą nazwy `calculateTax` i nie używa już słowa kluczowego `default`. Na listingu 4.28 pokazałem przykład zaimportowania w pliku `index.js` funkcji z modułu `tax` za pomocą jej nazwy.

Listing 4.28. Importowanie nazwanej funkcji w kodzie pliku `index.js` w katalogu `primer`

```
import { calculateTax } from "./tax";
```

```
class Product {
  constructor(name, price) {
    this.id = Symbol();
    this.name = name;
    this.price = price;
  }
}
```

```
let product = new Product("czapka", 100);
let taxedPrice = calculateTax(product.price);
console.log(`nazwa: ${ product.name }, cena wraz z podatkiem: ${taxedPrice}`);
```

Nazwa funkcji przeznaczona do zaimportowania została podana w nawiasie klamrowym i jest stosowana w kodzie. Moduł może eksportować funkcje domyślne i nazwane, jak pokazałem na listingu 4.29.

Listing 4.29. Eksportowanie funkcji nazwanych i domyślnych w kodzie pliku `tax.js` w katalogu `primer`

```
export function calculateTax(price) {
  return Number(price) * 1.2;
}

export default function calcTaxandSum(...prices) {
  return prices.reduce((total, p) => total += calculateTax(p), 0);
}
```

Nowa funkcjonalność została wyeksportowana za pomocą słowa kluczowego `default`. Na listingu 4.30 możesz zobaczyć przykład użycia tej funkcjonalności jako domyślnie eksportowanej przez moduł.

Listing 4.30. Importowanie funkcjonalności domyślnej w kodzie pliku `index.js` w katalogu `primer`

```
import calcTaxAndSum, { calculateTax } from "./tax";
```

```
class Product {
  constructor(name, price) {
    this.id = Symbol();
    this.name = name;
    this.price = price;
  }
}
```

```
let product = new Product("czapka", 100);
let taxedPrice = calculateTax(product.price);
console.log(`nazwa: ${ product.name }, cena wraz z podatkiem: ${taxedPrice}`);
```

```
let products = [new Product("rękawiczki", 23), new Product("buty", 100)];
let totalPrice = calcTaxAndSum(...products.map(p => p.price));
console.log(`cena całkowita: ${totalPrice.toFixed(2)}`);
```

Jest to wzorzec często stosowany podczas tworzenia aplikacji internetowych za pomocą frameworków takich jak React, w których podstawowa funkcjonalność jest dostarczana jako eksportowana domyślnie przez moduł, a funkcjonalność opcjonalna jest dostępna za pomocą nazwanych funkcji. Kod przedstawiony na listingu 4.30 powoduje wygenerowanie następujących danych wyjściowych:

```
nazwa: czapka, cena wraz z podatkiem: 120
cena całkowita: 147.60
```

Definiowanie w modelu wielu funkcjonalności nazwanych

Moduł może zawierać więcej niż jedną nazwaną funkcjonalność lub wartość, co jest użyteczne podczas grupowania powiązanych ze sobą funkcjonalności. Aby to zademonstrować, w katalogu *primer* utwórz plik o nazwie *utils.js* i zawartości przedstawionej na listingu 4.31.

Listing 4.31. Zawartość pliku *utils.js* w katalogu *primer*

```
import { calculateTax } from "./tax";

export function printDetails(product) {
  let taxedPrice = calculateTax(product.price);
  console.log(`nazwa: ${product.name}, cena wraz z podatkiem: ${taxedPrice}`);
}

export function applyDiscount(product, discount = 5) {
  product.price = product.price - 5;
}
```

Ten moduł definiuje dwie funkcje, dla których zastosowano słowo kluczowe `export`. W przeciwieństwie do poprzedniego przykładu, słowo kluczowe `default` nie jest używane, a każda funkcja ma swoją nazwę. Podczas importowania modułu zawierającego wiele funkcji nazwy funkcjonalności są podawane w postaci rozdzielonej przecinkami listy elementów umieszczonych w nawiasie klamrowym, jak pokazałem na listingu 4.32.

Listing 4.32. Importowanie nazwanych funkcjonalności w kodzie pliku *index.js* w katalogu *primer*

```
import calcTaxAndSum, { calculateTax } from "./tax";
import { printDetails, applyDiscount } from "./utils";

class Product {
  constructor(name, price) {
    this.id = Symbol();
    this.name = name;
    this.price = price;
  }
}
```

```
let product = new Product("czapka", 100);
applyDiscount(product, 10);
let taxedPrice = calculateTax(product.price);
printDetails(product);
let products = [new Product("rękawiczki", 23), new Product("buty", 100)];
let totalPrice = calcTaxAndSum(...products.map(p => p.price));
console.log(`cena całkowita: ${totalPrice.toFixed(2)}`);
```

Nawias klamrowy umieszczony po słowie kluczowym `import` zawiera nazwy funkcji, które będą używane. Konieczne jest zadeklarowanie zależności od wymaganych funkcji, natomiast w poleceniu `import` nie trzeba podawać funkcji, które nie będą używane. Kod przedstawiony na listingu 4.32 powoduje wygenerowanie następujących danych wyjściowych:

```
nazwa: czapka, cena wraz z podatkiem: 114
cena całkowita: 147.60
```

Podsumowanie

W tym rozdziale przedstawiłem funkcje JavaScriptu przeznaczone do pracy z obiektami, sekwencjami wartości, kolekcjami oraz modułami. Wprawdzie to wszystko są funkcje oferowane przez JavaScript, ale ich poznanie pomoże w umieszczeniu TypeScriptu we właściwym kontekście, a także ułatwi zdobycie podstaw do efektywnego programowania w języku TypeScript. W następnym rozdziale przedstawię wprowadzenie do kompilatora TypeScriptu, który jest sednem funkcji TypeScriptu oferowanych programistom.

ROZDZIAŁ 5.



Używanie kompilatora TypeScriptu

W tym rozdziale pokażę, jak używać kompilatora TypeScriptu odpowiedzialnego za konwersję kodu TypeScriptu na kod JavaScriptu, który następnie może być wykonany przez przeglądarkę WWW lub środowisko uruchomieniowe Node.js. Omówię także najużyteczniejsze w trakcie programowania z użyciem TypeScriptu opcje kompilatora, m.in. te stosowane we frameworkach aplikacji internetowych, które poznasz w trzeciej części książki.

Przygotowanie projektu

Aby przygotować się do utworzenia projektu omawianego w rozdziale, w dogodnym miejscu utwórz katalog o nazwie *tools*. Następnie przejdź do powłoki i wydaj polecenia przedstawione na listingu 5.1, aby wejść do nowego katalogu i nakazać menedżerowi pakietów Node (ang. *node package manager*) utworzenie pliku o nazwie *package.json*. Plik ten będzie używany do monitorowania pakietów dodanych do projektu, co dokładniej omówię w sekcji „Używanie menedżera pakietów Node”.

Listing 5.1. Tworzenie pliku package.json w katalogu projektu

```
$ cd tools
$ npm init --yes
```

W powłoce z poziomu katalogu *tools* wydaj polecenia przedstawione na listingu 5.2, aby zainstalować pakiety niezbędne podczas pracy nad projektem omawianym w tym rozdziale.

Listing 5.2. Dodawanie pakietów przy użyciu menedżera pakietów Node

```
$ npm install --save-dev typescript@4.2.2
$ npm install --save-dev tsc-watch@4.2.9
```

Argument `install` nakazuje menedżerowi pakietów pobranie i dodanie pakietów do bieżącego katalogu. Z kolei argument `--save-dev` informuje menedżer pakietów, że wymienione pakiety będą używane podczas tworzenia aplikacji, ale nie są jej częścią. Ostatnim argumentem jest nazwa pakietu, po której znajduje się znak `@` i wymagana wersja pakietu.

■ **Uwaga** Ważne znaczenie ma używanie wersji podanych w przykładach. Jeżeli zdecydujesz się na użycie innych wersji, skutkiem może być odmienne zachowanie aplikacji lub nawet wystąpienie błędów.

Aby utworzyć plik konfiguracyjny dla kompilatora TypeScriptu, należy dodać do katalogu `tools` plik o nazwie `tsconfig.json` z zawartością przedstawioną na listingu 5.3.

Listing 5.3. Zawartość pliku `tsconfig.json` w katalogu `tools`

```
{
  "compilerOptions": {
    "target": "es2018",
    "outDir": "./dist",
    "rootDir": "./src"
  }
}
```

Dokończ konfigurację projektu przez utworzenie katalogu `tools/src` i umieść w nim plik o nazwie `index.ts` zawierający kod przedstawiony na listingu 5.4.

Listing 5.4. Zawartość pliku `index.ts` w katalogu `src`

```
function printMessage(msg: string): void {
  console.log(`Komunikat: ${ msg }`);
}

printMessage("Witaj, TypeScript");
```

Aby skompilować kod TypeScriptu, należy z poziomu katalogu `tools` wydać polecenie przedstawione na listingu 5.5.

Listing 5.5. Kompilowanie kodu TypeScriptu w projekcie

```
$ tsc
```

Wykonanie skompilowanego kodu nastąpi po wydaniu z poziomu katalogu `tools` w powłoce polecenia przedstawionego na listingu 5.6.

Listing 5.6. Uruchamianie skompilowanego kodu

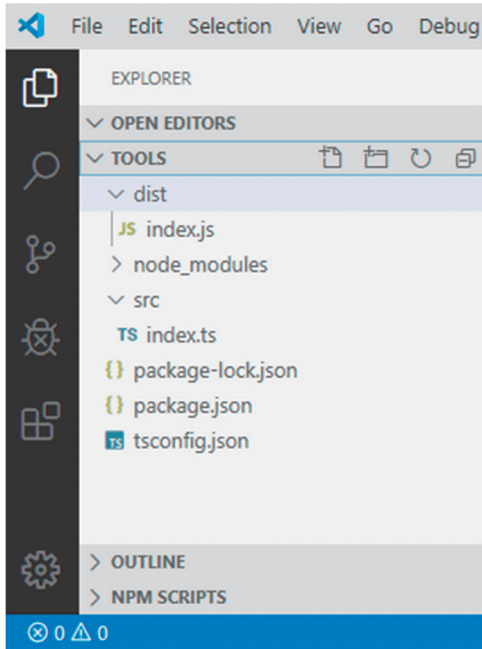
```
$ node dist/index.js
```

Jeżeli projekt został przygotowany prawidłowo, w powłoce zostaną wyświetlone następujące dane wyjściowe.

```
Komunikat: Witaj, TypeScript
```

Struktura projektu

Struktura przykładowego projektu to ta, z którą będziesz się najczęściej spotykać w większości projektów JavaScriptu i TypeScriptu, oczywiście z pewnymi różnicami związanymi z podstawowym frameworkiem użytym w aplikacji, np. React lub Angular. Na rysunku 5.1 pokazałem zawartość katalogu *tools* wyświetloną w Visual Studio Code.



Rysunek 5.1. Zawartość katalogu przykładowego projektu

Na rysunku 5.1 pokazałem katalog projektu wyświetlony w Visual Studio Code, czyli w używanym przeze mnie edytorze kodu źródłowego, w którym powstały projekty omówione w książce. Z kolei w tabeli 5.1 wymieniłem poszczególne elementy projektu, a nieco więcej informacji o jego najważniejszych elementach znajdziesz w dalszej części rozdziału.

Tabela 5.1. Pliki i katalogi tworzące przykładowy projekt

Nazwa	Opis
<i>dist</i>	Katalog zawiera dane wyjściowe wygenerowane przez kompilator
<i>node_modules</i>	Katalog zawiera pakiety wymagane przez aplikację i narzędzia programistyczne, jak to zostało omówione w podrozdziale „Używanie menedżera pakietów Node”
<i>src</i>	Katalog zawiera pliki kodu źródłowego, który ma zostać skompilowany przez kompilator TypeScriptu

Tabela 5.1. Pliki i katalogi tworzące przykładowy projekt (ciąg dalszy)

Nazwa	Opis
<i>package.json</i>	Katalog zawiera zależności pakietów najwyższego poziomu dla projektu, jak to zostało omówione w podrozdziale „Używanie menedżera pakietów Node”
<i>package-lock.json</i>	Plik zawiera pełną listę zależności pakietów projektu
<i>tsconfig.json</i>	Plik zawiera ustawienia konfiguracyjne dla kompilatora TypeScriptu

Używanie menedżera pakietów Node

Podczas tworzenia projektów TypeScriptu i JavaScriptu wykorzystywany jest bogaty ekosystem pakietów. Większość projektów TypeScriptu będzie wymagała pakietów dostarczających kompilator TypeScriptu, framework aplikacji (o ile jest używany), a także narzędzia niezbędne do kompilacji kodu źródłowego, aby mógł być rozprowadzany i wykonywany.

Menedżer pakietów Node jest używany do pobierania wspomnianych pakietów i umieszczania ich w katalogu *node_modules* projektu. Każdy pakiet deklaruje zbiór zależności od innych pakietów i określa wersję, z którą może współdziałać. Menedżer pakietów Node stosuje się do łańcucha zależności, ustala wersje wymagane przez poszczególne pakiety, a następnie pobiera wszystkie niezbędne komponenty.

Plik *package.json* jest używany do monitorowania pakietów dodawanych za pomocą polecenia `npm install`. Spójrz na zawartość pliku *package.json* w przykładowym projekcie:

```
...
{
  "name": "tools",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Błąd: nie podano nazwy katalogu\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "tsc-watch": "^4.2.9",
    "typescript": "^4.2.2"
  }
}
...
```

Podstawowa zawartość pliku została utworzona przez polecenie `npm init` przedstawione na listingu 5.1, a następnie zmodyfikowana przez każde kolejne wykonanie polecenia `npm install` na listingu 5.2. Pakiety dzielą się na narzędzia używane podczas tworzenia aplikacji i te, które stanowią jej część. Pakiety używane w trakcie pracy nad aplikacją są instalowane z użyciem argumentu `--save-dev` i trafiają do sekcji `devDependencies` pliku *package.json*. Z kolei pakiety

tworzące część aplikacji są instalowane bez argumentu `--save-dev` i trafiają do sekcji `dependencies`. Polecenia przedstawione na listingu 5.2 spowodowały zainstalowanie jedynie pakietów narzędziowych, które trafiły do sekcji `devDependencies`, i dlatego plik `package.json` w ogóle nie zawiera sekcji `dependencies`. W przykładach przedstawionych w dalszej części książki będę dodawał pakiety trafiające do sekcji `dependencies`, natomiast w tym rozdziale skoncentruję się na narzędziach używanych podczas programowania w języku TypeScript. W tabeli 5.2 pokrótce omówiłem pakiety dodane do przykładowego projektu.

Tabela 5.2. Pakiety dodane do przykładowego projektu

Nazwa	Opis
tsc-watch	Pakiet monitoruje katalog kodu źródłowego i po wykryciu jakiegokolwiek zmiany uruchamia kompilator TypeScriptu, a następnie wywołuje skompilowany kod JavaScriptu
typescript	Pakiet zawiera kompilator TypeScriptu oraz narzędzia wspomagające jego działanie

Pakiety globalne i lokalne

Menedżer pakietów może instalować pakiety dla pojedynczego projektu (mówimy wówczas o *instalacji lokalnej*) lub dostępne z poziomu każdego katalogu w systemie (mówimy wówczas o *instalacji globalnej*). W rozdziale 1. pakiet `typescript` został zainstalowany globalnie, co pozwala na używanie polecenia `tsc` do kompilowania kodu znajdującego się w dowolnym katalogu. Na listingu 5.2 ten sam pakiet został zainstalowany lokalnie, nawet mimo że oferowana przez niego funkcjonalność była już dostępna. Dzięki temu pozostałe pakiety umieszczone w danym projekcie również będą miały dostęp do funkcjonalności oferowanej przez kompilator TypeScriptu.

Dla każdego pakietu w pliku `package.json` są umieszczone szczegóły zawierające m.in. akceptowany numer wersji w formacie przedstawionym w tabeli 5.3.

Tabela 5.3. System numerowania wersji pakietu

Format	Opis
4.2.2	Określenie konkretnej wersji spowoduje zainstalowanie pakietu w podanej wersji, np. 4.2.2
*	Gwiazdka pozwala na pobranie i zainstalowanie dowolnej dostępnej wersji pakietu
>4.2.2 >=4.2.2	Poprzedzenie numeru wersji prefiksem <code>></code> lub <code>>=</code> pozwala na zainstalowanie dowolnej wersji pakietu większej niż podana wersja bądź też większej lub równej podanej wersji
<4.2.2 <=4.2.2	Poprzedzenie numeru wersji prefiksem <code><</code> lub <code><=</code> pozwala na zainstalowanie dowolnej wersji pakietu mniejszej niż podana wersja bądź też mniejszej lub równej podanej wersji

Tabela 5.3. System numerowania wersji pakietu (ciąg dalszy)

Format	Opis
~4.2.2	Poprzedzenie numeru wersji tyldą (znak ~) pozwala na zainstalowanie wersji pakietu nawet w przypadku niedopasowania wersji poprawki (ostatnia z trzech liczb rozdzielonych kropkami). Dlatego też podanie ~4.2.2 pozwala na zainstalowanie wersji 4.2.3 lub 4.2.4 (to poprawione wersje wydania 4.2.2), ale już nie 4.3.0, ponieważ to będzie kolejne wydanie pomocnicze
^4.2.2	Poprzedzenie numeru wersji znakiem ^ pozwala na zainstalowanie wersji pakietu nawet w przypadku niedopasowania wersji pomocniczej (druga z trzech liczb rozdzielonych kropkami) lub wersji poprawki (ostatnia z trzech liczb rozdzielonych kropkami). Dlatego też podanie ^4.2.2 pozwala na zainstalowanie wersji np. 4.2.3 lub 4.3.0, ale już nie 5.0.0

Menedżer pakietów Node to zaawansowane narzędzie, którego poznanie ma ważne znaczenie podczas programowania z użyciem języków JavaScript i TypeScript. W tabeli 5.4 wymieniałem wybrane polecenia menedżera pakietów Node, które mogą się okazać użyteczne podczas tworzenia aplikacji. Wszystkie te polecenia powinny być wydawane z poziomu katalogu projektu, czyli tego zawierającego plik *package.json*.

Tabela 5.4. Użyteczne polecenia menedżera pakietów Node

Polecenie	Opis
npm install	To polecenie lokalnie instaluje pakiety wymienione w pliku <i>package.json</i>
npm install pakiet@wersja	To polecenie lokalnie instaluje określoną wersję pakietu i aktualizuje plik <i>package.json</i> , aby dodać dany pakiet do sekcji <i>dependencies</i> pliku
npm install --save-dev pakiet@wersja	To polecenie lokalnie instaluje określoną wersję pakietu i aktualizuje plik <i>package.json</i> , aby dodać dany pakiet do sekcji <i>devDependencies</i> pliku. Wymieniona sekcja zawiera pakiety wymagane podczas pracy nad aplikacją, ale niebędące jej częścią
npm install -global pakiet@wersja	To polecenie globalnie instaluje określoną wersję pakietu
npm list	To polecenie wyświetla listę wszystkich pakietów lokalnych i ich zależności
npm run	To polecenie wykonuje jeden ze skryptów zdefiniowanych w pliku <i>package.json</i>
npx package	To polecenie wykonuje kod znajdujący się w pakiecie

Katalog *npm_modules* jest zwykle wykluczony z systemu kontroli wersji, ponieważ zawiera ogromną liczbę plików, a pakiety mogą zawierać komponenty przeznaczone dla konkretnej platformy, które mogą nie działać po ich pobraniu w innym komputerze. Zamiast tego należy w nowym komputerze wydać polecenie `npm install`, które utworzy nowy katalog *node_modules* i zainstaluje wymagane pakiety.

Takie podejście może prowadzić do wygenerowania odmiennego zestawu pakietów w trakcie każdego wydania polecenia `npm install`, ponieważ zależności mogą być wyrażane jako zakresy wersji, jak to przedstawiłem w tabeli 5.3. Aby zapewnić spójność, menedżer pakietów Node tworzy plik *package-lock.json* zawierający pełną listę pakietów zainstalowanych w katalogu *node_modules* z użytymi wersjami. Plik *package-lock.json* jest uaktualniany przez menedżer pakietów Node po wprowadzeniu zmiany w pakietach projektu, a podane wersje są pobierane i instalowane po wydaniu polecenia `npm install`.

■ **Uwaga** Pliki *package.json* i *package-lock.json* powinny być umieszczone w systemie kontroli wersji, aby mieć pewność, że każdy programista pracujący nad danym projektem będzie korzystał z tych samych pakietów. Po pobraniu zaktualizowanej wersji projektu z repozytorium upewnij się o wydaniu polecenia `npm install`, które pobierze i zainstaluje wszelkie nowe pakiety dodane do projektu przez innych programistów.

Plik konfiguracyjny kompilatora TypeScriptu

Kompilator TypeScriptu, *tsc*, jest odpowiedzialny za kompilację plików TypeScriptu. To kompilator odpowiada za implementację funkcjonalności TypeScriptu takich jak typowanie statyczne, a wynikiem działania jest czysty kod JavaScriptu pozbawiony słów kluczowych i wyrażeń języka TypeScript.

Kompilator TypeScriptu ma wiele opcji konfiguracyjnych, które omówię w dalszej części rozdziału. Plik konfiguracyjny jest używany do nadpisania ustawień domyślnych i zagwarantowania tego, że zawsze będzie stosowana spójna konfiguracja. Nazwa pliku konfiguracyjnego kompilatora TypeScriptu to *tsconfig.json*, a jego zawartość utworzona przez kod przedstawiony na listingu 5.3 jest następująca:

```
{
  "compilerOptions": {
    "target": "es2018",
    "outDir": "./dist",
    "rootDir": "./src"
  }
}
```

Wprawdzie plik *tsconfig.json* może zawierać ustawienia najwyższego poziomu, wymienione w tabeli 5.5, ale w omawianym tutaj przykładowym projekcie zawiera jedynie ustawienia w sekcji *compilerOptions*, które dokładniej przedstawię w podrozdziale „Użyteczne ustawienia konfiguracji kompilatora” w dalszej części rozdziału.

Tabela 5.5. Ustawienia najwyższego poziomu umieszczane w pliku *tsconfig.json*

Nazwa	Opis
<code>compilerOptions</code>	Ta sekcja grupuje ustawienia używane przez kompilator, jak w podrozdziale „Użyteczne ustawienia konfiguracji kompilatora” w dalszej części rozdziału
<code>files</code>	Te ustawienia określają kompilowane pliki i nadpisują zachowanie domyślne wskazujące lokalizacje sprawdzane, gdy kompilator szuka plików do skompilowania
<code>include</code>	Te ustawienia są używane podczas stosowania wzorca określającego pliki do skompilowania. Jeżeli ta opcja nie będzie użyta, zostaną wybrane pliki z rozszerzeniami <i>.ts</i> , <i>.tsx</i> i <i>.d.ts</i> . (Omówienie plików TSX znajdziesz w rozdziale 15., a plików <i>.d.ts</i> w rozdziale 14.)
<code>exclude</code>	To ustawienie jest używane do wykluczenia plików ze wzorca dopasowującego pliki przeznaczone do kompilacji
<code>compileOnSave</code>	Gdy ma przypisaną wartość <code>true</code> , to ustawienie podpowiada edytorowi kodu, że po każdym zapisaniu pliku powinien zostać uruchomiony kompilator. Ta funkcjonalność nie jest obsługiwana przez wszystkie edytory kodu źródłowego. Omówiona w dalszej części rozdziału funkcjonalność <code>watch</code> stanowi wówczas użyteczną alternatywę

Opcje `files`, `include` i `exclude` są użyteczne, gdy korzystasz z niestandardowej struktury projektu. Przykładem może być tutaj integracja TypeScriptu z projektem zawierającym inny framework lub pakiet narzędziowy z zestawem plików powodującym konflikty. Zestaw plików odnajdywany przez kompilator można sprawdzić za pomocą ustawienia `listFiles`, które definiuje się w sekcji `compilerOptions` pliku *tsconfig.json* lub podaje w powłoce. Na przykład w powłoce z poziomu katalogu *tools* wydaj polecenie przedstawione na listingu 5.7, a poznasz listę plików, które będą wykorzystane przez aktualną konfigurację kompilatora.

Listing 5.7. Wyświetlanie listy plików przeznaczonych do skompilowania

```
$ tsc -listFiles
```

Argument `listFiles` wyświetla długą listę plików znalezionych przez kompilator, np.:

```
...
C:/npm/node_modules/typescript/lib/lib.es5.d.ts
C:/npm/node_modules/typescript/lib/lib.es2015.d.ts
C:/npm/node_modules/typescript/lib/lib.es2016.d.ts
C:/npm/node_modules/typescript/lib/lib.es2017.d.ts
C:/npm/node_modules/typescript/lib/lib.es2018.d.ts
...
```

Pliki wyświetlane przez opcję `listFiles` zawierają deklaracje typu odnalezione przez kompilator. Jak już wyjaśniłem w rozdziale 1., deklaracje typu opisują typy danych wykorzystywane przez kod JavaScriptu, aby mógł być bezpiecznie używany w aplikacji TypeScriptu. Pakiet TypeScriptu zawiera deklaracje typu dla różnych wersji języka JavaScript, dla API dostępnych w Node.js oraz dla przeglądarek WWW. Dokładniejsze omówienie deklaracji typu znajdziesz w rozdziale 14., natomiast te konkretne pliki przedstawiłem w dalszej części rozdziału.

-
- **Uwaga** Ścieżki dostępu dla plików deklaracji typu wskazują lokalizację poza projektem, ponieważ polecenie `tsc` uruchomiło kompilator TypeScriptu z pakietu zainstalowanego globalnie w rozdziale 1. Ten sam pakiet został zainstalowany lokalnie w katalogu `node_modules` i jest używany podczas pracy nad aplikacją, zgodnie z opisem w następnej sekcji. Jeżeli zachodzi potrzeba uruchomienia kompilatora z pakietu zainstalowanego lokalnie w projekcie, wówczas można skorzystać z polecenia `npx`, np. `npx tsc --listFiles`, którego wynik działania jest taki sam jak polecenia przedstawionego na listingu 5.7, ale używającego pakietu lokalnego.
-

Na końcu listy wygenerowanej po użyciu opcji `listFile` pojawia się następujący plik:

```
...
C:/tools/src/index.ts
...
```

W trakcie procesu odkrywania kompilator TypeScriptu wyszukuje pliki TypeScriptu w położeniu wskazanym przez ustawienie `rootDir` w pliku `tsconfig.json`. Kompilator analizuje katalog `src` i odkrywa umieszczony w nim plik `index.ts`.

Kompilacja kodu TypeScriptu

Kompilator sprawdza kod TypeScriptu w celu wymuszenia zastosowania funkcjonalności takich jak typowanie statyczne i generuje czysty kod JavaScriptu pozbawiony wszelkich dodatków TypeScriptu. Kompilator można uruchomić bezpośrednio z poziomu powłoki i będzie on przetwarzał wszystkie pliki wymienione przez opcję `listFile`. Z poziomu katalogu `tools` w powłoce wydaj polecenie przedstawione na listingu 5.8, aby uruchomić kompilator.

Listing 5.8. Uruchamianie kompilatora TypeScriptu

```
$ tsc
```

W projekcie znajduje się tylko jeden plik TypeScriptu, `src/index.ts`, a ustawienia konfiguracyjne w pliku `tsconfig.json` nakazują kompilatorowi umieszczanie w katalogu `dist` wygenerowanych przez niego plików JavaScriptu. Jeżeli zajrzysz do katalogu `dist`, znajdziesz w nim plik o nazwie `index.js` z następującą zawartością:

```
...
function printMessage(msg) {
    console.log(`Komunikat: ${msg}`);
}
printMessage("Witaj, TypeScript");
...
```

Ten plik zawiera skompilowany kod pochodzący z pliku `src/index.ts`, ale bez dodatkowych informacji o typie dla funkcji `printMessage()`. Związek między kodem TypeScriptu a wygenerowanym przez kompilator kodem JavaScriptu nie zawsze będzie aż tak bezpośredni, zwłaszcza jeśli kompilator ma generować kod dla innej wersji języka JavaScript, co dokładnie omówię w dalszej części rozdziału.

-
- **Ostrzeżenie** Nie przeprowadzaj edycji plików JavaScriptu znajdujących się w katalogu *dist*, ponieważ te zmiany zostaną nadpisane w trakcie następnego uruchomienia kompilatora TypeScript. Zmiany należy wprowadzać jedynie w plikach kodu TypeScriptu.
-

Błędy generowane przez kompilator

Kompilator TypeScriptu sprawdza kompilowany kod i upewnia się o jego zgodności ze specyfikacją JavaScriptu, a także stosuje funkcjonalność TypeScriptu, np. typowanie statyczne i słowa kluczowe związane z kontrolą dostępu. W celu przedstawienia przykładu prostego błędu kompilatora w kodzie na listingu 5.9 dodałem polecenie używające nieprawidłowego typu danych podczas wywoływania funkcji `printMessage()`.

Listing 5.9. Wprowadzenie niedopasowania typu w kodzie pliku *index.ts* w katalogu *src*

```
function printMessage(msg: string): void {
    console.log(`Komunikat: ${ msg }`);
}
```

```
printMessage("Witaj, TypeScript");
```

```
printMessage(100);
```

Uruchomienie kompilatora następuje po wydaniu z poziomu katalogu *tools* polecenia przedstawionego na listingu 5.10.

Listing 5.10. Uruchamianie kompilatora TypeScriptu

```
$ tsc
```

-
- **Wskazówka** Funkcja `printMessage()` określa akceptowany typ danych za pomocą parametru `msg` przy użyciu adnotacji typu, co dokładnie omówię w rozdziale 7. W tym miejscu wystarczy wiedzieć, że wywołanie funkcji `printMessage()` w pokazany sposób z jedynie wartością liczbową jest błędem w TypeScriptie.
-

Kompilator wykrywa, że typem argumentu w nowym poleceniu jest `number`, a nie `string`, jak to zostało zdefiniowane w funkcji `printMessage()`, więc generuje następujący komunikat błędu:

```
src/index.ts:6:14 - error TS2345: Argument of type '100' is not assignable to parameter
of type 'string'.
```

```
6 printMessage(100);
```

```
~~~~~
Found 1 error.
```

Pod wieloma względami kompilator TypeScriptu działa podobnie jak każdy inny kompilator. Istnieje jednak pewna różnica, o której trzeba wiedzieć: domyślnie kompilator kontynuuje generowanie kodu JavaScriptu nawet po napotkaniu błędu. Jeżeli przeanalizujesz zawartość pliku *index.js* w katalogu *dist*, zauważysz w nim następujący fragment kodu:

```
...
function printMessage(msg) {
    console.log(`Komunikat: ${msg}`);
}
printMessage("Witaj, TypeScript");
printMessage(100);
...
```

Jest to przykład dziwnego zachowania, które może sprawić problemy podczas łączenia narzędzi wykonujących lub dalej przetwarzających kod JavaScriptu wygenerowany przez kompilator TypeScriptu, ponieważ w takim przypadku te narzędzia będą operowały na plikach JavaScriptu potencjalnie zawierających błędy. Na szczęście takie zachowanie można wyłączyć za pomocą opcji konfiguracyjnej `noEmitOnError`, której w pliku *tsconfig.json* należy przypisać wartość `true`, jak pokazałem na listingu 5.11.

Listing 5.11. Zmiana ustawień kompilatora w pliku *tsconfig.json* w katalogu *tools*

```
{
  "compilerOptions": {
    "target": "es2018",
    "outDir": "./dist",
    "rootDir": "./src",
    "noEmitOnError": true
  }
}
```

Po uruchomieniu kompilatora dane wyjściowe zostaną wygenerowane tylko wtedy, gdy w kodzie JavaScriptu nie będzie żadnych błędów.

Używanie trybu monitorowania i wykonywania skompilowanego kodu

Samodzielne uruchamianie kompilatora po każdej modyfikacji szybko staje się uciążliwe i dlatego obsługuje on opcję monitorowania projektu, automatycznie przeprowadzającą kompilację plików, w których zostały wykryte zmiany. Aby uruchomić kompilator w trybie monitorowania projektu pod kątem zmian w plikach, z poziomu katalogu *tools* wydaj polecenie przedstawione na listingu 5.12.

Listing 5.12. Uruchamianie kompilatora w trybie monitorowania projektu pod kątem zmian

```
$ tsc -watch
```

Kompilator zostanie uruchomiony i zgłosi ten sam błąd co wcześniej, a następnie rozpocznie monitorowanie projektu pod kątem zmian. Aby wywołać kompilację, błędne polecenie w pliku *index.ts* poprzedź znakiem komentarza, jak pokazałem na listingu 5.13.

Listing 5.13. Umieszczenie polecenia w komentarzu w kodzie pliku *index.ts* w katalogu *src*

```
function printMessage(msg: string): void {
    console.log(`Komunikat: ${ msg }`);
}

printMessage("Witaj, TypeScript");
//printMessage(100);
```

■ **Ostrzeżenie** Po uruchomieniu kompilatora TypeScriptu w trybie monitorowania projektu pod kątem zmian możesz natknąć się na błąd `Check failed: U_SUCCESS(status)`. Jeżeli ten błąd wystąpi, konieczne jest uaktualnienie do najnowszej wersji Node.js. Ewentualnie możesz przejść do następnej sekcji, ponieważ kompilator w trybie monitorowania został użyty jedynie w tej części książki i nigdzie indziej.

Po zapisaniu pliku kompilator będzie automatycznie uruchomiony. W kodzie nie ma żadnych błędów i dlatego kompilator wygeneruje następujące dane wyjściowe:

```
[6:37:35 AM] File change detected. Starting incremental compilation...
[6:37:35 AM] Found 0 errors. Watching for file changes.
```

Aby uruchomić skompilowany kod, należy przejść do drugiego okna powłoki, następnie wejść do katalogu *tools* i wydać polecenie przedstawione na listingu 5.14.

Listing 5.14. Uruchamianie skompilowanego kodu

```
$ node dist/index.js
```

Środowisko uruchomieniowe Node.js wykona polecenia umieszczone w pliku *index.js* w katalogu *dist* i wygeneruje następujące dane wyjściowe:

```
Komunikat: Witaj, TypeScript
```

Automatyczne wykonywanie kodu po kompilacji

Jeżeli kompilator działa w trybie monitorowania projektu pod kątem zmian, skompilowany kod nie jest automatycznie uruchamiany. Kuszące wydaje się połączenie trybu monitorowania z narzędziem uruchamiającym kod, ale to może być trudne, ponieważ nie wszystkie pliki JavaScriptu są zapisywane jednocześnie i nie istnieje łatwy sposób na niezawodne ustalenie, kiedy kompilacja się zakończyła.

Jeżeli podczas tworzenia aplikacji internetowej używasz frameworków takich jak React, Angular lub Vue.js, kompilator TypeScriptu jest zintegrowany z większym łańcuchem narzędzi wykonywanym automatycznie, jak to pokażę w trzeciej części książki. Natomiast w przypadku projektów niekorzystających ze wspomnianych frameworków istnieją pewne narzędzia typu *open source* zbudowane na bazie funkcjonalności dostarczanych przez kompilator i oferujące dodatkowe możliwości. Jednym z takich pakietów jest właśnie *ts-watch*, który w przykładowym projekcie został zainstalowany przez polecenie przedstawione na listingu 5.2. Pakiet *ts-watch*

uruchamia kompilator w trybie monitorowania projektu pod kątem zmian, analizuje jego dane wyjściowe i na ich podstawie podejmuje odpowiednie działania. Jeżeli chcesz uruchomić ten pakiet, z poziomu katalogu *tools* wydaj polecenie pokazane na listingu 5.15.

Listing 5.15. Uruchamianie polecenia pakietu

```
$ npx tsc-watch --onsuccess "node dist/index.js"
```

Argumenty wiersza poleceń PowerShell

Jeżeli używasz wiersza poleceń Microsoft PowerShell, otrzymasz ostrzeżenie informujące, że *index.js* to plik JavaScriptu. Tak się dzieje, ponieważ PowerShell nie potrafi prawidłowo obsługiwać argumentów zawierających spacje. Dlatego też zamiast polecenia przedstawionego na listingu 5.15 należy użyć następującego:

```
$ npx tsc-watch --onsuccess "\"node dist\\index.js\""
```

Zwróć szczególną uwagę na kolejność znaków sterujących, cudzysłowy i grawisy (```).

Argument *onsuccess* wskazuje polecenie wykonywane, gdy kompilacja zakończy się bezbłędnie. W pliku *index.ts* wprowadź zmianę przedstawioną na listingu 5.16, aby w ten sposób wywołać kompilację i uruchomienie jej wyniku.

■ **Wskazówka** Informacje o innych opcjach oferowanych przez pakiet *ts-watch* znajdziesz na stronie <https://github.com/gilamran/ts-watch>.

Listing 5.16. Wprowadzanie zmian w kodzie pliku *index.ts* w katalogu *src*

```
function printMessage(msg: string): void {
    console.log(`Komunikat: ${ msg }`);
}
```

```
printMessage("Witaj, TypeScript");
printMessage("Mamy dzisiaj słoneczny dzień");
```

Po zapisaniu pliku ze zmianami kompilator TypeScriptu wykryje zmianę i skompiluje ten plik. Gdy pakiet *ts-watch* ustali, że kompilator nie wygenerował żadnych błędów, wykona polecenie powodujące uruchomienie skompilowanego kodu, czego skutkiem będzie wyświetlenie następujących danych wyjściowych:

```
7:20:25 AM - File change detected. Starting incremental compilation...
7:20:25 AM - Found 0 errors. Watching for file changes.
Komunikat: Witaj, TypeScript
Komunikat: Mamy dzisiaj słoneczny dzień
```

■ **Uwaga** Kompilator TypeScriptu dostarcza również API, który może być używany do tworzenia własnych narzędzi, które z kolei będą przydatne, gdy zachodzi potrzeba integracji kompilatora z jeszcze bardziej złożonym rozwiązaniem. Wprawdzie Microsoft nie oferuje dokładnej dokumentacji tego API, ale pewne przykłady i informacje można znaleźć w repozytorium GitHub pod adresem <https://github.com/Microsoft/TypeScript/wiki/Using-the-Compiler-API>.

Uruchamianie kompilatora za pomocą menedżera pakietów Node

Kompilator TypeScriptu nie reaguje na zmiany każdej właściwości konfiguracyjnej i zdarzają się sytuacje, gdy trzeba będzie zatrzymać działanie kompilatora, a następnie ponownie go uruchomić. Zamiast wpisywać polecenie przedstawione na listingu 5.16, bardziej niezawodne rozwiązanie polega na użyciu sekcji `scripts` pliku `package.json`, jak pokazałem na listingu 5.17.

Listing 5.17. Dodawanie elementu do sekcji `scripts` pliku `package.json` w katalogu `tools`

```
{
  "name": "tools",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "tsc-watch --onsuccess \"node dist/index.js\""
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "tsc-watch": "^4.2.9",
    "typescript": "^4.2.2"
  }
}
```

Należy zachować ostrożność i prawidłowo zapisać wartość argumentu `onsuccess`. Zapisz zmiany w pliku `package.json`, a następnie z poziomu katalogu `tools` wydaj polecenie przedstawione na listingu 5.18.

Listing 5.18. Uruchamianie kompilatora

```
$ npm start
```

Efekt jest taki sam, ale kompilator można teraz uruchomić bez konieczności pamiętania kombinacji różnych nazw pakietów i plików, która w rzeczywistych projektach może być naprawdę skomplikowana.

Używanie funkcjonalności wersjonowania celu

TypeScript opiera swoje działanie na najnowszych wersjach języka JavaScript, w których zostały wprowadzone funkcjonalności takie jak klasy. Aby ułatwić adaptację TypeScriptu, kompilator potrafi wygenerować kod JavaScriptu przeznaczony również dla starszych wersji

języka JavaScript. Oznacza to, że najnowsze funkcje mogą być używane podczas pracy nad aplikacją, a następnie zostaną skonwertowane na postać starszego kodu JavaScriptu wykonywanego np. w starszych wersjach przeglądarek WWW.

Docelowa wersja języka JavaScript jest definiowana za pomocą ustawienia `target` w pliku `tsconfig.json`, jak pokazałem na listingu 5.19.

Listing 5.19. Wybieranie docelowej wersji języka JavaScript w pliku `tsconfig.json` w katalogu `tools`

```
{
  "compilerOptions": {
    "target": "es5",
    "outDir": "./dist",
    "rootDir": "./src",
    "noEmitOnError": true
  }
}
```

Wartością właściwości `target` może być dowolny element z wymienionych w tabeli 5.6.

Tabela 5.6. Wartości dla ustawienia `target`

Nazwa	Opis
es3	Ta wartość wskazuje trzecią wersję specyfikacji języka, zdefiniowaną w grudniu 1999 r. i uznawaną za bazową dla języka. Jest to wartość domyślna w przypadku braku ustawienia <code>target</code>
es5	Ta wartość wskazuje piątą wersję specyfikacji języka, zdefiniowaną w grudniu 2009 r. i skoncentrowaną na zachowaniu spójności. (Nie było czwartej wersji specyfikacji)
es6	Ta wartość wskazuje szóstą wersję specyfikacji języka, zawierającą wiele nowych funkcji wymaganych do tworzenia skomplikowanych aplikacji takich jak klasy, moduły, funkcje strzałek i obietnice
es2015	Ta wartość jest odpowiednikiem <code>es6</code>
es2016	Ta wartość wskazuje siódmą wersję specyfikacji języka, w której wprowadzono metodę <code>includes()</code> dla tablic i operator wykładnika
es2017	Ta wartość wskazuje ósmą wersję specyfikacji języka, w której wprowadzono funkcje przeznaczone do introspekcji obiektów i nowe słowa kluczowe dla operacji asynchronicznych
es2018	Ta wartość wskazuje dziewiątą wersję specyfikacji języka, w której wprowadzono operatory rozszczepiania i resztowy, a także usprawnienia w zakresie obsługi ciągów tekstowych i operacji asynchronicznych
es2019	Ta wartość wskazuje dziesiątą wersję specyfikacji języka, w której wprowadzono nowe funkcje tablic, zmodyfikowaną obsługę błędów, a także usprawnienia w zakresie formatowania danych JSON
es2020	Ta wartość wskazuje jedenastą wersję specyfikacji języka, w której wprowadzono operator koalescencyjny <code>nullish</code> , łączenie opcjonalne, a także funkcjonalność dynamicznego wczytywania modułów
esNext	Ta wartość wskazuje funkcje, które znajdują się w następnym wydaniu specyfikacji języka. Konkretnie funkcje obsługiwane przez kompilator TypeScriptu mogą się zmienić między wydaniem. Jest to ustawienie zaawansowane i podczas jego stosowania należy zachować ostrożność

■ **Uwaga** Wartość ES w tych ustawieniach odwołuje się do ECMAScriptu, czyli standardu definiującego funkcjonalność implementowaną przez język JavaScript. Historia języków JavaScript i ECMAScript jest długa, wyboista i niekoniecznie interesująca. Na potrzeby programowania w TypeScriptie języki JavaScript i ECMAScript można potraktować jako jedno i to samo i takie podejście przyjąłem w książce. Jeżeli jesteś ciekawy, więcej informacji na temat ECMAScriptu znajdziesz na stronie <https://en.wikipedia.org/wiki/ECMAScript>.

Wcześniejsze wersje standardu ECMAScriptu były oznaczane liczbami, ale w ostatnim czasie to się zmieniło i teraz wersje zawierają także rok wydania. Zmiana ta została wprowadzona podczas przygotowywania definicji ES6 i dlatego też ta wersja jest oznaczona jako ES6 i ES2015. Największą zmianą wprowadzoną w wydaniu ES6/ES2015 jest to, co można określić mianem „unowocześniania” JavaScriptu. Wydanie ES6 zapoczątkowało przejście do corocznych uaktualnień specyfikacji języka, stąd w wydaniach 2016-2020 wprowadzono tylko niewielkie zmiany.

Na podstawie ustawień przedstawionych na listingu 5.19 widać, że wybrano wersję ES5 standardu JavaScriptu, co oznacza brak obsługi dla nowoczesnych funkcji takich jak słowo kluczowe `let` i funkcja strzałki. Aby pokazać kompilatorowi, jak poradzić sobie z tą funkcjonalnością, należy w pliku `index.ts` wprowadzić zmiany zaprezentowane na listingu 5.20.

Listing 5.20. Używanie nowoczesnych funkcjonalności w kodzie pliku `index.ts` w katalogu `src`

```
let printMessage = (msg: string): void => console.log(`Komunikat: ${ msg }`);

let message = ("Witaj, TypeScript");
printMessage(message);
```

Po zapisaniu zmian w pliku kod zostanie skompilowany i uruchomiony. Kod JavaScriptu wygenerowany przez kompilator można zobaczyć w pliku `index.js` w katalogu `dist`. Wymieniony plik zawiera następujące polecenia:

```
var printMessage = function (msg) { return console.log("Komunikat: " + msg); };
var message = ("Witaj, TypeScript");
printMessage(message);
```

Słowo kluczowe `let` zostało zastąpione przez `var`, a funkcja strzałki została zastąpiona zwykłą funkcją. Działanie tego kodu ma taki sam efekt jak w przypadku kodu przeznaczonego dla najnowszych wersji JavaScriptu i powoduje wygenerowanie następujących danych wyjściowych:

```
Komunikat: Witaj, TypeScript
```

Wybór plików biblioteki do kompilacji

Dane wyjściowe wygenerowane po użyciu opcji `listFiles` pokazały pliki odkryte przez kompilator i obejmowały także serię plików deklaracji typu. Te pliki dostarczają kompilatorowi informacje typu dotyczące funkcjonalności dostępnych w różnych wersjach JavaScriptu oraz funkcjonalności dostarczane aplikacji uruchomionej w przeglądarce WWW — pozwalające na zarządzanie treścią HTML-a za pomocą API modelu DOM.

Domyślnie kompilator ustala potrzebne mu pliki informacji typu na podstawie właściwości `target`, co oznacza wygenerowanie błędów w przypadku użycia funkcji z nowszych wersji JavaScriptu, jak pokazałem na listingu 5.21.

Listing 5.21. Używanie w kodzie pliku `index.ts` w katalogu `src` funkcjonalności z nowszych wersji JavaScriptu

```
let printMessage = (msg: string): void => console.log(`Komunikat: ${ msg }`);

let message = ("Witaj, TypeScript");
printMessage(message);

let data = new Map();
data.set("Bartek", "Londynie");
data.set("Alicja", "Paryżu");
data.forEach((val, key) => console.log(`${key} mieszka w ${val}`));
```

Obiekt `Map` został dodany do JavaScriptu jako część specyfikacji ES2015 i nie jest częścią wersji wskazywanej przez właściwość `target` w pliku konfiguracyjnym `tsconfig.json`. Po zapisaniu zmian w pliku kompilator wygeneruje następujące ostrzeżenie:

```
src/index.ts(6,16): error TS2583: Cannot find name 'Map'. Do you need to change your
target library? Try changing the `lib` compiler option to es2015 or later.
```

```
6:50:49 AM - Found 1 error. Watching for file changes.
```

Rozwiązaniem problemu jest podanie nowszej wersji języka JavaScript lub też zmiana definicji używanych przez kompilator. W tym drugim przypadku należy skorzystać z właściwości konfiguracyjnej `lib`, której przypisywana jest tablica wartości z tabeli 5.7.

Tabela 5.7. Wartości opcji kompilatora `lib`

Nazwa	Opis
es5, es2015, es2016, es2017, es2018, es2019, es2020	Te wartości odpowiadają plikom definicji typu, które z kolei odpowiadają określonym wersjom specyfikacji JavaScriptu. Istnieje możliwość skorzystania ze starego nazewnictwa, więc zamiast es2015 można użyć es6
esnext	Ta wartość pozwala wskazać funkcje będące dodatkiem do specyfikacji JavaScriptu, które nie zostały jeszcze oficjalnie zatwierdzone. Ten zbiór funkcji nieustannie się zmienia
dom	Ta wartość pozwala wskazać pliki informacji typu dla API modelu DOM. Aplikacje internetowe będą używały tego API do przeprowadzania operacji na treści HTML-a wyświetlanej przez przeglądarkę WWW. Ta opcja jest również użyteczna dla aplikacji Node.js

Tabela 5.7. Wartości opcji kompilatora lib (ciąg dalszy)

Nazwa	Opis
<code>dom.iterable</code>	Ta wartość pozwala na dostarczenie informacji typu dla API modelu DOM umożliwiającego iterację przez elementy HTML-a
<code>scripHost</code>	Ta wartość pozwala na wybór informacji typu dla środowiska Windows Script Host umożliwiającego automatyzację w systemach Windows
<code>webworker</code>	Ta wartość pozwala na wybór informacji typu dla funkcji workera umożliwiającej aplikacji internetowej wykonywanie zadań w tle

Istnieją również wartości przeznaczone do wskazywania konkretnych funkcji z danej wersji specyfikacji. Najużyteczniejsze z nich wymienię w tabeli 5.8.

Tabela 5.8. Wartości użytecznych pojedynczych funkcji dla opcji kompilatora lib

Nazwa	Opis
<code>es2015.core</code>	To ustawienie obejmuje informację typu dla głównych funkcjonalności wprowadzonych przez specyfikację ES2015
<code>es2015.collection</code>	To ustawienie obejmuje informację typu dla kolekcji Map i Set omówionych w rozdziałach 4. i 13.
<code>es2015.generator</code> <code>es2015.iterable</code>	Te ustawienia obejmują informacje typu dla funkcjonalności generatora i iteratora omówionych w rozdziałach 4. i 13.
<code>es2015.promise</code>	To ustawienie obejmuje informacje typu dla obietnic, które opisują operacje asynchroniczne
<code>es2015.reflect</code>	To ustawienie obejmuje informacje typu dla funkcjonalności refleksji, która zapewnia dostęp do właściwości i prototypów, jak to zostało omówione w rozdziale 16.
<code>es2015.symbol</code> <code>es2015.symbol.wellknown</code>	Te ustawienia obejmują informacje typu dla symboli omówionych w rozdziale 4.

Trzeba się koniecznie zastanowić nad implikacjami używania ustawienia konfiguracyjnego `lib`, ponieważ wskazuje ono kompilatorowi TypeScriptu, że środowisko uruchomieniowe aplikacji będzie oferowało obsługę określonego zbioru funkcjonalności, np. Map, jak w omawianym przykładzie. Kompilator ma możliwość zaadaptowania generowanego kodu JavaScriptu dla różnych funkcjonalności języka, przy czym nie dotyczy to obiektów takich jak kolekcje. Zmiana wartości ustawienia `lib` wskazuje kompilatorowi istnienie niestandardowego zbioru funkcjonalności dostępnych po uruchomieniu skompilowanego kodu JavaScriptu, a programista odpowiada za zagwarantowanie, że tak faktycznie będzie — ponieważ ma większą niż kompilator wiedzę o środowisku uruchomieniowym lub też wykorzystał w aplikacji skrypt typu polyfill taki jak `core-js` (<https://github.com/zloirock/core-js>).

Wersja Node.js zainstalowana w rozdziale 1. zapewnia obsługę większości najnowszych funkcji JavaScriptu i pozwala opierać się na istnieniu obiektu Map. Oznacza to, że można bezpiecznie zmienić ustawienie `lib` w pliku `tsconfig.json`, jak pokazałem na listingu 5.22.

Listing 5.22. Zmiana konfiguracji kompilatora w pliku `tsconfig.json` w katalogu `tools`

```
{
  "compilerOptions": {
    "target": "es5",
    "outDir": "./dist",
    "rootDir": "./src",
    "noEmitOnError": true,
    "lib": ["es5", "dom", "es2015.collection"]
  }
}
```

Wybrany przeze mnie zbiór typów obejmuje standardowe typy dla wersji języka JavaScript wskazywanej przez właściwość `target`, ustawienie `dom` (zapewnia dostęp do obiektu `console`) oraz funkcje kolekcji ES2015 z tabeli 5.8.

Kompilator wykryje zmianę w pliku konfiguracyjnym i przeprowadzi ponowną kompilację kodu źródłowego. Zmiana ustawienia `lib` wskazuje kompilatorowi dostępność obiektu `Map` i nie zostanie wygenerowany komunikat błędu. Podczas wykonywania kodu zostaną wygenerowane następujące dane wyjściowe:

```
Komunikat: Witaj, TypeScript
Bartek mieszka w Londynie
Alicja mieszka w Paryżu
```

Ten przykład działa, ponieważ wersja Node.js użyta w książce obsługuje najnowszą specyfikację języka JavaScript, która oferuje np. obiekt `Map`. W omawianym przykładzie wiem więcej o środowisku uruchomieniowym niż kompilator TypeScriptu i zmiana ustawienia `lib` pozwoliła na otrzymanie działającej aplikacji. Ten sam efekt można było uzyskać również przez zmianę ustawienia `target` na wskazującą nowszą wersję JavaScriptu, o której kompilator wie, że zapewnia obsługę kolekcji. Jeżeli tworzyłbyś kod dla środowiska uruchomieniowego obsługującego jedynie specyfikację ES5, wówczas musiałbyś dostarczyć implementację polyfill obiektu `Map`, np. oferowaną przez pakiet `core-js`.

Wybór formatu modułu

W rozdziale 4. wyjaśniłem, że moduły mogą być używane do podziału aplikacji JavaScriptu na wiele plików i dzięki temu projekt staje się łatwiejszy w zarządzaniu. Moduły zostały ustandaryzowane w specyfikacji ES2016, natomiast wcześniej stosowano różne rozwiązania w zakresie definiowania i używania modułów. Podczas tworzenia kodu TypeScriptu używane są ustandaryzowane funkcje związane z modułami. Na listingu 5.23 pokazałem zawartość pliku `calc.ts` dodanego do katalogu `src`.

Listing 5.23. Zawartość pliku `calc.ts` w katalogu `src`

```
export function sum(...vals: number[]): number {
  return vals.reduce((total, val) => total += val);
}
```

Nowy plik używa słowa kluczowego `export` do zdefiniowania funkcji o nazwie `sum()` redukującej tablicę wartości typu `number` w celu obliczenia ich sumy. Kod przedstawiony na listingu 5.24 importuje tę funkcję w pliku *index.ts*, a następnie ją wywołuje.

Listing 5.24. Używanie modułu w kodzie pliku *index.ts* w katalogu *src*

```
import { sum } from "./calc";

let printMessage = (msg: string): void => console.log(`Komunikat: ${ msg }`);

let message = ("Witaj, TypeScript");
printMessage(message);

let total = sum(100, 200, 300);
console.log(`Wartość całkowita: ${total}`);
```

Po zapisaniu pliku zostanie on przetworzony przez kompilator, a po uruchomieniu wygenerowanego kodu JavaScriptu będą wyświetlone następujące dane wyjściowe:

```
Komunikat: Witaj, TypeScript
Wartość całkowita: 600
```

Przeanalizuj zawartość pliku *index.ts* w katalogu *dist*, a przekonasz się, że kompilator TypeScript umieścił kod przeznaczony do pracy z modułami.

```
"use strict";
Object.defineProperty(exports, "__esModule", { value: true });
var calc_1 = require("./calc");
var printMessage = function (msg) { return console.log("Komunikat: " + msg); };
var message = ("Witaj, TypeScript");
printMessage(message);
var total = calc_1.sum(100, 200, 300);
console.log("Wartość całkowita: " + total);
```

Kompilator TypeScript używa właściwości konfiguracyjnej `target` w celu wybrania podejścia podczas pracy z modułami. Gdy wartością ustawienia `target` jest `es5`, wówczas użyty będzie styl modułu `commonjs`, który był wynikiem wcześniejszych prób wprowadzenia standardu modułu. Środowisko uruchomieniowe `Node.js` domyślnie obsługuje system modułu `commonjs` i dlatego kod wygenerowany przez kompilator TypeScript działa bezbłędnie.

Gdy kod jest przeznaczony dla nowszych wersji języka JavaScript, kompilator przechodzi do systemu modułu z wersji ES2015/ES6 specyfikacji JavaScriptu. Oznacza to przekazywanie słów kluczowych `import` i `export` z kodu TypeScript do JavaScriptu bez żadnych zmian. Na listingu 5.25 pokazałem zmianę konfiguracji kompilatora polegającą na wybraniu specyfikacji ES2018 języka JavaScript i usunięciu ustawienia `lib`, aby kompilator mógł używać domyślnych definicji typu.

Listing 5.25. Zmiana konfiguracji kompilatora w pliku *tsconfig.json* w katalogu *tools*

```
{
  "compilerOptions": {
    "target": "es2018",
    "outDir": "./dist",
    "rootDir": "./src",
    "noEmitOnError": true,
```

```

    // "lib": ["es5", "dom", "es2015.collection"]
  }
}

```

Po zapisaniu zmian w pliku konfiguracyjnym kompilator ponownie wygeneruje kod JavaScriptu, używając przy tym modułów standardowych. W chwili powstawania książki środowisko uruchomieniowe Node.js nie obsługuje modułów specyfikacji ES2015, więc wykonanie otrzymanego kodu JavaScriptu powoduje wygenerowanie następującego komunikatu błędu:

```

C:\tools\dist\index.js:1
import { sum } from "./calc";
~~~~~

```

SyntaxError: Cannot use import statement outside a module

System modułów można wyraźnie wybrać za pomocą ustawienia `module` w pliku konfiguracyjnym `tsconfig.json` za pomocą wartości wymienionych w tabeli 5.9.

Tabela 5.9. Typy systemów modułów w konfiguracji kompilatora TypeScriptu

Nazwa	Opis
none	Ta wartość powoduje wyłączenie obsługi modułów
commonjs	Ta wartość wybiera format modułu CommonJS obsługiwany przez Node.js
amd	Ta wartość wybiera format AMD (ang. <i>asynchronous module definition</i>) obsługiwany przez procedurę wczytującą moduły o nazwie RequireJS
system	Ta wartość wybiera format modułu obsługiwany przez procedurę wczytującą moduły o nazwie SystemJS
umd	Ta wartość wybiera format UMD (ang. <i>universal module definition</i>)
es2015, es6	Te wartości wybierają formy modułu zdefiniowane w specyfikacji ES2016 języka
es2020	Ta wartość wybiera format modułu zdefiniowany w specyfikacji ES2020 języka, która obsługuje dynamiczne wczytywanie modułów
esnext	Ta wartość wybiera funkcjonalność modułu zaproponowaną dla następnej wersji języka JavaScript

Wybór formatu modułu jest podyktowany środowiskiem uruchomieniowym, które będzie używane do wykonywania kodu. W czasie gdy powstaje ta książka, Node.js obsługuje moduły CommonJS i ECMAScript, choć wymaga to podania rozszerzenia pliku w poleceniu `import`, co okazuje się problematyczne podczas używania TypeScriptu, który obsługuje pliki z rozszerzeniami `.ts` i `.js`.

W przypadku aplikacji internetowych, zwłaszcza tych utworzonych za pomocą frameworków takich jak React, Angular lub Vue.js, format modułu będzie dyktowany przez łańcuch narzędzi frameworka, czyli przez tzw. bundler łączący wszystkie moduły w pojedynczy plik JavaScriptu przeznaczony do wdrożenia lub przez procedurę wczytywania modułu, która wykonuje żądania HTTP do serwera w celu pobrania niezbędnych plików JavaScriptu. Przykłady używania kompilatora

TypeScriptu z wymienionymi frameworkami przedstawię w trzeciej części książki. W omawianym przykładzie kod jest przeznaczony dla najnowszych wersji JavaScriptu w Node.js, więc wybrałem format `commonjs`, jak pokazałem na listingu 5.26.

-
- **Wskazówka** Alternatywne podejście polega na użyciu pakietu zewnętrznego w celu dodania obsługi dla modułów ES2015 w Node.js — takie rozwiązanie zostało zastosowane w rozdziale 4.
-

Listing 5.26. Wybieranie formatu modułu w pliku `tsconfig.json` w katalogu `tools`

```
{
  "compilerOptions": {
    "target": "es2018",
    "outDir": "./dist",
    "rootDir": "./src",
    "noEmitOnError": true,
    "lib": ["es5", "dom", "es2015.collection"]
    "module": "commonjs"
  }
}
```

Kompilator automatycznie nie uwzględnia niektórych zmian we właściwościach konfiguracyjnych. Aby mieć pewność o użyciu określonego formatu modułu, należy zakończyć działanie procesu kompilatora przez naciśnięcie klawiszy `Ctrl+C`, a następnie z poziomu katalogu `tools` wydać polecenie przedstawione na listingu 5.27.

Listing 5.27. Uruchamianie kompilatora

```
$ npm start
```

Kompilator dołączy kod wymagany przez format modułu `CommonJS`, a następnie kod JavaScriptu wygeneruje przedstawione tutaj dane wyjściowe:

```
Komunikat: Witaj, TypeScript
Wartość całkowita: 600
```

Ustalanie położenia modułu

Kompilator TypeScriptu może zastosować dwa odmienne podejścia w zakresie ustalania zależności modułów, wybierane na podstawie używanego formatu modułu. Dwa wspomniane tryby to *klasyczny*, w którym moduły są szukane w projekcie lokalnym, i *Node*, w którym moduły są szukane w katalogu `node_modules`. Kompilator TypeScriptu używa pierwszego z wymienionych trybów, gdy wartością właściwości `module` jest `es2015`, `system` lub `amd`. W przypadku wszystkich pozostałych ustawień modułów używany będzie drugi z wymienionych trybów. Wybór trybu ustalania położenia modułu odbywa się za pomocą właściwości konfiguracyjnej `moduleResolution` w pliku `tsconfig.json`, której należy przypisać wartość `classic` lub `node`.

Użyteczne ustawienia konfiguracji kompilatora

Kompilator TypeScriptu obsługuje ogromną liczbę ustawień konfiguracji. W drugiej części książki na początku każdego rozdziału będę zamieszczał tabelę ustawień kompilatora używanych przez funkcje omówione w przykładach danego rozdziału. W tabeli 5.10 wymieniałem opcje kompilatora użyte w książce. Wprawdzie wiele z nich nie ma większego sensu w tym momencie, ale każda zostanie omówiona w momencie użycia i będzie miała dla Ciebie sens, gdy zakończysz lekturę książki.

■ **Wskazówka** Pełna lista opcji obsługiwanych przez kompilator TypeScriptu została zamieszczona na stronie <https://www.typescriptlang.org/docs/handbook/compiler-options.html>.

Tabela 5.10. Opcje kompilatora TypeScriptu użyte w książce

Opcja	Opis
<code>allowJs</code>	Ta opcja powoduje dodanie wszystkich plików JavaScriptu w procesie kompilacji
<code>allowSyntheticDefaultImports</code>	Ta opcja powoduje importowanie modułów, które nie deklarują wyraźnie eksportowanych funkcji. Jest używana w celu zwiększenia zgodności kodu źródłowego
<code>baseUrl</code>	Ta opcja określa katalog główny używany podczas rozwiązywania zależności modułów
<code>checkJs</code>	Ta opcja nakazuje kompilatorowi sprawdzenie kodu JavaScriptu pod kątem najczęściej popełnianych błędów
<code>declaration</code>	Ta opcja generuje pliki deklaracji typu dostarczające informacje typu dla kodu JavaScriptu
<code>downlevelIteration</code>	Ta opcja włącza obsługę iteratorów, gdy kod jest przeznaczony dla starszych wersji języka JavaScript
<code>emitDecoratorMetadata</code>	Ta opcja powoduje dołączenie metadanych dekoratora w kodzie JavaScriptu wyemitowanym przez kompilator i jest używana z opcją <code>experimentalDecorators</code>
<code>esModuleInterop</code>	Ta opcja dodaje kod pomocniczy pozwalający na importowanie funkcjonalności z modułów niedefiniujących domyślnie eksportowanych funkcji i jest używana z opcją <code>allowSyntheticDefaultImports</code>
<code>experimentalDecorators</code>	Ta opcja włącza obsługę dekoratorów
<code>forceConsistentCasingInFileNames</code>	Ta opcja gwarantuje, że nazwy w poleceniach <code>import</code> będą dopasowane do wielkości znaków użytej w zaimportowanym pliku
<code>importHelpers</code>	Ta opcja określa, czy kod pomocniczy zostanie dodany do JavaScriptu w celu zmniejszenia ogólnej ilości wygenerowanego kodu

Tabela 5.10. Opcje kompilatora TypeScriptu użyte w książce (ciąg dalszy)

Opcja	Opis
<code>isolatedModules</code>	Ta opcja powoduje traktowanie każdego pliku jako oddzielnego modułu, co z kolei zwiększa zgodność z narzędziem Babel
<code>jsx</code>	Ta opcja określa sposób przetwarzania elementów HTML-a w plikach JSX/TSX
<code>jsxFactory</code>	Ta opcja określa nazwę funkcji fabryki, która jest używana do zastępowania elementów HTML-a w plikach JSX/TSX
<code>lib</code>	Ta opcja pozwala na wybór używanych przez kompilator plików deklaracji typu
<code>module</code>	Ta opcja określa format używany dla modułów
<code>moduleResolution</code>	Ta opcja określa styl używany podczas rozwiązywania zależności modułów
<code>noEmit</code>	Ta opcja uniemożliwia kompilatorowi emisję kodu JavaScriptu, a wynik jest jedynie sprawdzany pod kątem błędów
<code>noImplicitAny</code>	Ta opcja uniemożliwia niejawne używanie typu <code>any</code> , który kompilator stosuje, gdy nie jest w stanie ustalić konkretnego typu
<code>noImplicitReturns</code>	Ta opcja powoduje, że wszystkie ścieżki wykonywania w funkcji muszą się kończyć zwróceniem wyniku
<code>noUncheckedIndexedAccess</code>	Ta opcja uniemożliwia uzyskiwanie dostępu do właściwości za pomocą sygnatury indeksu, o ile nie będą one chronione przed wartościami <code>undefined</code> .
<code>noUnusedParameters</code>	Ta opcja powoduje wygenerowanie przez kompilator ostrzeżenia, jeśli funkcja definiuje nieużywane parametry
<code>outDir</code>	Ta opcja określa katalog, w którym zostaną umieszczone wygenerowane pliki JavaScriptu
<code>paths</code>	Ta opcja określa katalogi używane podczas rozwiązywania zależności modułu
<code>resolveJsonModule</code>	Ta opcja pozwala na importowanie plików JSON, jakby były modułami
<code>rootDir</code>	Ta opcja określa katalog główny używany przez kompilator do wyszukiwania plików TypeScriptu
<code>skipLibCheck</code>	Ta opcja przyspiesza kompilację przez pominięcie standardowo stosowanej operacji sprawdzania plików deklaracji
<code>sourceMap</code>	Ta opcja określa, czy kompilator ma generować pliki map źródłowych stosowane podczas debugowania
<code>strict</code>	Ta opcja włącza ściślejsze sprawdzanie kodu TypeScriptu

Tabela 5.10. Opcje kompilatora TypeScriptu użyte w książce (ciąg dalszy)

Opcja	Opis
<code>strictNullChecks</code>	Ta opcja uniemożliwia używanie <code>null</code> i <code>undefined</code> jako wartości dla innych typów
<code>suppressExcessPropertyErrors</code>	Ta opcja uniemożliwia kompilatorowi generowanie błędów w przypadku obiektów definiujących właściwości, które nie są w określonym kształcie
<code>target</code>	Ta opcja określa docelową wersję języka JavaScript, dla której kompilator wygeneruje kod
<code>typeRoots</code>	Ta opcja określa katalog główny, który kompilator będzie sprawdzał w poszukiwaniu plików deklaracji
<code>types</code>	Ta opcja określa listę plików deklaracji dołączanych w trakcie procesu kompilacji

Podsumowanie

W tym rozdziale przedstawiłem kompilator TypeScriptu odpowiedzialny za konwersję kodu TypeScriptu na czysty kod JavaScriptu. Wyjaśniłem sposób konfigurowania kompilatora, przedstawiłem różne sposoby jego używania, pokazałem, jak zmienić docelową wersję języka JavaScript dla generowanego kodu, a także jak zmienić sposób ustalania położenia modułów. Na końcu rozdziału zamieściłem listę opcji konfiguracyjnych kompilatora, które zostały użyte w książce. Wiele z nich jest w tym momencie niejasnych, ale nabiorą one większego sensu, gdy będziesz wykonywać kolejne przykłady. W następnym rozdziale będę kontynuował temat narzędzi programistycznych TypeScriptu i wyjaśnię kwestie związane z debugowaniem i testami jednostkowymi kodu TypeScriptu.

ROZDZIAŁ 6.



Testowanie i debugowanie kodu TypeScriptu

W tym rozdziale będę kontynuował temat narzędzi programistycznych TypeScriptu rozpoczęty w rozdziale 5. od wprowadzenia do kompilatora TypeScriptu. Pokażę różne sposoby debugowania kodu TypeScriptu, zaprezentuję używanie TypeScriptu z linterem oraz wyjaśnię, jak skonfigurować testy jednostkowe dla kodu TypeScriptu.

Przygotowanie projektu

W rozdziale będę kontynuował używanie projektu *tools* utworzonego w rozdziale 5. Nie trzeba wprowadzać żadnych zmian w projekcie.

■ **Wskazówka** Przykładowy projekt dla tego rozdziału, a także projekty dla pozostałych rozdziałów książki zostały umieszczone w archiwum dostępnym pod adresem <https://ftp.helion.pl/przyklady/tys4o2.zip>.

Przejdź do powłoki, a następnie z poziomu katalogu *tools* wydaj polecenie przedstawione na listingu 6.1, aby uruchomić kompilator w trybie monitorowania, wykorzystując pakiet *tsc-watch* zainstalowany w rozdziale 5.

Listing 6.1. Uruchamianie kompilatora

```
$ npm start
```

Kompilator zostanie uruchomiony, pliki TypeScriptu projektu skompilowane, a następnie zobaczysz następujące dane wyjściowe:

```
7:04:50 AM - Starting compilation in watch mode...
7:04:52 AM - Found 0 errors. Watching for file changes.
Komunikat: Witaj, TypeScript
Wartość całkowita: 600
```

Debugowanie kodu TypeScriptu

Kompilator TypeScriptu doskonale radzi sobie ze zgłaszaniem błędów składni i problemów związanych z typami danych. Jednak czasami zdarza się, że pomimo zakończonej sukcesem kompilacji kod nie działa w oczekiwany sposób. Użycie debuggera pozwala na przeanalizowanie stanu aplikacji w trakcie jej działania i może ujawnić powody występujących problemów. W tym podrozdziale dowiesz się, jak debugować aplikację TypeScriptu uruchomioną w środowisku Node.js. Z kolei w trzeciej części książki pokażę debugowanie aplikacji internetowych utworzonych w języku TypeScript.

Przygotowanie do debugowania

Trudność w debugowaniu aplikacji TypeScriptu polega na tym, że wykonywany kod jest produktem wygenerowanym przez kompilator, który przeprowadza konwersję kodu TypeScriptu na czysty kod JavaScriptu. Aby pomóc debuggerowi w powiązaniu kodu TypeScriptu i JavaScriptu, kompilator może wygenerować pliki nazywane *mapami źródła*. Na listingu 6.2 pokażę włączenie w pliku konfiguracyjnym *tsconfig.json* obsługi map źródła.

Listing 6.2. Włączenie obsługi map źródła w pliku *tsconfig.json* w katalogu *tools*

```
{
  "compilerOptions": {
    "target": "es2018",
    "outDir": "./dist",
    "rootDir": "./src",
    "noEmitOnError": true,
    "module": "commonjs",
    "sourceMap": true
  }
}
```

Gdy następnym razem kompilator będzie kompilował pliki TypeScriptu, w katalogu *dist* poza plikami JavaScriptu zostaną wygenerowane również pliki mapy z rozszerzeniem *.map*.

Dodawanie punktów przerwania

Edytory kodu źródłowego zapewniające dobrą obsługę języka TypeScript, np. Visual Studio Code, pozwalają na dodanie punktów przerwania do plików kodu źródłowego. Moje doświadczenia w pracy z tą funkcjonalnością są zróżnicowane i uznaję ją za zawodną. Dlatego też korzystam ze znacznie mniej eleganckiego, choć jednocześnie bardziej przewidywalnego słowa kluczowego debugger w JavaScriptcie. Gdy aplikacja JavaScriptu jest wykonywana przez debugger, wówczas zostanie zatrzymana po napotkaniu słowa kluczowego debugger i kontrola nad przebiegiem jej działania będzie przekazana do programisty. Zaletą takiego podejścia jest jego wszechstronność i niezawodność, choć trzeba pamiętać o usunięciu słowa kluczowego debugger przed wdrożeniem aplikacji. W większości przypadków środowisko uruchomieniowe ignoruje to słowo kluczowe podczas wykonywania programu w zwykły sposób, ale nie należy bezwzględnie polegać na takim

zachowaniu. (Lintowanie omówione w dalszej części rozdziału może chronić przed pozostawieniem słów kluczowych debugger w plikach kodu źródłowego). Na listingu 6.3 pokazałem przykład dodania słowa kluczowego debugger do pliku *index.ts*.

Listing 6.3. Dodawanie słowa kluczowego debugger w kodzie pliku *index.ts* w katalogu *src*

```
import { sum } from "./calc";

let printMessage = (msg: string): void => console.log(`Komunikat: ${ msg }`);

let message = ("Witaj, TypeScript");
printMessage(message);
debugger;

let total = sum(100, 200, 300);
console.log(`Wartość całkowita: ${total}`);
```

Wygenerowane dane wyjściowe nie zmienia się po wykonaniu tego fragmentu kodu, ponieważ domyślnie Node.js ignoruje słowo kluczowe debugger.

Używanie Visual Studio Code do debugowania

Większość dobrych edytorów kodu źródłowego przynajmniej do pewnego stopnia zapewnia obsługę debugowania kodu TypeScriptu i JavaScriptu. W tej sekcji pokażę, jak przeprowadzać debugowanie za pomocą Visual Studio Code, co powinno dostarczyć ogólne informacje na temat tego procesu. Jeżeli korzystasz z innego edytora, kroki konieczne do wykonania mogą być nieco odmienne, ale podstawowa zasada i reguły pozostaną podobne.

Aby przygotować konfigurację debugowania, z menu *Run* wybierz opcję *Add Configuration...*, a następnie z wyświetlonej listy wybierz *Node.js*, jak pokazałem na rysunku 6.1.

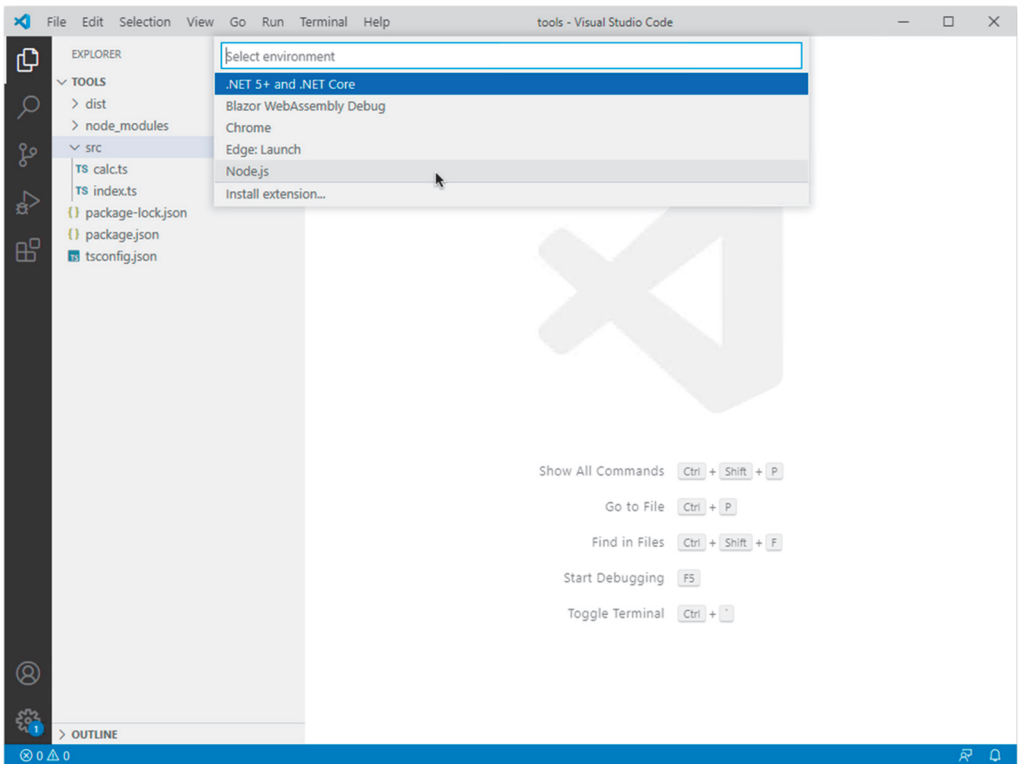
■ **Uwaga** Jeżeli nie uda się wybrać opcji *Add Configuration...*, wówczas zamiast niej spróbuj użyć opcji *Start Debugging*.

W katalogu projektu edytor utworzy podkatalog *.vscode* i umieści w nim plik o nazwie *launch.json* używany do konfigurowania debuggera. Wartość właściwości *program* zmień tak, aby debugger wykonywał kod JavaScriptu umieszczony w katalogu *dist*, jak pokazałem na listingu 6.4.

Listing 6.4. Zmianianie ścieżki wykonywania kodu za pomocą ustawień w pliku *launch.json* w podkatalogu *.vscode*

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "type": "node",
      "request": "launch",
      "name": "Launch Program",
      "skipFiles": [

```



Rysunek 6.1. Wybór środowiska debugowania

```

    "<node_internals>/**"
  ],
  "program": "${workspaceFolder}/dist/index.js"
}
]

```

Zapisz zmiany w pliku *launch.json*, a następnie z menu *Run* wybierz opcję *Start Debugging*. Visual Studio Code uruchomi pod kontrolą debuggera Node.js plik *index.js* umieszczony w katalogu *dist*. Działanie aplikacji będzie przebiegało w zwykły sposób aż do napotkania polecenia debugger. W tym momencie wykonywanie zostanie zatrzymane i kontrola nad przebiegiem działania programu będzie przekazana do wyskakującego okna debuggera, jak pokazałem na rysunku 6.2.

Stan aplikacji jest wyświetlany na pasku bocznym, na którym znajdują się informacje o zdefiniowanych zmiennych istniejących w chwili zatrzymania programu. Dostępny jest standardowy zbiór funkcji debugowania, m.in.: definiowanie wartowników, wchodzenie do lub opuszczanie konstrukcji, a także wznowienie działania programu. Okno *Debug Console* pozwala na wykonywanie poleceń JavaScriptu w kontekście aplikacji, więc np. wpisanie nazwy zmiennej i naciśnięcie klawisza *Enter* spowoduje wyświetlenie wartości przypisanej tej zmiennej.

Debugger ponownie wstrzymuje działanie programu po napotkaniu polecenia debugger. Daje to możliwość wykonywania poleceń przeznaczonych do analizy stanu aplikacji za pomocą np. wywołania `exec()`, choć wyrażenia muszą być umieszczane w cudzysłowie, podobnie jak ciąg tekstowy. W wierszu poleceń debuggera wpisz polecenie przedstawione na listingu 6.7.

Listing 6.7. Wykonywanie wyrażenia w debuggerze Node.js

```
exec("message")
```

Po naciśnięciu klawisza *Enter* debugger wyświetli wartość zmiennej `message`. W omawianym przykładzie będą to następujące dane wyjściowe:

```
'Witaj, TypeScript'
```

Lista dostępnych poleceń zostanie wyświetlona po wpisaniu polecenia `help` i naciśnięciu klawisza *Enter*. Z kolei dwukrotne naciśnięcie *Ctrl+C* spowoduje zakończenie sesji debugowania i powrót do zwykłej powłoki.

Używanie funkcji zdalnego debugowania w Node.js

Wprawdzie zintegrowany debugger Node.js jest użyteczny, ale jednocześnie trudny w obsłudze. Te same funkcje mogą być używane zdalnie za pomocą narzędzi programistycznych w przeglądarce WWW Google Chrome. Przede wszystkim należy zacząć od wydania z poziomu katalogu *tools* polecenia przedstawionego na listingu 6.8.

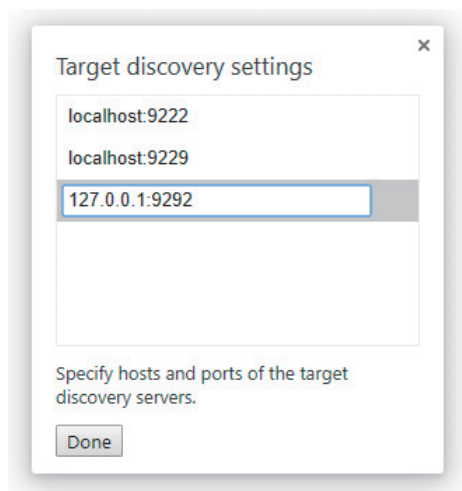
Listing 6.8. Uruchamianie Node.js w trybie zdalnego debugowania

```
$ node --inspect-brk dist/index.js
```

Argument `inspect-brk` uruchamia debugger i natychmiast wstrzymuje działanie aplikacji. Jest to wymagane w przypadku omawianej aplikacji, ponieważ wykonuje ona swoje zadanie, a następnie kończy działanie. Z kolei jeśli aplikacja po uruchomieniu wchodzi do pętli działającej w nieskończoność, np. jest serwerem WWW, wówczas można użyć argumentu `inspect`. Po uruchomieniu Node.js wyświetli komunikat podobny do następującego:

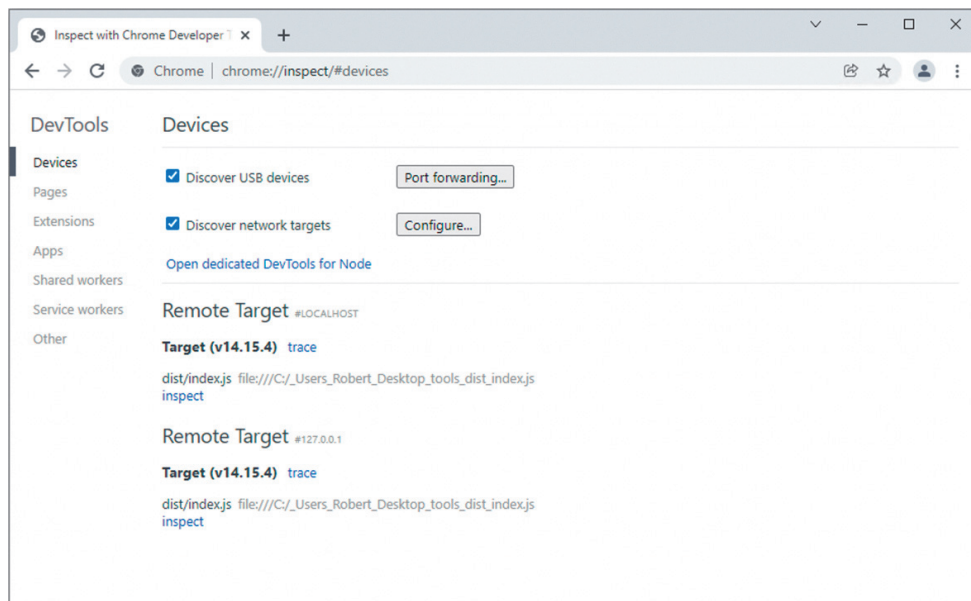
```
Debugger listening on ws://127.0.0.1:9229/e3cf5393-23c8-4393-99a1-d311c585a762
For help, see: https://nodejs.org/en/docs/inspector
```

Adres URL w danych wyjściowych jest używany do nawiązania połączenia z debuggerem i przejścia kontroli nad przebiegiem działania aplikacji. Otwórz nowe okno przeglądarki WWW Google Chrome, a następnie na pasku adresu wpisz `chrome://inspect`. Kliknij przycisk *Configure...* i wpisz adres IP z numerem portu z adresu URL widocznego w wyświetlonym wcześniej komunikacie. W moim przypadku jest to adres `127.0.0.1:9229`, jak pokazałem na rysunku 6.3.



Rysunek 6.3. Konfigurowanie Google Chrome do zdalnego debugowania Node.js

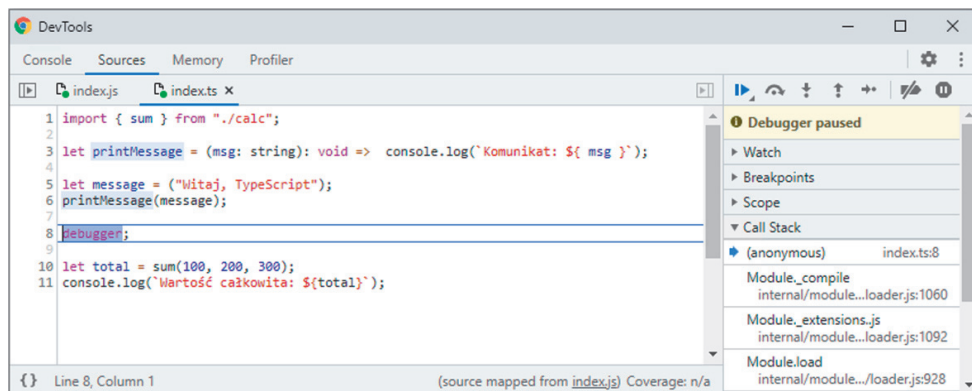
Kliknij przycisk *Done* i poczekaj chwilę, aż Chrome odszuka środowisko uruchomieniowe Node.js. Po odszukaniu zostanie ono umieszczone na liście *Remote Target*, jak pokazałem na rysunku 6.4.



Rysunek 6.4. Odkrycie środowiska uruchomieniowego Node.js

Kliknięcie łącza *inspect* spowoduje otworenie nowego okna Chrome z narzędziami programistycznymi, które będzie powiązane ze środowiskiem uruchomieniowym Node.js. Kontrola nad przebiegiem działania programu jest możliwa za pomocą standardowych przycisków

narzędzi, a wznowienie aplikacji pozwala środowisku uruchomieniowemu na kontynuowanie działania aż do chwili napotkania polecenia debugger. Początkowo w oknie debuggera będzie wyświetlony kod JavaScriptu, ale mapy źródła zostaną użyte po wznowieniu działania, jak pokazałem na rysunku 6.5.



Rysunek 6.5. Debugowanie aplikacji za pomocą narzędzi programistycznych w przeglądarce Chrome

Używanie lintera TypeScriptu

Lintar to narzędzie sprawdzające plik kodu źródłowego za pomocą zestawu reguł opisujących problemy, które mogą być źródłem błędów, prowadzić do wygenerowania nieoczekiwanych wyników, a także zmniejszyć czytelność kodu. Standardowym linterem dla TypeScriptu jest `typescript-eslint`. Jeżeli chcesz go dodać do projektu, przejdź do powłoki, a następnie z poziomu katalogu `tools` wydaj polecenie przedstawione na listingu 6.9.

-
- **Uwaga** Standardowym linterem dla TypeScriptu był TSLint, ale rozwój tego oprogramowania zakończono na rzecz pakietu `typescript-eslint`.
-

Listing 6.9. Dodawanie pakietu do przykładowego projektu

```
$ npm install --save-dev eslint@7.18.0
$ npm install --save-dev @typescript-eslint/parser@4.13.0
$ npm install --save-dev @typescript-eslint/eslint-plugin@4.13.0
```

Aby utworzyć konfigurację niezbędną do pracy z linterem, należy dodać do katalogu `tools` plik o nazwie `tslint.json`, którego zawartość przedstawiłem na listingu 6.10.

Listing 6.10. Zawartość pliku `.eslintrc` w katalogu `tools`

```
{
  "root": true,
  "ignorePatterns": ["node_modules", "dist"],
  "parser": "@typescript-eslint/parser",
```

```

    "parserOptions": {
      "project": "./tsconfig.json"
    },
    "plugins": [
      "@typescript-eslint"
    ],
    "extends": [
      "eslint:recommended",
      "plugin:@typescript-eslint/eslint-recommended",
      "plugin:@typescript-eslint/recommended"
    ]
  }
}

```

Lint jest dostarczany z prekonfigurowanym zbiorem reguł określonych za pomocą ustawienia `extends`, jak przedstawiłem w tabeli 6.1.

Tabela 6.1. Prekonfigurowane zbiory reguł TSLint

Nazwa	Opis
<code>eslint:recommended</code>	Zbiór reguł sugerowanych przez programistów tworzących ESLint przeznaczony do ogólnego stosowania podczas programowania w języku JavaScript
<code>@typescript-eslint/eslint-recommended</code>	Zbiór nadpisujący zbiór <code>recommended</code> w celu wyłączenia reguł niepotrzebnych podczas sprawdzania kodu źródłowego TypeScriptu
<code>@typescript-eslint/recommended</code>	Zbiór zawierający reguły dodatkowe, które są przeznaczone dla kodu TypeScriptu

Zatrzymaj proces `node` przez naciśnięcie klawiszy `Ctrl+C`, a następnie z poziomu katalogu `tools` wydaj polecenie przedstawione na listingu 6.11, aby tym samym uruchomić linter w przykładowym projekcie.

Listing 6.11. Uruchamianie lintera TypeScriptu

```
$ npx eslint .
```

Argument `project` nakazuje linterowi używanie ustawień kompilatora podczas wyszukiwania plików kodu źródłowego przeznaczonych do sprawdzenia, choć w tym przykładowym projekcie znajduje się tylko jeden plik TypeScriptu. Linter sprawdzi zgodność kodu z regułami i wygeneruje następujące dane wyjściowe:

```

C:\tools\src\index.ts
  3:5  error  'printMessage' is never reassigned. Use 'const' instead  prefer-const
  5:5  error  'message' is never reassigned. Use 'const' instead       prefer-const
  8:1  error  Unexpected 'debugger' statement                          no-debugger
 10:5  error  'total' is never reassigned. Use 'const' instead         prefer-const

4 problems (4 errors, 0 warnings)
  3 errors and 0 warnings potentially fixable with the `--fix` option.

```

Linter odszukał pliki kodu TypeScriptu i sprawdził ich zgodność z regułami zdefiniowanymi w pliku konfiguracyjnym. Kod w przykładowym projekcie łamie dwie reguły: reguła `prefer-const` wymaga stosowania słowa kluczowego `const` zamiast `let` podczas przypisywania wartości, która nie ulega zmianie, a reguła `no-debugger` uniemożliwia używanie słowa kluczowego `debugger`.

Wyłączanie reguł lintowania

Problem polega na tym, że wartość reguły lintowania często jest kwestią własnego stylu i preferencji, więc nawet jeśli dana reguła jest użyteczna, to nie zawsze będzie pomocna w każdej sytuacji. Lintowanie najlepiej sprawdza się, gdy generuje ostrzeżenia, którymi programista może się zająć później. Jeżeli otrzyma listę ostrzeżeń, które go nie interesują, istnieje duże prawdopodobieństwo, że nie zwróci na nie uwagi po zgłoszeniu poważniejszego problemu.

Reguła `prefer-const` podkreśla mankament w moim stylu tworzenia kodu, który nauczyłem się akceptować. Doskonale wiem, że powinienem używać `const` zamiast `let`, i próbuję to robić. Jednak moje nawyki stosowane podczas tworzenia kodu są głęboko zakorzenione i uważam, że pewnych problemów nie warto usuwać, zwłaszcza jeśli może to odciągnąć uwagę od ważniejszej kwestii związanej ze sposobem działania kodu. Akceptuję swoje niedoskonałości i wiem, że nadal będę stosował słowo kluczowe `let` w sytuacjach, w których `const` byłoby znacznie lepszym wyborem. Nie chcę, aby linter generował komunikaty informujące o złamaniu reguł. Reguły, które nigdy nie mają być stosowane w projekcie, można wyłączyć w konfiguracji lintera, jak pokazałem na listingu 6.12.

Listing 6.12. Wyłączenie reguł lintowania w `.eslintrc` w katalogu `tools`

```
{
  "root": true,
  "ignorePatterns": ["node_modules", "dist"],
  "parser": "@typescript-eslint/parser",
  "parserOptions": {
    "project": "./tsconfig.json"
  },
  "plugins": [
    "@typescript-eslint"
  ],
  "extends": [
    "eslint:recommended",
    "plugin:@typescript-eslint/eslint-recommended",
    "plugin:@typescript-eslint/recommended"
  ],
  "rules": {
    "prefer-const": 0
  }
}
```

Sekcja konfiguracyjna `rules` zawiera nazwy reguł z wartościami 1 lub 0 wskazującymi na włączenie lub wyłączenie danej reguły. W omawianym przykładzie przypisanie wartości 0 regule `prefer-const` wskazuje linterowi, aby ignorował stosowanie słowa kluczowego `let` tam, gdzie lepszym rozwiązaniem byłoby słowo kluczowe `const`.

Część reguł jest użyteczna w projekcie, ale wyłączona dla określonych plików lub poleceń. Do tej kategorii zalicza się reguła `no-debugger`. Ogólnie rzecz biorąc, słowo kluczowe `debugger` nie powinno być pozostawione w plikach kodu źródłowego, ponieważ może powodować problemy podczas wykonywania kodu. Jednak na etapie analizowania problemu to słowo kluczowe stanowi użyteczny sposób na niezawodne przejście kontroli nad wykonywaniem aplikacji, co pokazałem we wcześniejszej części rozdziału.

W takich przypadkach nie ma sensu wyłączenie reguły w pliku konfiguracyjnym lintera. Zamiast tego należy umieścić znak komentarza rozpoczynający się od ciągu tekstowego `eslint-disable-line`, a następnie zawierający jedną lub więcej nazw reguł wyłączonych dla kolejnego polecenia, jak pokazałem na listingu 6.13.

Listing 6.13. Wyłączanie reguły lintera dla pojedynczego polecenia w kodzie pliku `index.ts` w katalogu `src`

```
import { sum } from './calc';

let printMessage = (msg: string): void => console.log(`Komunikat: ${ msg }`);

let message = ("Witaj, TypeScript");
printMessage(message);

debugger; // eslint-disable-line no-debugger

let total = sum(100, 200, 300);
console.log(`Wartość całkowita: ${total}`);
```

Komentarz dodany w kodzie przedstawionym na listingu 6.13 nakazuje linterowi, aby reguła `no-debugger` nie była stosowana dla danego polecenia.

-
- **Wskazówka** Reguły mogą być wyłączane dla wszystkich poleceń znajdujących się po bloku komentarza (zdefiniowany między znakami `/*` i `*/`) rozpoczynającym się ciągiem tekstowym `eslint-disable`. Do wyłączenia wszystkich reguł lintowania służą komentarze `eslint-disable` lub `eslint-disable-line` bez żadnych nazw reguł.
-

Radości i smutki podczas lintowania

Linter to narzędzie o potężnych możliwościach, które może zdziałać wiele dobrego, zwłaszcza w zespole programistów o zróżnicowanych poziomach umiejętności i doświadczenia. Linter może wykryć najczęściej występujące problemy i trudne do wychwycenia błędy prowadzące do nieoczekiwanego zachowania lub kłopotów związanych z późniejszą konserwacją kodu. Lubię taki rodzaj lintowania i stosuję go w moim kodzie po utworzeniu ważnej funkcjonalności aplikacji lub przed przekazaniem kodu do systemu kontroli wersji.

Jednak linter może być także narzędziem wprowadzającym podziały i konflikty. Poza wykrywaniem błędów popełnionych w kodzie źródłowym, może być używany do wymuszania reguł związanych z wcięciami, umieszczaniem nawiasów, używaniem średników i spacji, a także do dziesiątek innych kwestii dotyczących stylu programowania. Większość programistów ma własne preferencje co do stylu, których się trzymają i uważają, że inni również powinni się do nich stosować. To dotyczy także mnie:

używam czterech spacji jako wcięcia i lubię mieć otwierający nawias klamrowy w tym samym wierszu, w którym znajduje się powiązane z nim wyrażenie. Wiem, że ten styl jest częścią „jedyne prawdziwego sposobu” tworzenia kodu źródłowego. Dlatego też np. stosowanie dwóch spacji przez innych programistów jest dla mnie źródłem ukrywanego zdziwienia od początku mojej kariery programowania.

Linter pozwala osobom mocno przywiązanim do własnego stylu formatowania na wymuszanie jego stosowania przez innych, najczęściej w sposób nieznoszący sprzeciwu. W efekcie programiści zaczynają spędzać zbyt wiele czasu na kłótniach związanych z różnymi stylami tworzenia kodu źródłowego, więc lepiej unikać wymuszania na wszystkich stosowania tych samych reguł formatowania kodu. Z mojego doświadczenia wynika, że programiści znajdą sobie inne powody do kłótni, a wymuszanie stosowania pewnego stylu kodu jest często wymówką do tego, by preferencje jednej osoby stały się obowiązkiem dla całego zespołu.

Bardzo często pomagam czytelnikom, którzy nie mogą poradzić sobie z przykładami przedstawionymi w moich książkach (jeżeli potrzebujesz pomocy, możesz do mnie napisać na adres adam@adam-freeman.com), więc każdego tygodnia spotykam się z różnymi stylami tworzenia kodu źródłowego. W głębi serca czuję, że każdy, kto nie stosuje się do moich preferencji, po prostu robi źle. Jednak zamiast wymuszać na nich stosowanie mojego stylu, pozwalam edytorowi kodu na ponowne sformatowanie kodu źródłowego. Taką funkcjonalność oferuje w zasadzie każdy edytor.

Moja rada brzmi: ostrożnie korzystaj z lintera i skoncentruj się na sprawach, które prowadzą do poważnych problemów. Kwestie związane z formatowaniem pozostaw każdemu do samodzielnego rozwiązania i wykorzystuj oferowaną przez edytor możliwość ponownego sformatowania kodu źródłowego, jeśli musisz pracować z kodem utworzonym przez innego programistę, który stosuje odmienne preferencje w tym zakresie.

Testy jednostkowe w TypeScriptie

Część frameworków testów jednostkowych zapewnia obsługę języka TypeScript, choć nie jest aż tak użyteczna, jak mogłoby się wydawać. Obsługa TypeScriptu dla testów jednostkowych oznacza możliwość zdefiniowania testów w plikach TypeScriptu i czasami automatyczną kompilację kodu TypeScriptu przed jego przetestowaniem. Testy jednostkowe są wykonywane przez uruchamianie niewielkich fragmentów aplikacji, co można zrobić jedynie za pomocą JavaScriptu, ponieważ środowisko uruchomieniowe JavaScriptu nie ma żadnych informacji o funkcjonalności TypeScriptu. W efekcie testy jednostkowe nie mogą być używane do testowania funkcjonalności TypeScriptu wymuszanej jedynie przez kompilator TypeScriptu.

W książce użyłem frameworka testów Jest, który jest łatwy w użyciu i zapewnia obsługę testów TypeScriptu. Ponadto po dodaniu pakietu dodatkowego gwarantuje, że pliki TypeScriptu w projekcie zostaną skompilowane na kod JavaScriptu przed wykonaniem testów. Aby zainstalować pakiety niezbędne do przeprowadzania testów jednostkowych, z poziomu katalogu *tools* w powłoce wydaj polecenia przedstawione na listingu 6.14.

Listing 6.14. Dodawanie niezbędnych pakietów do projektu

```
$ npm install --save-dev jest@26.6.3
$ npm install --save-dev ts-jest@26.4.4
```

Pakiet jest zawiera framework testów. Z kolei pakiet ts-jest to wtyczka dla frameworka Jest odpowiedzialna za kompilację plików TypeScriptu przed wykonaniem testów.

Podjęcie decyzji o wykonywaniu testów jednostkowych

Testy jednostkowe to temat wywołujący spore emocje. W tym rozdziale przyjąłem założenie, że chcesz je przeprowadzać, i dlatego pokażę, jak skonfigurować narzędzia oraz zastosować je w języku TypeScript. To zdecydowanie nie jest wprowadzenie do tematu testów jednostkowych. Nie podjąłem również wysiłku mającego na celu przekonanie sceptycznie nastawionych czytelników do zmiany zdania dotyczącego wartości testów jednostkowych. Jeżeli szukasz tego rodzaju wprowadzenia, dobry artykuł znajdziesz na stronie https://pl.wikipedia.org/wiki/Test_jednostkowy.

Sam lubię testy jednostkowe i stosuję je w moich projektach, ale nie zawsze i nie tak konsekwentnie, jak mógłbyś tego oczekiwać. Koncentruję się na przygotowaniu testów jednostkowych dla funkcji i funkcjonalności, o których wiem, że ich opracowanie będzie trudnym zadaniem, a ponadto mogą stać się źródłem błędów w projekcie. W tego rodzaju sytuacjach testy jednostkowe pomagają mi zebrać myśli i zastanowić się nad jak najlepszą implementacją potrzebnego rozwiązania. Przekonałem się, że myślenie o tym, co powinno zostać przetestowane, pomaga mi w opracowaniu rozwiązań potencjalnych problemów, najczęściej zanim jeszcze zacznę się zmagać z rzeczywistymi błędami i problemami.

Mając to wszystko na uwadze, testy jednostkowe należy uznać za narzędzie, a nie religię. Tylko Ty wiesz, ile testów jest potrzebnych w danym projekcie. Jeżeli testów jednostkowych nie uważasz za użyteczne lub jeżeli masz zupełnie inną metodologię, w stosowaniu której czujesz się lepiej, wtedy nie musisz przeprowadzać testów jednostkowych tylko dlatego, że tego rodzaju podejście stało się modne. (Z drugiej strony, jeśli nie masz lepszej metodologii i w ogóle nie testujesz aplikacji, wówczas zadanie wyszukiwania błędów przerywasz na użytkowników aplikacji, co można uznać za niedopuszczalne).

Konfigurowanie frameworka testów

Aby przeprowadzić konfigurację frameworka Jest, dodaj do katalogu *tools* plik o nazwie *jest.config.js* i zawartości przedstawionej na listingu 6.15.

Listing 6.15. Zawartość pliku *jest.config.js* w katalogu *tools*

```
module.exports = {
  "roots": ["src"],
  "transform": {"^.+\\.tsx?$": "ts-jest"}
}
```

Ustawienie *roots* jest używane do określenia położenia plików kodu źródłowego i testów jednostkowych. Właściwość *transform* jest używana do wskazania frameworkowi Jest, że pliki z rozszerzeniami *.ts* i *.tsx* powinny być przetwarzane za pomocą pakietu *ts-jest*, który gwarantuje odzwierciedlenie w testach zmian wprowadzonych w kodzie bez konieczności wyraźnego uruchamiania kompilatora (więcej informacji na temat plików TSX znajdziesz w rozdziale 14.).

Tworzenie testów jednostkowych

Testy są definiowane w plikach z rozszerzeniem *test.ts* i zwykle tworzone z kodem, którego dotyczą. Aby utworzyć prosty test jednostkowy dla przykładowej aplikacji, w katalogu *src* umieść plik o nazwie *calc.test.ts* z zawartością przedstawioną na listingu 6.16.

Listing 6.16. Zawartość pliku *calc.test.ts* w katalogu *src*

```
import { sum } from "./calc";

test("sprawdzenie wartości wynikowej", () => {
  let result = sum(10, 20, 30);
  expect(result).toBe(60);
});
```

Testy są definiowane za pomocą funkcji `test()` dostarczanej przez framework Jest. Argumenty funkcji `test()` to nazwa testu i funkcja odpowiedzialna za przeprowadzenie testu. Test jednostkowy na listingu 6.16 ma nazwę `sprawdzenie wartości wynikowej`, powoduje wywołanie funkcji `sum()` z trzema argumentami i analizuje wynik jej działania. Framework Jest dostarcza funkcję `expect()` przekazującą wynik i używającą funkcji dopasowania określającej oczekiwany wynik. W omawianym kodzie funkcją dopasowania jest `toBe()`, która podaje frameworkowi Jest oczekiwaną wartość wyniku. W tabeli 6.2 wymieniłem najużyteczniejsze funkcje dopasowania. (Pełną listę funkcji dopasowania znajdziesz na stronie <https://jestjs.io/docs/en/expect>).

Tabela 6.2. Użyteczne funkcje dopasowania oferowane przez framework Jest

Nazwa	Opis
<code>toBe(wartość)</code>	Metoda sprawdza, czy wynik jest taki sam jak podana wartość (to nie musi być ten sam obiekt)
<code>toEqual(obiekt)</code>	Metoda sprawdza, czy wynik jest takim samym obiektem jak podana wartość
<code>toMatch(wyrażenieRegularne)</code>	Metoda sprawdza, czy wynik pasuje do podanego wyrażenia regularnego
<code>toBeDefined()</code>	Metoda sprawdza, czy wynik został zdefiniowany
<code>toBeUndefined()</code>	Metoda sprawdza, czy wynik nie został zdefiniowany
<code>toBeNull()</code>	Metoda sprawdza, czy wynik ma wartość <code>null</code>
<code>toBeTruthy()</code>	Metoda sprawdza, czy wynik ma wartość prawdy
<code>toBeFalsy()</code>	Metoda sprawdza, czy wynik ma wartość fałszu
<code>toContain(podciąg)</code>	Metoda sprawdza, czy wynik zawiera podany podciąg tekstowy
<code>toBeLessThan(wartość)</code>	Metoda sprawdza, czy wynik jest mniejszy niż podana wartość
<code>toBeGreaterThan(wartość)</code>	Metoda sprawdza, czy wynik jest większy niż podana wartość

Uruchamianie frameworka testów

Testy jednostkowe mogą być uruchamiane jako jednorazowe zadanie lub w trybie monitorowania, w którym testy będą wykonywane po wykryciu wprowadzonych zmian. Przekonałem się, że tryb monitorowania jest najbardziej użyteczny, więc zwykle mam otwarte dwa okna powłoki. W pierwszym widzę dane wyjściowe wygenerowane przez kompilator, w drugim wynik wykonania testów jednostkowych. Aby uruchomić testy, należy w nowym oknie powłoki przejść do katalogu *tools* i wydać polecenie przedstawione na listingu 6.17.

Listing 6.17. Uruchamianie frameworka testów jednostkowych w trybie monitorowania

```
$ npx jest --watchAll
```

Framework Jest zostanie uruchomiony, odszuka pliki testów w projekcie, wykona je i wygeneruje następujące dane wyjściowe:

```
PASS  src/calc.test.ts
  ✓ sprawdzenie wartości wynikowej (3ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        3.214s
Ran all test suites.
Watch Usage
  › Press f to run only failed tests.
  › Press o to only run tests related to changed files.
  › Press p to filter by a filename regex pattern.
  › Press t to filter by a test name regex pattern.
  › Press q to quit watch mode.
  › Press Enter to trigger a test run.
```

Z tych danych jasno wynika, że framework Jest odkrył jeden test i wykonał go z powodzeniem. Po zdefiniowaniu kolejnych testów lub wprowadzeniu jakiegokolwiek zmiany w kodzie źródłowym aplikacji framework Jest ponownie wykona testy i wygeneruje następne podsumowanie. Jeśli chcesz zobaczyć, co się dzieje w przypadku testu zakończonego niepowodzeniem, w testowanej funkcji *sum()* wprowadź zmianę przedstawioną na listingu 6.18.

Listing 6.18. Wprowadzenie zmiany w kodzie pliku *calc.ts* w katalogu *src*

```
export function sum(...vals: number[]): number {
  return vals.reduce((total, val) => total += val) + 10;
}
```

Funkcja *sum()* nie zwraca już wartości oczekiwanej przez test jednostkowy, więc wygenerowany będzie następujący komunikat ostrzeżenia:

```
FAIL  src/calc.test.ts
  ✕ sprawdzenie wartości wynikowej (6ms)
    · sprawdzenie wartości wynikowej
```



```
expect(received).toBe(expected) // Object.is equality
```

```
Expected: 60
```

```
Received: 70
```

```

3 | test("sprawdzenie wartości wynikowej", () => {
4 |     let result = sum(10, 20, 30);
> 5 |     expect(result).toBe(60);
    |                       ^
6 | });
```

```
at Object.<anonymous> (src/calc.test.ts:5:20)
```

```
Test Suites: 1 failed, 1 total
```

```
Tests:      1 failed, 1 total
```

```
Snapshots:  0 total
```

```
Time:       4.726s
```

```
Ran all test suites.
```

```
Watch Usage: Press w to show more.
```

Te dane wyjściowe wyraźnie pokazują rozbieżność między wynikami wygenerowanymi przez kod a oczekiwanymi przez test jednostkowy. Zakończony niepowodzeniem test jednostkowy może być zaliczony po poprawieniu kodu źródłowego w taki sposób, aby generował wynik zgodny z oczekiwanym przez test. Ewentualnie jeśli przeznaczenie kodu źródłowego uległo zmianie, wówczas należy uaktualnić test jednostkowy, aby to odzwierciedlał. Na listingu 6.19 przedstawiłem zmodyfikowaną wersję testu jednostkowego.

Listing 6.19. Zmodyfikowany test jednostkowy w pliku *calc.test.ts* w katalogu *src*

```
import { sum } from "./calc";

test("sprawdzenie wartości wynikowej", () => {
    let result = sum(10, 20, 30);
    expect(result).toBe(70);
});
```

Po zapisaniu zmiany framework Jest ponownie wykona test, który tym razem zakończy się sukcesem:

```

PASS  src/calc.test.ts
  ✓ sprawdzenie wartości wynikowej (3ms)
Test Suites: 1 passed, 1 total
Tests:      1 passed, 1 total
Snapshots:  0 total
Time:       5s
Ran all test suites.
```

```
Watch Usage: Press w to show more.
```

Podsumowanie

W tym rozdziale wprowadziłem trzy narzędzia często używane podczas programowania w języku TypeScript. Debugger Node.js oferuje użyteczny sposób analizy stanu aplikacji podczas jej wykonywania. Linter pomaga w unikaniu najczęściej popełnianych błędów programistycznych, które pozostają niewykryte przez kompilator, a mimo to mogą powodować problemy. Z kolei framework testów jednostkowych jest używany do sprawdzenia, czy kod działa zgodnie z oczekiwaniami. W następnym rozdziale zacznę dokładniej omawiać funkcjonalność języka TypeScript, a na pierwszy ogień pójdzie statyczne sprawdzanie typu.

CZĘŚĆ II



Praca z językiem TypeScript



Typowanie statyczne

W tym rozdziale zamierzam wprowadzić kluczowe funkcje TypeScriptu przeznaczone do pracy z typami danych. Omówione tutaj funkcje stanowią ważne podstawy do programowania w języku TypeScript i są punktem wyjścia dla funkcji zaawansowanych, które będą omówione w dalszej części książki.

Zacznę od pokazania, jak typy TypeScriptu różnią się od typów JavaScriptu. Pokażę również, że kompilator TypeScriptu ma możliwość ustalenia typu danych na podstawie kodu źródłowego. Później przejdę do omówienia funkcjonalności zapewniającej dokładniejszą kontrolę nad typami danych poprzez dostarczenie kompilatorowi informacji o oczekiwanym zachowaniu sekcji kodu bądź też przez zmianę sposobu, w jaki kompilator zarządza typami. Streszczenie materiału przedstawionego w rozdziale znajdziesz w tabeli 7.1.

Tabela 7.1. Streszczenie materiału przedstawionego w rozdziale

Problem	Rozwiązanie	Listing
Określenie typu	Użyj adnotacji typu lub pozostaw kompilatorowi ustalenie typu	Od 10 do 13
Sprawdzanie typu ustalonego przez kompilator	Włącz opcję <code>declarations</code> kompilatora i sprawdź skompilowany kod	14 i 15
Zgoda na użycie dowolnego typu	Określ typ <code>any</code> lub <code>unknown</code>	Od 16 do 19 oraz 29 i 30
Uniemożliwienie kompilatorowi używania typu <code>any</code>	Włącz opcję kompilatora <code>noImplicitAny</code>	20
Łączenie typów	Użyj unii typów	21 i 22
Zastąpienie typu oczekiwanego przez kompilator	Użyj asercji typu	Od 23 do 25
Testowanie wartości typów podstawowych	Użyj operatora <code>typeof</code> jako wartownika typu	Od 26 do 28

Tabela 7.1. Streszczenie materiału przedstawionego w rozdziale (ciąg dalszy)

Problem	Rozwiązanie	Listing
Uniemożliwienie użycia <code>null</code> lub <code>undefined</code> jako wartości innych typów	Użyj opcji kompilatora <code>strictNullChecks</code>	Od 31 do 33
Nadpisanie ustawień kompilatora w celu usunięcia wartości <code>null</code> z unii	Użyj asercji innych niż <code>null</code> lub wartownika typu	34 i 35
Umożliwienie użycia zmiennej, która nie ma przypisanej wartości	Użyj konkretnej asercji przypisania	36 i 37

W tabeli 7.2 wymieniałem opcje kompilatora TypeScriptu użyte w rozdziale.

Tabela 7.2. Opcje kompilatora TypeScriptu użyte w rozdziale

Opcja	Opis
<code>declaration</code>	Ta opcja generuje pliki deklaracji typu, które pomagają w zrozumieniu, w jaki sposób zostały ustalone typy dla kodu JavaScriptu. Wspomniane pliki zostaną szczegółowo omówione w rozdziale 14.
<code>noImplicitAny</code>	Ta opcja uniemożliwia niejawne używanie typu <code>any</code> , który kompilator stosuje, gdy nie jest w stanie ustalić konkretnego typu
<code>outDir</code>	Ta opcja określa katalog, w którym zostaną umieszczone wygenerowane pliki JavaScriptu
<code>rootDir</code>	Ta opcja określa katalog główny używany przez kompilator do wyszukiwania plików TypeScriptu
<code>strictNullChecks</code>	Ta opcja uniemożliwia używanie <code>null</code> i <code>undefined</code> jako wartości dla innych typów
<code>target</code>	Ta opcja określa docelową wersję języka JavaScript, dla której kompilator wygeneruje kod

Przygotowanie projektu

Aby przygotować się do utworzenia projektu zaprezentowanego w rozdziale, w dogodnym miejscu utwórz katalog o nazwie `types`. Następnie z poziomu powłoki przejdź do katalogu `types` i wydaj w nim polecenie przedstawione na listingu 7.1, które zainicjalizuje katalog do użycia z menedżerem pakietów Node.js.

-
- **Wskazówka** Przykładowy projekt dla tego rozdziału, a także projekty dla pozostałych rozdziałów książki zostały umieszczone w archiwum dostępnym pod adresem <https://ftp.helion.pl/przyklady/tys4o2.zip>.
-

Listing 7.1. Inicjalizacja menedżera pakietów Node

```
$ npm init --yes
```

Z poziomu katalogu *types* wydaj polecenie przedstawione na listingu 7.2 i tym samym dodaj pakiety wymagane w projekcie dla tego rozdziału.

Listing 7.2. Dodawanie do projektu niezbędnych pakietów

```
$ npm install --save-dev typescript@4.2.2
$ npm install --save-dev tsc-watch@4.2.9
```

Aby skonfigurować kompilator TypeScriptu, dodaj do katalogu *types* plik o nazwie *tsconfig.json* z zawartością przedstawioną na listingu 7.3.

Listing 7.3. Zawartość pliku *tsconfig.json* w katalogu *types*

```
{
  "compilerOptions": {
    "target": "es2020",
    "outDir": "./dist",
    "rootDir": "./src"
  }
}
```

Te ustawienia konfiguracyjne nakazują kompilatorowi TypeScriptu wygenerowanie kodu przeznaczonego dla najnowszych implementacji JavaScriptu, w katalogu *src* kompilator będzie szukał plików TypeScriptu, a w katalogu *dist* umieści pliki z wygenerowanym kodem. Jeżeli chcesz przygotować konfigurację pozwalającą na łatwe uruchamianie kompilatora, dodaj do pliku *package.json* opcję konfiguracyjną przedstawioną na listingu 7.4.

Listing 7.4. Konfigurowanie menedżera pakietów Node.js w pliku *package.json* w katalogu *types*

```
{
  "name": "types",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "tsc-watch --onSuccess \"node dist/index.js\""
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "tsc-watch": "^4.2.9",
    "typescript": "^4.2.2"
  }
}
```

Aby zdefiniować punkt wejścia do projektu, utwórz w katalogu *types* podkatalog *src* i umieść w nim plik o nazwie *index.ts* z zawartością przedstawioną na listingu 7.5.

Listing 7.5. Zawartość pliku *index.ts* w katalogu *src*

```
console.log("Witaj, TypeScript");
```

Z poziomu katalogu *types* wydaj polecenie przedstawione na listingu 7.6, które spowoduje uruchomienie kompilatora TypeScriptu.

Listing 7.6. Uruchamianie kompilatora TypeScriptu

```
$ npm start
```

Kompilator skompiluje zawartość pliku *index.ts*, wykona wygenerowany kod JavaScriptu, a następnie przejdzie do trybu monitorowania i wyświetli następujące dane wyjściowe:

```
6:43:06 AM - Starting compilation in watch mode...
```

```
6:43:08 AM - Found 0 errors. Watching for file changes.  
Witaj, TypeScript
```

Typy statyczne

Jak już wyjaśniłem w rozdziale 4., JavaScript jest językiem typowanym dynamicznie. Dla programistów posiadających doświadczenie w programowaniu w innym języku największą trudnością w JavaScriptcie jest to, że to *wartości*, a nie zmienne mają przypisany typ. Jeżeli chcesz sobie przypomnieć, jak to działa, zastąp zawartość pliku *index.ts* kodem przedstawionym na listingu 7.7.

Listing 7.7. Nowa zawartość pliku *index.ts* w katalogu *src*

```
let myVar;  
  
myVar = 12;  
myVar = "Witaj";  
myVar = true;
```

Typ zmiennej *myVar* zmienia się na podstawie przypisanej jej wartości. Słowo kluczowe *typeof* w JavaScriptcie pozwala na ustalenie typu, jak pokazałem na listingu 7.8.

Listing 7.8. Wyświetlanie typu zmiennej w kodzie pliku *index.ts* w katalogu *src*

```
let myVar;  
console.log(`${myVar} = ${typeof myVar}`);  
myVar = 12;  
console.log(`${myVar} = ${typeof myVar}`);  
myVar = "Witaj";  
console.log(`${myVar} = ${typeof myVar}`);  
myVar = true;  
console.log(`${myVar} = ${typeof myVar}`);
```

Po zapisaniu zmian w pliku i ponownym wykonaniu skompilowanego kodu zostaną wyświetlone następujące dane wyjściowe:


```

undefined = undefined
12 = number
Witaj = string
true = boolean

```

Pierwsze polecenie w kodzie na listingu 7.8 definiuje zmienną bez przypisywania jej wartości, co oznacza, że jej typem jest `undefined`. Zmienna tego typu zawsze będzie miała wartość `undefined`, jak widać w wygenerowanych danych wyjściowych.

Wartość `12` jest typu `number` i po przypisaniu jej zmiennej typ danych tej zmiennej ulega zmianie. Z kolei wartość `Witaj` jest typu `string`, natomiast wartość `true` jest typu `boolean` — to kolejne typy danych zmiennej po przypisaniu jej kolejnych wartości. W JavaScriptcie nie trzeba podawać typu danych, ponieważ będzie on ustalony automatycznie na podstawie wartości. Dla przypomnienia w tabeli 7.3 wymienię wbudowane typy danych dostarczane przez JavaScript.

Tabela 7.3. Wbudowane typy danych w JavaScriptcie

Nazwa	Opis
<code>number</code>	Ten typ jest używany do przedstawienia wartości liczbowej
<code>string</code>	Ten typ jest używany do przedstawienia danych tekstowych
<code>boolean</code>	Ten typ wykorzystuje wartości <code>true</code> i <code>false</code>
<code>symbol</code>	Ten typ wykorzystuje unikatowe wartości stałe, takie jak klucze w kolekcji
<code>null</code>	Ten typ może mieć przypisaną tylko wartość <code>null</code> i służy do wskazania nieistniejącego lub nieprawidłowego odwołania
<code>undefined</code>	Ten typ jest używany w przypadku, gdy zmienna została zdefiniowana, ale nie przypisano jej wartości
<code>object</code>	Ten typ służy do przedstawiania wartości złożonych, utworzonych z poszczególnych właściwości i wartości

Typy dynamiczne oferują ogromną elastyczność, choć jednocześnie prowadzą do występowania problemów podobnych do tego z listingu 7.9, którego kod ma zastąpić dotychczasową zawartość pliku *index.ts*. Mamy tutaj funkcję i kilka wywołujących ją poleceń.

Listing 7.9. Definiowanie funkcji w kodzie pliku *index.ts* w katalogu *src*

```

function calculateTax(amount) {
    return amount * 1.2;
}

console.log(`${12} = ${calculateTax(12)}`);
console.log(`${\"Witaj\"} = ${calculateTax(\"Witaj\")}`);
console.log(`${true} = ${calculateTax(true)}`);

```

Typ parametru funkcji również jest dynamiczny, co oznacza, że wywołanie `calculateTax()` może otrzymać wartość dowolnego typu. Polecenie znajdujące się po definicji funkcji wywołuje ją z wartościami typów `number`, `string` i `boolean`, a następnie zostają wygenerowane przedstawione tutaj dane wyjściowe:

```
12 = 14.399999999999999
Witaj = NaN
true = 1.2
```

Z perspektywy kodu JavaScriptu nie ma nic złego w przedstawionym przykładzie. Parametr funkcji może otrzymywać wartość dowolnego typu, a JavaScript prawidłowo obsługuje te typy. Jednak funkcja `calculateTax()` została opracowana z założeniem, że będzie otrzymywała wartości jedynie typu `number`, stąd tylko pierwszy wynik ma sens. (Drugi wynik, `NaN`, oznacza, że wartością nie jest liczba [ang. *not a number*], a trzeci powstał na skutek koercji wartości `true` na liczbę 1 użytą później w obliczeniach — w rozdziale 4. znajdziesz więcej informacji na temat koercji typu w JavaScriptcie).

Bardzo łatwo można zrozumieć poczynione założenia o typie parametru funkcji, analizując kod znajdujący się obok używającego jej polecenia. Znacznie trudniej będzie, jeśli funkcja została utworzona przez innego programistę i znajduje się gdzieś głęboko w skomplikowanym projekcie lub pakiecie.

Tworzenie typu statycznego za pomocą adnotacji typu

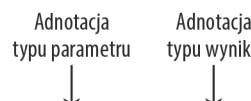
Większość programistów jest przyzwyczajona do typów statycznych. Funkcjonalność typów statycznych w TypeScriptie opiera się na wyraźnie definiowanych założeniach i pozwala kompilatorowi zgłaszać błąd w przypadku użycia innego typu danych. Do zdefiniowania typu statycznego są używane tzw. *adnotacje typu*, jak pokazałem na listingu 7.10.

Listing 7.10. Używanie adnotacji typu w kodzie pliku `index.ts` w katalogu `src`

```
function calculateTax(amount: number): number {
    return amount * 1.2;
}

console.log(`12 = ${calculateTax(12)}`);
console.log(`"Witaj" = ${calculateTax("Witaj")}`);
console.log(`true = ${calculateTax(true)}`);
```

W kodzie przedstawionym na listingu 7.10 znajdują się dwie adnotacje zdefiniowane przez umieszczenie dwukropka i nazwy typu statycznego, jak pokazałem na rysunku 7.1.



```
function calculateTax(amount: number): number {
```

Rysunek 7.1. Stosowanie adnotacji typu

W omawianym przykładzie adnotacja typu parametru funkcji wskazuje kompilatorowi, że dana funkcja akceptuje jedynie wartości typu `number`. Adnotacja, która znajduje się po sygnaturze funkcji, określa typ wyniku i wskazuje kompilatorowi, że funkcja zwraca wartość jedynie typu `number`.

Po skompilowaniu tego kodu kompilator TypeScriptu analizuje typ danych i wartości przekazane funkcji `calculateTax()`, a następnie ustala, że niektóre wartości są nieprawidłowego typu. Dlatego też zostają wygenerowane następujące dane wyjściowe:

```
src/index.ts(6,42): error TS2345: Argument of type '"Witaj"' is not assignable to
parameter of type 'number'.
src/index.ts(7,39): error TS2345: Argument of type 'true' is not assignable to parameter
of type 'number'.
```

■ **Wskazówka** Jeżeli używany przez Ciebie edytor kodu zapewnia dobrą obsługę języka TypeScript, komunikaty ostrzeżenia mogą być wyświetlane również bezpośrednio w edytorze. Do programowania w TypeScriptie używam Visual Studio Code, który sygnalizuje problemy bezpośrednio w oknie edytora.

Adnotacje typu mogą być stosowane również dla zmiennych i stałych, jak pokazałem na listingu 7.11.

Listing 7.11. Zastosowanie adnotacji typu dla zmiennej w kodzie pliku `index.ts` w katalogu `src`

```
function calculateTax(amount: number): number {
    return amount * 1.2;
}

let price: number = 100;
let taxAmount: number = calculateTax(price);
let halfShare: number = taxAmount / 2;

console.log(`Pełna kwota z podatkiem: ${taxAmount}`);
console.log(`Połowa kwoty z podatkiem: ${halfShare}`);
```

Adnotacje są stosowane po nazwie za pomocą dwukropka i typu, podobnie jak w przypadku adnotacji używanych dla funkcji. Wszystkie trzy zmienne w kodzie na listingu 7.11 otrzymały adnotacje wskazujące kompilatorowi, że będą używane z wartościami typu `number`. Po wykonaniu ten kod powoduje wygenerowanie następujących danych wyjściowych:

```
Pełna kwota z podatkiem: 120
Połowa kwoty z podatkiem: 60
```

Używanie niejawnie zdefiniowanego typu statycznego

Kompilator TypeScriptu ma możliwość ustalenia typu, co oznacza, że możesz skorzystać z zalet statycznego typowania, unikając przy tym konieczności używania adnotacji, jak pokazałem w przykładzie na listingu 7.12.

Listing 7.12. Wykorzystanie niejawnego typowania w kodzie pliku `index.ts` w katalogu `src`

```
function calculateTax(amount: number) {
    return amount * 1.2;
}

let price = 100;
```

```
let taxAmount = calculateTax(price);
let halfShare = taxAmount / 2;

console.log(`Pełna kwota z podatkiem: ${taxAmount}`);
console.log(`Połowa kwoty z podatkiem: ${halfShare}`);
```

Kompilator TypeScriptu jest w stanie określić typ zmiennej `price` na podstawie literału wartości przypisanego jej w chwili definiowania zmiennej. Kompilator uznaje 100 za wartość typu `number` i traktuje zmienną `price`, jakby została zdefiniowana z adnotacją typu `number`. Dlatego też jest to akceptowalna wartość do użycia jako argument funkcji `calculateTax()`.

Kompilator jest również w stanie ustalić wynik działania funkcji `calculateTax()`, ponieważ wie, że akceptowane są tylko parametry typu `number`. Wartość 1.2 jest typu `number`, podobnie jak wynik użycia operatora mnożenia z dwiema wartościami typu `number`.

Wynik działania funkcji zostaje przypisany zmiennej `taxAmount`, której typ kompilator również określa jako `number`. Ponadto kompilator zna typ wartości wygenerowanej po użyciu operatora dzielenia z dwiema wartościami typu `number` i ustala typ zmiennej `halfShare`.

Kompilator TypeScriptu nie generuje żadnych komunikatów w przypadku prawidłowego używania typów, więc łatwo można zapomnieć o przeprowadzanej operacji sprawdzania kodu. Jeżeli chcesz się dowiedzieć, co się stanie, gdy typ będzie nieprawidłowo ustalony, w kodzie pliku `index.ts` wprowadź zmianę przedstawioną na listingu 7.13.

Listing 7.13. Zmiana typu wyniku w kodzie pliku `index.ts` w katalogu `src`

```
function calculateTax(amount: number) {
    return (amount * 1.2).toFixed(2);
}

let price = 100;
let taxAmount = calculateTax(price);
let halfShare = taxAmount / 2;

console.log(`Pełna kwota z podatkiem: ${taxAmount}`);
console.log(`Połowa kwoty z podatkiem: ${halfShare}`);
```

Metoda `toFixed()` formatuje wartość typu `number` w taki sposób, aby zawierała określoną liczbę cyfr po przecinku. Wynikiem działania metody `toFixed()` jest ciąg tekstowy (typ `string`), co oznacza zmianę typu wyniku funkcji `calculateTax()`. Gdy kompilator pracuje z łańcuchem typów, napotyka użycie operatora dzielenia z operandami typów `string` i `number`:

```
...
let halfShare = taxAmount / 2;
...
```

Jest to dozwolone w języku JavaScript i taka operacja powoduje użycie koercji typu, jak to dokładnie omówiłem w rozdziale 3. W omawianym przykładzie wartość typu `string` zostanie skonwertowana na wartość typu `number`. Skutkiem będzie dzielenie dwóch wartości typu `number` lub wartości `NaN`, jeśli nie uda się skonwertować wartości typu `string` na `number`.

Z kolei w języku TypeScript automatyczna koercja typu jest niedozwolona, więc kompilator generuje komunikat błędu, zamiast próbować przeprowadzać konwersję wartości:

```
src/index.ts(7,17): error TS2362: The left-hand side of an arithmetic operation must be of type 'any', 'number', 'bigint' or an enum type.
```

Wprowadzie kompilator TypeScriptu nie zabrania używania funkcji typów w JavaScriptcie, ale generuje błędy po napotkaniu poleceń, których wykonanie może prowadzić do problemów.

Będą się zdarzać sytuacje, zwłaszcza jeśli dopiero rozpocząłeś poznawanie języka TypeScript, w których otrzymasz błędy ze względu na ustalenie przez kompilator nieoczekiwanego w danej sytuacji typu danych. W niemalże każdym przypadku kompilator prawidłowo określa typ, ale istnieje użyteczna funkcja kompilatora, która po włączeniu ujawnia typy wykorzystane w kodzie, jak pokazałem na listingu 7.14.

Listing 7.14. Konfigurowanie kompilatora TypeScriptu w pliku *tsconfig.json* w katalogu *types*

```
{
  "compilerOptions": {
    "target": "es2020",
    "outDir": "./dist",
    "rootDir": "./src",
    "declaration": true
  }
}
```

Ustawienie `declaration` nakazuje kompilatorowi wygenerowanie z kodem JavaScriptu także plików zawierających informacje o typie. Dokładne omówienie tych plików znajdziesz w rozdziale 14., natomiast teraz wystarczy wiedzieć, że pomagają one kompilatorowi w ustalaniu typów, choć nie jest to ich podstawowe przeznaczenie. Zmiana konfiguracji kompilatora będzie uwzględniona po jego ponownym uruchomieniu. Aby wywołać kompilację, dodaj do pliku *index.js* polecenie przedstawione na listingu 7.15, a następnie zapisz ten plik.

Listing 7.15. Dodawanie polecenia do kodu pliku *index.ts* w katalogu *src*

```
function calculateTax(amount: number) {
  return (amount * 1.2).toFixed(2);
}

let price = 100;
let taxAmount = calculateTax(price);
let halfShare = taxAmount / 2;

console.log(`Cena: ${price}`);
console.log(`Pełna kwota z podatkiem: ${taxAmount}`);
console.log(`Połowa kwoty z podatkiem: ${halfShare}`);
```

Po uruchomieniu kompilatora wygeneruje on w katalogu *dist* plik o nazwie *index.d.ts* z następującą zawartością:

```
...
declare function calculateTax(amount: number): string;
declare let price: number;
declare let taxAmount: string;
declare let halfShare: number;
...
```

Przeznaczenie słowa kluczowego `declare` — i samego pliku — dokładnie wyjaśnię w rozdziale 14. Ten plik ujawnia typy ustalone przez kompilator dla poleceń zdefiniowanych w kodzie na listingu 7.15. Jak widać, typ wartości zwrótej funkcji `calculateTax()` i zmiennej

taxAmount to string. Gdy kompilator wygeneruje komunikat błędu, analiza plików wygenerowanych po przypisaniu wartości true ustawieniu declaration w konfiguracji kompilatora może naprawdę pomóc w ustaleniu źródła problemu, zwłaszcza jeśli nie znajdujesz żadnych oczywistych powodów tego błędu.

Używanie typu any

TypeScript nie zabrania korzystania z elastycznego systemu typów języka JavaScript, choć próbuje chronić programistę przed przypadkowym użyciem tego systemu. Aby umożliwić używanie wszystkich typów jako parametrów funkcji i wyników ich działania lub pozwolić na przypisanie dowolnego typu zmiennym i stałym, TypeScript oferuje typ o nazwie any, którego przykład użycia przedstawiłem na listingu 7.16.

Listing 7.16. Użycie typu any w kodzie pliku index.ts w katalogu src

```
function calculateTax(amount: any): any {
    return (amount * 1.2).toFixed(2);
}

let price = 100;
let taxAmount = calculateTax(price);
let halfShare = taxAmount / 2;

console.log(`Cena: ${price}`);
console.log(`Pełna kwota z podatkiem: ${taxAmount}`);
console.log(`Połowa kwoty z podatkiem: ${halfShare}`);
```

Te adnotacje wskazują kompilatorowi, że parametr amount może akceptować wartość, a wynik działania funkcji może być dowolnego typu. Użycie typu any uniemożliwia kompilatorowi zgłaszanie błędów wygenerowanych podczas kompilacji kodu przedstawionego na listingu 7.15, ponieważ nie będzie już sprawdzał, czy wynik działania funkcji calculateTax() jest możliwy do użycia z operatorem dzielenia. Kod będzie wykonany z powodzeniem, a JavaScript automatycznie skonwertuje operandy dzielenia na wartości typu number, aby wartość typu string zwrócona przez funkcję calculateTax() została skonwertowana na number. Wynikiem wykonania kodu będą następujące dane wyjściowe:

```
Cena: 100
Pełna kwota z podatkiem: 120.00
Połowa kwoty z podatkiem: 60
```

Gdy używasz typu any, bierzesz odpowiedzialność za zagwarantowanie, że kod prawidłowo korzysta z typów, podobnie jak w przypadku czystego kodu JavaScriptu. Na listingu 7.17 przedstawiłem modyfikację kodu funkcji calculateTax() polegającą na dodaniu liter zł po wartości walutowej.

Listing 7.17. Zmiana wyniku działania funkcji w kodzie pliku index.ts w katalogu src

```
function calculateTax(amount: any): any {
    return `${(amount * 1.2).toFixed(2)} zł`;
}
```

```
let price = 100;
let taxAmount = calculateTax(price);
let halfShare = taxAmount / 2;

console.log(`Cena: ${price}`);
console.log(`Pełna kwota z podatkiem: ${taxAmount}`);
console.log(`Połowa kwoty z podatkiem: ${halfShare}`);
```

Wynik wykonania funkcji nie może być skonwertowany na wartość typu `number`, więc po uruchomieniu ten kod powoduje wygenerowanie następujących danych wyjściowych:

```
Cena: 100
Pełna kwota z podatkiem: 120.00 zł
Połowa kwoty z podatkiem: NaN
```

Jedną z konsekwencji używania typu `any` jest możliwość jego przypisania wszystkim pozostałym typom bez wywoływania ostrzeżenia kompilatora. Spójrz na przykładowy program przedstawiony na listingu 7.18.

Listing 7.18. Przypisywanie typu `any` w kodzie pliku `index.ts` w katalogu `src`

```
function calculateTax(amount: any): any {
    return `${(amount * 1.2).toFixed(2)} zł`;
}

let price = 100;
let taxAmount = calculateTax(price);
let halfShare = taxAmount / 2;

console.log(`Cena: ${price}`);
console.log(`Pełna kwota z podatkiem: ${taxAmount}`);
console.log(`Połowa kwoty z podatkiem: ${halfShare}`);

let newResult: any = calculateTax(200);
let myNumber: number = newResult;
console.log(`Wartość liczbowa: ${myNumber.toFixed(2)}`);
```

Wartość zmiennej `newResult` typu `any` jest przypisywana typowi `number` bez wygenerowania ostrzeżenia przez kompilator. W trakcie działania programu funkcja `calculateTax()` zwraca wynik w postaci wartości typu `string`, która nie definiuje metody `toFixed()` wywołanej w ostatnim poleceniu kodu przedstawionego na listingu 7.18. Wynikiem jest więc następujący błąd wygenerowany po uruchomieniu aplikacji:

```
console.log(`Wartość liczbowa: ${myNumber.toFixed(2)}`);
                                     ^
TypeError: myNumber.toFixed is not a function
```

Kompilator próbuje potraktować wartość typu `any` jak wartość typu `number`, co oznacza niedopasowanie typu w trakcie działania programu. Typ `any` pozwala w pełni wykorzystać funkcjonalności typów JavaScriptu, co może być użyteczne; z drugiej strony skutkiem mogą być nieoczekiwane wyniki po przeprowadzeniu automatycznej koercji typów w trakcie działania aplikacji.

- **Wskazówka** TypeScript oferuje również typ `unknown` zapewniający wyraźny dostęp do funkcjonalności typu dynamicznego przy jednoczesnym ograniczeniu jego przypadkowego użycia, co dokładnie omówię w dalszej części rozdziału.

Używanie niejawnie zdefiniowanego typu `any`

Kompilator TypeScript będzie używał typu `any` podczas niejawnego przypisywania typów oraz gdy nie potrafi ustalić konkretnego typu do zastosowania. To znacznie ułatwia selektywne wykorzystanie TypeScriptu w istniejącym projekcie JavaScriptu i może usprawnić pracę z pakietami JavaScriptu opracowanymi przez podmioty zewnętrzne. W kodzie przedstawionym na listingu 7.19 usunąłem adnotację typu z parametru `calculateTax()`.

Listing 7.19. Usunięcie adnotacji i definiowanie zmiennej w kodzie pliku `index.ts` w katalogu `src`

```
function calculateTax(amount): any {
    return `${(amount * 1.2).toFixed(2)} zł`;
}

let price = 100;
let taxAmount = calculateTax(price);
let halfShare = taxAmount / 2;

let personVal = calculateTax("Bartek");

console.log(`Cena: ${price}`);
console.log(`Pełna kwota z podatkiem: ${taxAmount}`);
console.log(`Połowa kwoty z podatkiem: ${halfShare}`);
console.log(`Imię: ${personVal}`);
```

Kompilator użyje niejawnie typu `any` dla parametru funkcji, ponieważ nie jest w stanie ustalić lepszego typu do zastosowania. Dlatego też nie zostanie wygenerowany komunikat błędu przez kompilator, gdy funkcja będzie wywołana z argumentem typu `string`, co spowoduje wygenerowanie następujących danych wyjściowych:

```
Cena: 100
Pełna kwota z podatkiem: 120.00 zł
Połowa kwoty z podatkiem: NaN
Imię: NaN zł
```

Możesz potwierdzić niejawne użycie typu `any` przez analizę zawartości pliku `index.d.ts` w katalogu `dist`, który to plik będzie zawierał przedstawiony tutaj opis funkcji `calculateTax()`:

```
...
declare function calculateTax(amount: any): any;
...
```

Wyłączenie niejawnego używania typu `any`

Jawne używanie typu `any` powoduje uniknięcie sprawdzania typu, co może okazać się użyteczne, pod warunkiem że będzie stosowane z zachowaniem ostrożności. Umożliwienie kompilatorowi niejawnego stosowania typu `any` powoduje powstanie luk w sprawdzaniu typu, których możesz nawet nie zauważyć — niwelują one korzyści płynące z używania języka TypeScript.

Dobłą praktyką jest blokowanie możliwości niejawnego używania typu `any` za pomocą ustawienia `noImplicitAny`, jak przedstawiłem w kodzie na listingu 7.20. (Niejawne używanie `any` zostaje również wyłączone w przypadku stosowania opcji `strict` kompilatora, o czym już wspominałem w tabeli 7.3).

Listing 7.20. Konfigurowanie kompilatora w pliku `tsconfig.json` w katalogu `types`

```
{
  "compilerOptions": {
    "target": "es2020",
    "outDir": "./dist",
    "rootDir": "./src",
    "declaration": true,
    "noImplicitAny": true
  }
}
```

Po zapisaniu zmiany wprowadzonej w pliku konfiguracyjnym kompilatora kod zostanie ponownie skompilowany i otrzymasz następujący komunikat błędu:

```
src/index.ts(1,23): error TS7006: Parameter 'amount' implicitly has an 'any' type.
```

Kompilator wyświetla ostrzeżenie, gdy nie jest w stanie określić konkretnego typu, choć nie uniemożliwia to jawnego zastosowania typu `any`.

Używanie unii typów

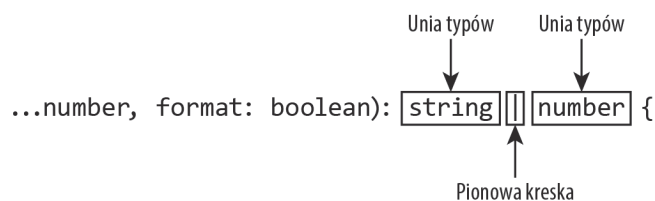
Na jednym końcu spektrum bezpieczeństwa typów mamy funkcjonalność `any`, która zapewnia pełną dowolność. Na drugim końcu znajdują się adnotacje typów stosowane dla pojedynczego typu, które zawężają zakres dozwolonych wartości. Między tymi dwoma rozwiązaniami granicznymi TypeScript oferuje tzw. *unie typów*, które określają zbiór typów. W kodzie przedstawionym na listingu 7.21 została zdefiniowana funkcja, która zwraca różne typy danych i używa adnotacji typu z unią typów w celu opisanego wyniku kompilatorowi.

Listing 7.21. Używanie unii typów w kodzie pliku `index.ts` w katalogu `src`

```
function calculateTax(amount: number, format: boolean): string | number {
  const calcAmount = amount * 1.2;
  return format ? `${calcAmount.toFixed(2)} zł` : calcAmount;
}

let taxNumber = calculateTax(100, false);
let taxString = calculateTax(100, true);
```

Typ zwrócony przez funkcję `calculateTax()` jest unią typów `string` i `number`, która została zdefiniowana za pomocą pionowej kreski umieszczanej między nazwami typów, jak pokazałem na rysunku 7.2. Wprawdzie unia zdefiniowana na listingu 7.21 używa dwóch typów, ale w unii można łączyć dowolną liczbę typów.



Rysunek 7.2. Definiowanie unii typów

Trzeba koniecznie wiedzieć, że unia typów jest traktowana jako oddzielny typ, a jej funkcjonalność to wynik złączenia poszczególnych typów wykorzystanych w unii. Oznacza to, że typ zmiennej `taxNumber` na listingu 7.21 to `string | number`, a nie `number`, nawet pomimo zwracania przez funkcję `calculateTax()` wartości liczbowej, gdy argument `boolean` ma wartość `false`. Aby podkreślić efekt unii typów, w kodzie na listingu 7.22 unia została wyraźnie podana jako typ wartości.

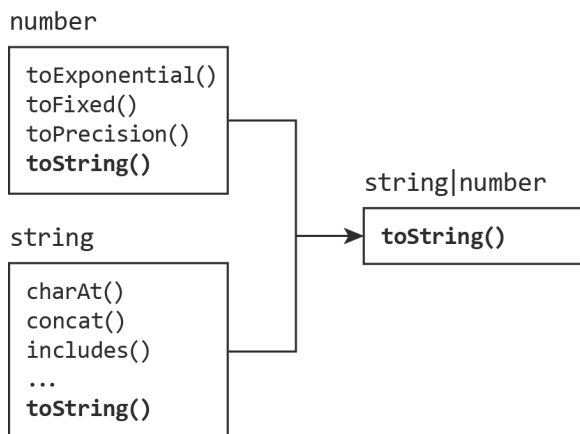
Listing 7.22. Wyraźne deklarowanie unii typów w kodzie pliku `index.ts` w katalogu `src`

```
function calculateTax(amount: number, format: boolean): string | number {
    const calcAmount = amount * 1.2;
    return format ? `${calcAmount.toFixed(2)} zł` : calcAmount;
}
```

```
let taxNumber: string | number = calculateTax(100, false);
let taxString: string | number = calculateTax(100, true);
```

```
console.log(`Wartość typu number: ${taxNumber.toFixed(2)}`);
console.log(`Wartość typu string: ${taxString.charAt(0)}`);
```

Można stosować jedynie właściwości i metody zdefiniowane przez wszystkie typy tworzące unię, co będzie użyteczne w przypadku skomplikowanych typów (jak to przedstawiłem w rozdziale 10.), choć jednocześnie jest to ograniczone przez małą liczbę wspólnych API oferowanych przez wartości podstawowe. Jedyna metoda współdzielona przez typy `number` i `string` użyta w unii na listingu 7.22 to `toString()`, jak widać na rysunku 7.3.



Rysunek 7.3. Efekt użycia unii typów

Dlatego też inne metody definiowane przez typy `number` i `string` nie mogą być używane — wywołanie metod `toFixed()` i `charAt()` w kodzie na listingu 7.22 powoduje wygenerowanie przez kompilator następujących komunikatów błędów:

```
src/index.ts(9,40): error TS2339: Property 'toFixed' does not exist on type 'string | number'. Property 'toFixed' does not exist on type 'string'.
src/index.ts(10,40): error TS2339: Property 'charAt' does not exist on type 'string | number'. Property 'charAt' does not exist on type 'number'.
```

Używanie asercji typu

Asercja typu informuje kompilator TypeScriptu, jak ma traktować wartość danego typu, co jest znane jako *zawężanie typu*. Asercja typu to jeden ze sposobów na zawężenie typu z unii, jak pokazałem na listingu 7.23.

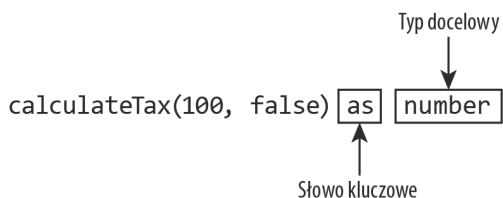
Listing 7.23. Używanie asercji typów w kodzie pliku `index.ts` w katalogu `src`

```
function calculateTax(amount: number, format: boolean): string | number {
    const calcAmount = amount * 1.2;
    return format ? `${calcAmount.toFixed(2)} zł` : calcAmount;
}

let taxNumber = calculateTax(100, false) as number;
let taxString = calculateTax(100, true) as string;

console.log(`Wartość typu number: ${taxNumber.toFixed(2)}`);
console.log(`Wartość typu string: ${taxString.charAt(7)}${taxString.charAt(8)}`);
```

Asercja typu jest definiowana za pomocą słowa kluczowego `as`, po którym znajduje się wymagany typ, jak pokazałem na rysunku 7.4.



Rysunek 7.4. Asercja typu

Na listingu 7.23 słowo kluczowe `as` zostało użyte w celu wskazania kompilatorowi, że wartość przypisana zmiennej `taxNumber` jest typu `number`, a wartość przypisana zmiennej `taxString` jest typu `string`:

```
...
let taxNumber = calculateTax(100, false) as number;
let taxString = calculateTax(100, true) as string;
...
```

■ **Ostrzeżenie** Asercja typu nie przeprowadza konwersji typu, a jedynie wskazuje kompilatorowi typ, który powinien być zastosowany dla wartości na potrzeby sprawdzania typu.

Podczas stosowania asercji typu w taki sposób TypeScript wykorzystuje podany typ jako typ zmiennej, co oznacza, że polecenia pogrubione na listingu 7.23 są odpowiednikami wcześniej przedstawionych:

```
let taxNumber: number = calculateTax(100, false) as number;
let taxString: string = calculateTax(100, true) as string;
...
```

Asercja typu wybiera określony typ z unii, dlatego można stosować metody i właściwości zdefiniowane w tym typie. W efekcie nie zostaną już wygenerowane błędy takie jak pokazane dla kodu na listingu 7.22, a wykonanie omawianego przykładu zakończy się wyświetleniem następujących danych wyjściowych:

```
Wartość typu number: 120.00
Wartość typu string: zł
```

Asercja typu nieoczekiwanego

Kompilator sprawdza, czy typ używany w asercji jest oczekiwany. Podczas używania asercji np. z unii typów asercja musi być jednym z typów unii. Aby sprawdzić, co się zdarzy w przypadku asercji typu nieoczekiwanego przez kompilator, zmodyfikuj zawartość pliku *index.ts* w sposób przedstawiony na listingu 7.24.

Listing 7.24. Asercja typu nieoczekiwanego w kodzie pliku *index.ts* w katalogu *src*

```
function calculateTax(amount: number, format: boolean): string | number {
    const calcAmount = amount * 1.2;
    return format ? `${calcAmount.toFixed(2)} zł` : calcAmount;
}
```

```
let taxNumber = calculateTax(100, false) as number;
let taxString = calculateTax(100, true) as string;
let taxBoolean = calculateTax(100, false) as boolean;
```

```
console.log(`Wartość typu number: ${taxNumber.toFixed(2)}`);
console.log(`Wartość typu string: ${taxString.charAt(0)}`);
console.log(`Wartość typu boolean: ${taxBoolean}`);
```

Asercja typu nakazuje kompilatorowi potraktowanie wartości `string | number` jako wartości typu `boolean`. Ponieważ kompilator ustalił, że `boolean` nie jest jednym z typów unii, po skompilowaniu kodu wygenerował następujący komunikat błędu:

```
...
src/index.ts(9,18): error TS2352: Conversion of type 'string | number' to type 'boolean'
may be a mistake because neither type sufficiently overlaps with the other. If this was
intentional, convert the expression to 'unknown' first.
  Type 'number' is not comparable to type 'boolean'.
...
```

W większości przypadków powinieneś przeglądać typy danych i asercje typu, a następnie usuwać problemy przez rozszerzenie unii typów lub zastosowanie asercji innego typu. Jednak istnieje możliwość wymuszenia asercji i zniwelowania ostrzeżenia generowanego przez kompilator przez zastosowanie asercji najpierw do typu `any`, a następnie dożądanego typu, jak pokazałem na listingu 7.25. (Błąd wyświetlony przez kompilator dotyczy typu `unknown`, do którego powrócę w dalszej części rozdziału).

Listing 7.25. Asercja do typu nieoczekiwanego w kodzie pliku `index.ts` w katalogu `src`

```
function calculateTax(amount: number, format: boolean): string | number {
    const calcAmount = amount * 1.2;
    return format ? `${calcAmount.toFixed(2)} zł` : calcAmount;
}

let taxNumber = calculateTax(100, false) as number;
let taxString = calculateTax(100, true) as string;
let taxBoolean = calculateTax(100, false) as any as boolean;

console.log(`Wartość typu number: ${taxNumber.toFixed(2)}`);
console.log(`Wartość typu string: ${taxString.charAt(7)}${taxString.charAt(8)}`);
console.log(`Wartość typu boolean: ${taxBoolean}`);
```

Ten dodatkowy krok uniemożliwia kompilatorowi wygenerowanie komunikatu ostrzeżenia, a wynik wywołania funkcji jest traktowany jako wartość typu `boolean`. Jednak jak już wcześniej wspomniałem, asercja wpływa jedynie na proces sprawdzania typu i nie przeprowadza koercji typu, co widać w wynikach wygenerowanych po skompilowaniu kodu:

```
Wartość typu number: 120.00
Wartość typu string: zł
Wartość typu boolean: 120
```

Wynik wygenerowany przez funkcję został opisany przez kompilator jako unia `string | number`, na której następnie zastosowano asercję do typu `boolean`. Natomiast po wykonaniu kodu funkcja generuje wartość typu `number` wyświetlaną w konsoli.

Alternatywna składnia asercji typu

Asercja typu może być również przeprowadzana za pomocą składni w postaci nawiasu ostrego. Dlatego też polecenie:

```
...
let taxString = calculateTax(100, true) as string;
...
    jest odpowiednikiem następującego polecenia:
...
let taxString = <string> calculateTax(100, true);
...
```

Problem związany z tą składnią polega na tym, że nie można jej używać w plikach typu TSX będących połączeniem elementów HTML-a i kodu TypeScriptu. Takie połączenie jest często spotykane podczas programowania z użyciem frameworka React, co dokładniej omówię w rozdziale 19. Dlatego też preferowany sposób na przeprowadzanie asercji typu opiera się na używaniu słowa kluczowego `as`.

Używanie wartownika typu

W przypadku wartości typów prostych słowo kluczowe `typeof` pozwala na sprawdzenie pod kątem określonego typu bez konieczności przeprowadzania asercji typu. Przykład takiego rozwiązania przedstawiłem na listingu 7.26.

Listing 7.26. Używanie wartownika typu w kodzie pliku `index.ts` w katalogu `src`

```
function calculateTax(amount: number, format: boolean): string | number {
    const calcAmount = amount * 1.2;
    return format ? `${calcAmount.toFixed(2)} zł` : calcAmount;
}

let taxValue = calculateTax(100, false);

if (typeof taxValue === "number") {
    console.log(`Wartość typu number: ${taxValue.toFixed(2)}`);
} else if (typeof taxValue === "string") {
    console.log(`Wartość typu string: ${taxValue.charAt(7)}${taxValue.charAt(8)}`);
}
```

Aby sprawdzić typ, słowo kluczowe `typeof` jest stosowane dla wartości i prowadzi do wygenerowania wartości typu `string`, którą następnie można porównywać z nazwami typów prostych w JavaScriptcie, takich jak `number` i `boolean`.

■ **Uwaga** Słowo kluczowe `typeof` może być używane jedynie z typami prostymi JavaScriptu. Do rozróżniania obiektów wymagane jest inne podejście, które zostało przedstawione w rozdziałach 3. i 10.

Kompilator nie implementuje słowa kluczowego `typeof`, które jest częścią specyfikacji JavaScriptu. Zamiast tego uznaje, że polecenia zdefiniowane w bloku konstrukcji warunkowej zostaną w trakcie działania aplikacji wykonane tylko wtedy, gdy sprawdzana wartość jest określonego typu. Takie ustalenia pozwalają kompilatorowi na traktowanie wartości jak sprawdzanego typu. Na przykład pierwsze sprawdzenie na listingu 7.26 dotyczy typu `number`:

```
...
if (typeof taxValue === "number") {
    console.log(`Wartość typu number: ${taxValue.toFixed(2)}`);
}
...
```

Kompilator TypeScriptu ustalił, że polecenia zdefiniowane w bloku kodu `if` będą wykonywane tylko wtedy, gdy wartość `taxValue` jest typu `number` i pozwala na używanie metody `toFixed()` wymienionego typu bez konieczności przeprowadzania asercji. Po skompilowaniu i uruchomieniu kodu zostaną wygenerowane następujące dane wyjściowe:

```
Wartość typu number: 120.00
```

Kompilator próbuje rozpoznać polecenia wartowników typu, nawet gdy nie znajdują się w konwencjonalnym bloku `if-else`. Kod przedstawiony na listingu 7.27 powoduje wygenerowanie

takich samych danych wyjściowych jak kod z listingu 7.26, ale w celu odróżniania typów korzysta z konstrukcji `switch`. W każdym bloku kodu kompilator traktuje `taxValue`, jakby została zdefiniowana tylko z typem wymienionym w poleceniu `case`.

Listing 7.27. Używanie wartowników typu w konstrukcji `switch` zdefiniowanej w kodzie pliku `index.ts` w katalogu `src`

```
function calculateTax(amount: number, format: boolean): string | number {
    const calcAmount = amount * 1.2;
    return format ? `${calcAmount.toFixed(2)} zł` : calcAmount;
}

let taxValue = calculateTax(100, false);

switch (typeof taxValue) {
    case "number":
        console.log(`Wartość typu number: ${taxValue.toFixed(2)}`);
        break;
    case "string":
        console.log(`Wartość typu string: ${taxValue.charAt(7)}${taxValue.charAt(8)}`);
        break;
}
```

Używanie typu `never`

TypeScript oferuje typ `never` stosowany w sytuacjach, gdy wartownik typu musi poradzić sobie z wszystkimi możliwymi typami dla wartości. Na przykład w kodzie na listingu 7.27 konstrukcja `switch` jest wartownikiem typów `number` i `string`, które są jedynymi zwracanymi z funkcji przez unię `string | number`. Po obsłużeniu wszystkich możliwych typów kompilator pozwoli na przypisanie wartości jedynie typowi `never`, jak pokazałem na listingu 7.28.

Listing 7.28. Używanie typu `never` w kodzie pliku `index.ts` w katalogu `src`

```
function calculateTax(amount: number, format: boolean): string | number {
    const calcAmount = amount * 1.2;
    return format ? `${calcAmount.toFixed(2)} zł` : calcAmount;
}

let taxValue = calculateTax(100, false);

switch (typeof taxValue) {
    case "number":
        console.log(`Wartość typu number: ${taxValue.toFixed(2)}`);
        break;
    case "string":
        console.log(`Wartość typu string: ${taxValue.charAt(7)}${taxValue.charAt(8)}`);
        break;
    default:
        let value: never = taxValue;
        console.log(`Nieoczekiwany typ dla wartości: ${value}`);
}
```

Czasami problem pojawia się po dotarciu programu do klauzuli default konstrukcji switch. TypeScript oferuje typ never gwarantujący, że w takiej sytuacji nie będzie można przypadkowo użyć wartości, gdy liczba typów możliwych do zastosowania została zawężona przez wartowniki typu.

Używanie typu unknown

Podczas omawiania typu any we wcześniejszej części rozdziału wyjaśniłem, że wartość any może być przypisywana innym typom, co prowadzi do powstania luki w procesie sprawdzania typów przez kompilator. TypeScript obsługuje również typ unknown uznawany za bezpieczniejszą alternatywę dla any. Wartość unknown może być przypisana tylko any lub samej sobie, o ile nie została użyta asercja typu lub wartownik typu. W kodzie przedstawionym na listingu 7.29 zostały powtórzone polecenia z wcześniejszego przykładu pokazującego sposób zachowania typu any, ale tym razem zdecydowałem się na użycie typu unknown.

Listing 7.29. Używanie typów any i unknown w kodzie pliku index.ts w katalogu src

```
function calculateTax(amount: number, format: boolean): string | number {
    const calcAmount = amount * 1.2;
    return format ? `${calcAmount.toFixed(2)} zł` : calcAmount;
}

let taxValue = calculateTax(100, false);
switch (typeof taxValue) {
    case "number":
        console.log(`Wartość typu number: ${taxValue.toFixed(2)}`);
        break;
    case "string":
        console.log(`Wartość typu string: ${taxValue.charAt(7)}${taxValue.charAt(8)}`);
        break;
    default:
        let value: never = taxValue;
        console.log(`Nieoczekiwany typ dla wartości: ${value}`);
}

let newResult: unknown = calculateTax(200, false);
let myNumber: number = newResult;
console.log(`Wartość liczbowa: ${myNumber.toFixed(2)}`);
```

Wartość unknown nie może być przypisywana innemu typowi bez asercji typu, więc po skompilowaniu i uruchomieniu kodu zostanie wygenerowany następujący komunikat błędu:

```
src/index.ts(18,5): error TS2322: Type 'unknown' is not assignable to type 'number'.
```

W kodzie przedstawionym na listingu 7.30 zastosowałem asercję typu w celu pozbycia się tego komunikatu i nakazania kompilatorowi przypisania wartości unknown jako typu number.

Listing 7.30. Asercja wartości typu unknown w kodzie pliku index.ts w katalogu src

```
function calculateTax(amount: number, format: boolean): string | number {
    const calcAmount = amount * 1.2;
```



```

    return format ? `${calcAmount.toFixed(2)} zł` : calcAmount;
}

let taxValue = calculateTax(100, false);

switch (typeof taxValue) {
  case "number":
    console.log(`Wartość typu number: ${taxValue.toFixed(2)}`);
    break;
  case "string":
    console.log(`Wartość typu string: ${taxValue.charAt(7)}${taxValue.charAt(8)}`);
    break;
  default:
    let value: never = taxValue;
    console.log(`Nieoczekiwany typ dla wartości: ${value}`);
}

let newResult: unknown = calculateTax(200, false);
let myNumber: number = newResult as number;
console.log(`Wartość liczbowa: ${myNumber.toFixed(2)}`);

```

W przeciwieństwie do wcześniejszego przykładu, wartość `unknown` faktycznie jest liczbą (typ `number`), więc kod nie generuje błędu w trakcie działania aplikacji i zostają wyświetlone następujące dane wyjściowe:

```

Wartość typu number: 120.00
Wartość liczbowa: 240.00

```

Używanie typów null

W systemie typów statycznych TypeScriptu istnieje pewna luka w postaci typów JavaScriptu `null` i `undefined`. Typ `null` może być przypisany tylko wartości `null` i używany do przedstawienia, że coś nie istnieje lub jest nieprawidłowe. Z kolei typ `undefined` może być przypisany tylko wartości `undefined` i używany, gdy zmienna wprawdzie została zdefiniowana, ale jeszcze nie ma przypisanej wartości.

Problem polega na tym, że domyślnie TypeScript traktuje wartości `null` i `undefined` jako poprawne dla wszystkich typów. Powodem tego jest wygoda, ponieważ spora ilość istniejącego kodu, który może być wymagany do użycia w aplikacji, korzysta z wymienionych wartości w trakcie swojego normalnego działania. To jednak prowadzi do niespójności podczas sprawdzania typu w TypeScriptie, jak pokazałem na listingu 7.31.

Listing 7.31. Używanie typów `null` w kodzie pliku `index.ts` w katalogu `src`

```

function calculateTax(amount: number, format: boolean): string | number {
  if (amount === 0) {
    return null;
  }
  const calcAmount = amount * 1.2;
  return format ? `${calcAmount.toFixed(2)} zł` : calcAmount;
}

```

```
let taxValue: string | number = calculateTax(0, false);

switch (typeof taxValue) {
  case "number":
    console.log(`Wartość typu number: ${taxValue.toFixed(2)}`);
    break;
  case "string":
    console.log(`Wartość typu string: ${taxValue.charAt(7)}${taxValue.charAt(8)}`);
    break;
  default:
    let value: never = taxValue;
    console.log(`Nieoczekiwany typ dla wartości: ${value}`);
}

let newResult: unknown = calculateTax(200, false);
let myNumber: number = newResult as number;
console.log(`Wartość liczbową: ${myNumber.toFixed(2)}`);
```

Zmiana wprowadzona w kodzie funkcji `calculateTax()` pokazuje przykład typowego użycia wartości `null` w sytuacji, gdy wartością parametru `amount` jest zero, które wskazuje na nieprawidłowy warunek. Typem wyniku działania funkcji i typem zmiennej `taxValue` jest `string | number`. Jednak w kodzie JavaScriptu zmiana wartości przypisanej zmiennej może spowodować również zmianę jej typu. Z taką sytuacją mamy do czynienia w omawianym przykładzie — drugie wywołanie `calculateTax()` zwraca `null`, co zmienia typ zmiennej `taxValue` na `null`. Gdy polecenia wartownika typu analizują typ zmiennej, ustalają, że nie należy on do unii `string | number`, więc efektem jest wygenerowanie następujących danych wyjściowych:

```
Nieoczekiwany typ dla wartości: null
Wartość liczbową: 240.00
```

W normalnych warunkach kompilator zgłosi błąd, gdy wartość jednego typu zostanie przypisana zmiennej innego typu. Jednak jak już wcześniej wspomniałem, kompilator pozwala na traktowanie `null` i `undefined` jako wartości dla wszystkich typów i dlatego nie generuje komunikatu błędu.

■ **Uwaga** Poza niespójnościami typów, wartości `null` mogą prowadzić do powstawania błędów w trakcie działania programu, które będą trudne do wychwycenia podczas pracy nad aplikacją i są często spotykane przez użytkowników. Na przykład w kodzie przedstawionym na listingu 7.31 nie ma łatwego sposobu na ustalenie przez użytkowników funkcji `calculateTax()`, czy i kiedy może ona zwrócić wartość `null`. Łatwo zobaczyć wartość `null` i ustalić powody jej użycia w przykładzie, ale staje się to znacznie trudniejsze w rzeczywistym projekcie lub pakiecie opracowanym przez podmiot zewnętrzny.

Ograniczenie przypisywania wartości `null`

Używanie wartości `null` i `undefined` może być ograniczone za pomocą opcji kompilatora o nazwie `strictNullChecks`, jak pokazałem na listingu 7.32. (To ustawienie ma również zastosowanie po wybraniu ustawienia `strict`).

Listing 7.32. Włączenie ścisłego sprawdzania pod kątem wartości null za pomocą opcji kompilatora w pliku `tsconfig.json` w katalogu `types`

```
{
  "compilerOptions": {
    "target": "es2020",
    "outDir": "./dist",
    "rootDir": "./src",
    "declaration": true,
    "noImplicitAny": true,
    "strictNullChecks": true
  }
}
```

Gdy ustawienie `strictNullChecks` ma przypisaną wartość `true`, kompilator nie może przypisywać wartości `null` lub `undefined` innym typom. Po zapisaniu zmiany w pliku konfiguracyjnym kompilator przeprowadzi ponowną kompilację pliku `index.ts` i wygeneruje następujący komunikat błędu:

```
src/index.ts(3,9): error TS2322: Type 'null' is not assignable to type 'string | number'.
```

Wprowadzona zmiana konfiguracyjna nakazuje kompilatorowi wygenerowanie błędu po przypisaniu wartości `null` lub `undefined` innemu typowi. W omawianym przykładzie pojawił się błąd, ponieważ wartość `null` zwrócona przez funkcję `calculateTax()` nie należy do żadnego z typów unii opisującej wartość zwrótną tej funkcji.

Aby usunąć ten błąd, należy zmodyfikować funkcję w taki sposób, by nie używała wartości `null`. Ewentualnie można rozszerzyć unię opisującą wartość zwrótną o wartość `null` i takie podejście zastosowałem w kodzie przedstawionym na listingu 7.33.

Listing 7.33. Rozszerzenie unii typów w kodzie pliku `index.ts` w katalogu `src`

```
function calculateTax(amount: number, format: boolean): string | number | null {
  if (amount === 0) {
    return null;
  }
  const calcAmount = amount * 1.2;
  return format ? `${calcAmount.toFixed(2)} zł` : calcAmount;
}

let taxValue: string | number | null = calculateTax(0, false);

switch (typeof taxValue) {
  case "number":
    console.log(`Wartość typu number: ${taxValue.toFixed(2)}`);
    break;
  case "string":
    console.log(`Wartość typu string: ${taxValue.charAt(7)}${taxValue.charAt(8)}`);
    break;
  default:
    if (taxValue === null) {
      console.log("Wartość to null");
    } else {
      console.log(typeof taxValue);
      let value: never = taxValue;
    }
  }
}
```

```
        console.log(`Nieoczekiwany typ dla wartości: ${value}`);
    }
}
```

Rozszerzona unia typów wyraźnie wskazuje, że funkcja może zwrócić wartość `null`. Dlatego też kod korzystający z tej funkcji będzie przygotowany na obsługę wartości `string`, `number` lub `null`. Jak już wspomniałem w rozdziale 3., wynikiem użycia operatora `typeof` z wartością `null` będzie `object`, więc ochrona przed wartościami `null` odbywa się za pomocą wyraźnego sprawdzenia wartości, które kompilator TypeScriptu uznaje za wartownika typu. Po uruchomieniu kod przedstawiony na listingu 7.33 powoduje wygenerowanie następujących danych wyjściowych:

```
Wartość to null
```

Usunięcie `null` z unii za pomocą asercji

Czy pamiętasz, że unia przedstawia wynik złączenia API poszczególnych typów? Wartości `null` i `undefined` nie dostarczają żadnych właściwości i metod, co oznacza, że wartości unii typów `null` nie mogą być używane bezpośrednio, nawet jeśli typy inne niż `null` pozwalają na złączenie użytecznych właściwości i metod (przykłady zaprezentuję w dalszej części książki). Asercja inna niż `null` wskazuje kompilatorowi, że wartość jest inna niż `null`, co prowadzi do usunięcia `null` z unii typów i pozwala na wykorzystanie złączenia z innymi typami, jak pokazałem na listingu 7.34.

-
- **Ostrzeżenie** Asercja inna niż `null` powinna być używana tylko wtedy, gdy wiadomo, że wartość `null` na pewno się nie pojawi. W przypadku pojawienia się wartości `null` po zastosowaniu asercji innej niż `null` wygenerowany będzie błąd podczas działania programu. Bezpieczniejsze podejście polega na użyciu wartownika typu, jak to przedstawię w dalszej części rozdziału.
-

Listing 7.34. Używanie asercji innej niż `null` w kodzie pliku `index.ts` w katalogu `src`

```
function calculateTax(amount: number, format: boolean): string | number | null {
    if (amount === 0) {
        return null;
    }
    const calcAmount = amount * 1.2;
    return format ? `${calcAmount.toFixed(2)} zł` : calcAmount;
}

let taxValue: string | number = calculateTax(100, false);

switch (typeof taxValue) {
    case "number":
        console.log(`Wartość typu number: ${taxValue.toFixed(2)}`);
        break;
    case "string":
        console.log(`Wartość typu string: ${taxValue.charAt(7)}${taxValue.charAt(8)}`);
        break;
    default:
        if (taxValue === null) {
```

```

        console.log("Wartość to null");
    } else {
        console.log(typeof taxValue);
        let value: never = taxValue;
        console.log(`Nieoczekiwany typ dla wartości: ${value}`);
    }
}

```

Wartość inna niż `null` jest poddawana asercji przez umieszczenie znaku `!` po wartości, jak pokazałem na rysunku 7.5. Asercja użyta na listingu wskazuje kompilatorowi, że wartość zwrótna funkcji `calculateTax()` będzie inna niż `null`. Pozwala to na przypisanie jej zmiennej `taxValue`, której typem jest `string | number`.

Asercja inna niż null
↓
`calculateTax(100, false) !;`

Rysunek 7.5. Asercja wartości innej niż `null`

Po skompilowaniu i uruchomieniu kod przedstawiony na listingu 7.34 spowoduje wygenerowanie następujących danych wyjściowych:

Wartość typu `number`: 120.00

Usuwanie wartości `null` z unii za pomocą wartownika typu

Alternatywne podejście polega na odfiltrowaniu wartości `null` i `undefined` za pomocą wartownika typu, jak pokazałem na listingu 7.35. Zaletą takiego podejścia jest sprawdzanie wartości w trakcie działania programu.

Listing 7.35. Usuwanie wartości `null` z unii za pomocą wartownika typu w kodzie pliku `index.ts` w katalogu `src`

```

function calculateTax(amount: number, format: boolean): string | number | null {
    if (amount === 0) {
        return null;
    }
    const calcAmount = amount * 1.2;
    return format ? `${calcAmount.toFixed(2)} zł` : calcAmount;
}

let taxValue: string | number | null = calculateTax(100, false);
if (taxValue !== null) {
    let nonNullTaxValue: string | number = taxValue;
    switch (typeof taxValue) {
        case "number":
            console.log(`Wartość typu number: ${taxValue.toFixed(2)}`);
            break;
        case "string":
            console.log(`Wartość typu string: ${taxValue.charAt(7)}${taxValue.charAt(8)}`);

```

```
        break;
    }
} else {
    console.log("To nie jest wartość typu string lub number");
}
```

Kompilator wie, że sprawdzenie wartości pod kątem `null` oznacza możliwość potraktowania jej jako nieprzyjmującej wartości `null` unii typu `string | number` w bloku kodu. (Kompilator ponadto wie, że zmienna `taxValue` może mieć wartość `null` jedynie w bloku `else`). Po skompilowaniu i uruchomieniu kod przedstawiony na listingu 7.35 spowoduje wygenerowanie następujących danych wyjściowych:

```
Wartość typu number: 120.00
```

Używanie asercji ostatecznego przypisania

Jeżeli została włączona opcja `strictNullChecks` kompilatora, wówczas kompilator zgłosi błąd, jeśli zmienna zostanie użyta przed przypisaniem jej wartości. To przydatna funkcjonalność, ale mogą się zdarzać sytuacje, w których wartość przypisana w taki sposób nie będzie widoczna dla kompilatora. Spójrz na przykładowy program przedstawiony na listingu 7.36.

■ **Ostrzeżenie** W kodzie na listingu 7.36 wbudowaną w JavaScriptcie funkcję `eval()` wykorzystałem do wykonania ciągu tekstowego jako polecenia kodu. Ta funkcja jest uznawana za niebezpieczną i nie powinna być używana w rzeczywistych projektach.

Listing 7.36. Użycie w kodzie pliku `index.ts` w katalogu `src` zmiennej, której nie przypisano wartości

```
function calculateTax(amount: number, format: boolean): string | number | null {
    if (amount === 0) {
        return null;
    }
    const calcAmount = amount * 1.2;
    return format ? `${calcAmount.toFixed(2)} zł` : calcAmount;
}
```

```
let taxValue: string | number | null;
eval("taxValue = calculateTax(100, false);");
```

```
if (taxValue !== null) {
    let nonNullTaxValue: string | number = taxValue;
    switch (typeof taxValue) {
        case "number":
            console.log(`Wartość typu number: ${taxValue.toFixed(2)}`);
            break;
        case "string":
            console.log(`Wartość typu string:
${taxValue.charAt(7)}${taxValue.charAt(8)}`);
            break;
    }
}
```

```

} else {
    console.log("To nie jest wartość typu string lub number");
}

```

Funkcja `eval()` akceptuje argument typu `string` i wykonuje go jako polecenie kodu. Kompilator JavaScriptu nie jest w stanie ustalić efektu działania funkcji `eval()` i nie wie, że przypisuje wartość zmiennej `taxValue`. Po skompilowaniu tego kodu zostaną wygenerowane następujące błędy:

```

src/index.ts(12,5): error TS2454: Variable 'taxValue' is used before being assigned.
src/index.ts(13,9): error TS2322: Type 'string | number | null' is not assignable to type 'string | number'.
    Type 'null' is not assignable to type 'string | number'.
src/index.ts(13,44): error TS2454: Variable 'taxValue' is used before being assigned.
src/index.ts(14,20): error TS2454: Variable 'taxValue' is used before being assigned.

```

Asercja ostatecznego przypisania informuje TypeScript, że wartość zostanie przypisana przed użyciem zmiennej, jak pokazałem na listingu 7.37.

Listing 7.37. Używanie asercji ostatecznego przypisania w kodzie pliku `index.ts` w katalogu `src`

```

function calculateTax(amount: number, format: boolean): string | number | null {
    if (amount === 0) {
        return null;
    }
    const calcAmount = amount * 1.2;
    return format ? `${calcAmount.toFixed(2)} zł` : calcAmount;
}

```

```

let taxValue!: string | number | null;
eval("taxValue = calculateTax(100, false)");

if (taxValue !== null) {
    let nonNullTaxValue: string | number = taxValue;
    switch (typeof taxValue) {
        case "number":
            console.log(`Wartość typu number: ${taxValue.toFixed(2)}`);
            break;
        case "string":
            console.log(`Wartość typu string:
${taxValue.charAt(7)}${taxValue.charAt(8)}`);
            break;
    }
} else {
    console.log("To nie jest wartość typu string lub number");
}

```

Asercją ostatecznego przypisania jest znak `!`, przy czym jest on stosowany po nazwie, gdy zmienna jest definiowana — to przeciwieństwo asercji innych niż `null`, w których znak `!` pojawia się w wyrażeniach. Podobnie jak w przypadku innych asercji, stajesz się odpowiedzialny za zagwarantowanie, że wartość faktycznie zostanie przypisana. Jeżeli użyjesz asercji, ale nie przeprowadzisz operacji przypisania, skutkiem może być błąd w trakcie działania programu.

Asercja przedstawiona na listingu 7.37 pozwala na skompilowanie kodu, który po uruchomieniu powoduje wygenerowanie następujących danych wyjściowych:

Wartość typu number: 120.00

Podsumowanie

W tym rozdziale wyjaśniłem, jak używać TypeScriptu do ograniczania systemu typów JavaScriptu przez przeprowadzanie operacji sprawdzania typu. Dowiedziałeś się, jak wykorzystać adnotacje typu do określania używanych typów oraz jak kompilator może określić typ na podstawie poleceń kodu. Omówiłem również typy `any`, `unknown` i `never`, a także unie typów, wartowników typu i ograniczanie zakresu typów. W następnym rozdziale dowiesz się nieco więcej na temat tego, jak w języku TypeScript przebiega praca z funkcjami.

ROZDZIAŁ 8.



Używanie funkcji

W tym rozdziale wyjaśnię, jak TypeScript jest używany przez funkcje. Pokażę, jak TypeScript pomaga w uniknięciu najczęściej pojawiających się problemów podczas definiowania funkcji. Poza tym dowiesz się, jak przebiega praca z parametrami i jak są generowane wyniki działania funkcji. Streszczenie materiału przedstawionego w rozdziale znajdziesz w tabeli 8.1.

Tabela 8.1. Streszczenie materiału przedstawionego w rozdziale

Problem	Rozwiązanie	Listing
Umożliwienie wywołania funkcji z mniejszą liczbą argumentów niż parametrów	Zdefiniuj parametry opcjonalne lub parametry o wartości domyślnej	7 i 8
Umożliwienie wywołania funkcji z większą liczbą argumentów niż parametrów	Użyj parametru resztowego	9 i 10
Ograniczenie typów, które mogą być użyte dla wartości parametrów i wyników	Zastosuj adnotację typu dla parametrów lub sygnatury funkcji	11, 17 i 18
Uniemożliwienie używania wartości <code>null</code> jako argumentów funkcji	Włącz opcję kompilatora <code>strictNullChecks</code>	Od 12 do 14
Zagwarantowanie, że wszystkie ścieżki wykonywania kodu funkcji będą zwracały wynik	Włącz opcję kompilatora <code>noImplicitReturns</code>	15 i 16
Opisanie relacji pomiędzy typami parametrów funkcji i jej wyników	Przeciążenie typów funkcji	19 i 20
Opisanie efektu użycia funkcji <code>assert()</code>	Użyj słowa kluczowego <code>asserts</code>	Od 21 do 23

W tabeli 8.2 wymieniałem opcje kompilatora TypeScriptu użyte w rozdziale.

Tabela 8.2. Opcje kompilatora TypeScriptu użyte w rozdziale

Opcja	Opis
target	Ta opcja określa docelową wersję języka JavaScript, dla której kompilator wygeneruje kod
outDir	Ta opcja określa katalog, w którym zostaną umieszczone wygenerowane pliki JavaScriptu
rootDir	Ta opcja określa katalog główny używany przez kompilator do wyszukiwania plików TypeScriptu
declaration	Ta opcja generuje pliki deklaracji typu, które pomagają w zrozumieniu, w jaki sposób zostały ustalone typy dla kodu JavaScriptu. Wspomniane pliki zostaną szczegółowo omówione w rozdziale 14.
strictNullChecks	Ta opcja uniemożliwia używanie <code>null</code> i <code>undefined</code> jako wartości dla innych typów
noImplicitReturns	Ta opcja powoduje, że wszystkie ścieżki wykonywania w funkcji muszą się kończyć zwróceniem wyniku
noUnusedParameters	Ta opcja powoduje wygenerowanie przez kompilator ostrzeżenia, jeśli funkcja definiuje nieużywane parametry

Przygotowanie projektu

W rozdziale będę kontynuował używanie projektu *types* utworzonego w rozdziale 7. Aby przygotować się do tego rozdziału, należy zastąpić zawartość pliku *index.ts* w katalogu *src* kodem przedstawionym na listingu 8.1.

-
- **Wskazówka** Przykładowy projekt dla tego rozdziału, a także projekty dla pozostałych rozdziałów książki zostały umieszczone w archiwum dostępnym pod adresem <https://ftp.helion.pl/przyklady/tys4o2.zip>.
-

Listing 8.1. Nowa zawartość pliku *index.ts* w katalogu *src*

```
function calculateTax(amount) {
    return amount * 1.2;
}

let taxValue = calculateTax(100);
console.log(`Wartość całkowita: ${taxValue}`);
```

Umieść znaki komentarza na początku opcji kompilatora uniemożliwiających niejawne używanie typu `any` oraz przypisywanie wartości `null` i `undefined` innym typom, jak pokazałem na listingu 8.2.

Listing 8.2. Wylączenie opcji kompilatora w pliku *tsconfig.json* w katalogu *types*

```
{
  "compilerOptions": {
    "target": "es2020",
```

```

    "outDir": "./dist",
    "rootDir": "./src",
    "declaration": true,
    // "noImplicitAny": true,
    // "strictNullChecks": true
  }
}

```

Otwórz nowe okno powłoki, przejdź do katalogu *types*, a następnie z jego poziomu wydaj polecenie przedstawione na listingu 8.3, uruchamiające kompilator TypeScriptu w trybie automatycznego kompilowania kodu po wykryciu zmiany w dowolnym pliku i wykonywania kodu po jego skompilowaniu.

Listing 8.3. Uruchamianie kompilatora TypeScriptu

```
$ npm start
```

Kompilator przeprowadzi kompilację kodu w pliku *index.ts*, wykona wygenerowany kod JavaScriptu, a następnie przejdzie do trybu monitorowania i wyświetli następujące dane wyjściowe:

```
6:52:41 AM - Starting compilation in watch mode...

6:52:43 AM - Found 0 errors. Watching for file changes.
Wartość całkowita: 120
```

Definiowanie funkcji

TypeScript konwertuje funkcje JavaScriptu, aby stały się bardziej przewidywalne, i umożliwia wyraźne definiowanie założeń dotyczących typu danych, co pozwala na ich sprawdzanie przez kompilator.

W pliku *index.ts* została zdefiniowana prosta funkcja:

```

...
function calculateTax(amount) {
    return amount * 1.2;
}
...

```

W rozdziale 7. pokazałem, jak funkcjonalność TypeScriptu taka jak adnotacje typów może być stosowana względem funkcji. W tym podrozdziale powrócę do wspomnianej funkcjonalności i zaprezentuję jeszcze inne sposoby, w jakie TypeScript usprawnia pracę z funkcjami.

Ponowne definiowanie funkcji

Jedną z najważniejszych zmian wprowadzanych przez TypeScript jest generowanie komunikatu ostrzeżenia w przypadku ponownego definiowania danej funkcji. W języku JavaScript funkcja może być zdefiniowana więcej niż tylko raz, a po jej wywołaniu zostanie użyta najnowsza implementacja. Prowadzi to do wielu problemów dla programistów, którzy mają doświadczenie

w programowaniu za pomocą innych języków. Przykład problemów, jakie można napotkać po ponownym zdefiniowaniu funkcji, będziesz mógł zobaczyć po wprowadzeniu w kodzie pliku *index.ts* zmian przedstawionych na listingu 8.4.

Listing 8.4. Ponowne zdefiniowanie funkcji w kodzie pliku *index.ts* w katalogu *src*

```
function calculateTax(amount) {
    return amount * 1.2;
}

function calculateTax(amount, discount) {
    return calculateTax(amount) - discount;
}

let taxValue = calculateTax(100);
console.log(`Wartość całkowita: ${taxValue}`);
```

W wielu językach programowania obsługiwane jest tzw. przeciążanie funkcji, które pozwala na zdefiniowanie wielu funkcji o takiej samej nazwie, jeśli mają one różną liczbę parametrów lub jeśli parametry mają odmienne typy. Jeżeli przywykłeś do takiego stylu programowania, wówczas kod przedstawiony na listingu 8.4 wygląda na zupełnie prawidłowy. Można przyjąć założenie, że druga funkcja `calculateTax()` korzysta z pierwszej podczas naliczania rabatu dla ceny produktu.

JavaScript nie obsługuje przeciążania funkcji i jeśli zdefiniujesz dwie funkcje o takich samych nazwach, wówczas druga zastąpi pierwszą niezależnie od parametrów funkcji. Liczba argumentów użytych do wywołania funkcji nie ma znaczenia w JavaScriptcie — jeśli istnieje więcej parametrów niż argumentów, wówczas dodatkowe parametry mają wartość `undefined`. Natomiast jeśli istnieje więcej argumentów niż parametrów, wówczas funkcja może je zignorować lub wykorzystać wartość specjalną `arguments` zapewniającą dostęp do wszystkich argumentów użytych do wywołania funkcji. W przypadku wykonania kodu przedstawionego na listingu 8.4 pierwsze wywołanie `calculateTax()` zostałoby zignorowane, a wywołane byłoby drugie, ale bez wartości dla drugiego parametru. Podczas wykonywania funkcji nieustannie wywoływałaby ona samą siebie aż do przepełnienia stosu i wygenerowania komunikatu błędu.

Aby uniknąć takiego problemu, kompilator TypeScriptu zgłasza błąd w przypadku zdefiniowania wielu funkcji o takiej samej nazwie. Spójrz na komunikaty błędu wygenerowane przez kompilator podczas próby wykonania kodu zamieszczonego na listingu 8.4:

```
src/index.ts(1,10): error TS2393: Duplicate function implementation.
src/index.ts(5,10): error TS2393: Duplicate function implementation.
```

Praktyczny skutek braku możliwości przeciążania funkcji jest taki, że trzeba używać różnych nazw funkcji, np. `calculateTax()` i `calculateTaxWithDiscount()`, bądź też jednej funkcji zmieniającej zachowanie na podstawie parametrów. Pierwsze podejście sprawdza się doskonale dla skompilowanych grup funkcjonalności, natomiast drugie preferuję w przypadku prostszych zadań. Na listingu 8.5 przedstawiłem drugie podejście i skonsolidowałem funkcjonalność w pojedynczą funkcję.

Listing 8.5. Konsolidacja funkcji w kodzie pliku *index.ts* w katalogu *src*

```
function calculateTax(amount, discount) {
    return (amount * 1.2) - discount;
}

let taxValue = calculateTax(100, 0);
console.log(`Wartość całkowita: ${taxValue}`);
```

Po skompilowaniu i uruchomieniu kodu przedstawionego na listingu 8.6 zostaną wygenerowane następujące dane wyjściowe:

```
Wartość całkowita: 120
```

Parametry funkcji

Aby zapewnić możliwość skompilowania kodu, na listingu 8.5 wprowadziłem dwie zmiany. Pierwsza polega na usunięciu duplikatu funkcji `calculateTax()` i umieszczeniu całej funkcjonalności w pojedynczej funkcji. Druga zmiana wiąże się z poleceniem wywołującym funkcję, do którego dodałem drugi argument.

```
...
let taxValue = calculateTax(100, 0);
...
```

Podczas wywoływania funkcji TypeScript stosuje ściślejsze podejście niż JavaScript i oczekuje używania funkcji z taką samą liczbą argumentów, odpowiadającą liczbie parametrów. Do pliku *index.ts* dodaj polecenia przedstawione na listingu 8.6, aby poznać reakcję kompilatora na użycie innej liczby argumentów niż parametrów funkcji.

Listing 8.6. Wywoływanie funkcji w kodzie pliku *index.ts* w katalogu *src*

```
function calculateTax(amount, discount) {
    return (amount * 1.2) - discount;
}

let taxValue = calculateTax(100, 0);
console.log(`Dwa argumenty: ${taxValue}`);
taxValue = calculateTax(100);
console.log(`Jeden argument: ${taxValue}`);
taxValue = calculateTax(100, 10, 20);
console.log(`Trzy argumenty: ${taxValue}`);
```

Pierwsze nowe wywołanie funkcji nie dostarcza wystarczającej liczby argumentów, natomiast drugie zawiera ich zbyt wiele. Podczas próby kompilacji i uruchomienia kodu kompilator wygeneruje następujące komunikaty błędów:

```
src/index.ts(7,12): error TS2554: Expected 2 arguments, but got 1.
src/index.ts(8,12): error TS2554: Expected 2 arguments, but got 3.
```

Kompilator nalega na dopasowanie liczby argumentów i parametrów, aby oczekiwania były wyraźnie zdefiniowane w kodzie, podobnie jak w przypadku funkcjonalności przedstawionych w rozdziale 7. Analiza zbioru parametrów nie pozwala łatwo ustalić sposobu zachowania funkcji w razie niedostarczenia jej pewnych wartości. Z kolei gdy funkcja zostanie wywołana z inną liczbą argumentów, trudno będzie ustalić, czy to jest celowe, czy to jednak błąd. TypeScript radzi sobie z tymi problemami, wymagając, aby argumenty odpowiadały liczbie parametrów, o ile funkcja nie wskazuje, że może być znacznie elastyczniejsza dzięki możliwościom omówionym w dalszej części rozdziału.

-
- **Wskazówka** Jeżeli opcja `noUnusedParameters` została włączona, kompilator wygeneruje ostrzeżenie o zdefiniowaniu przez funkcję nieużywanych parametrów.
-

Używanie parametrów opcjonalnych

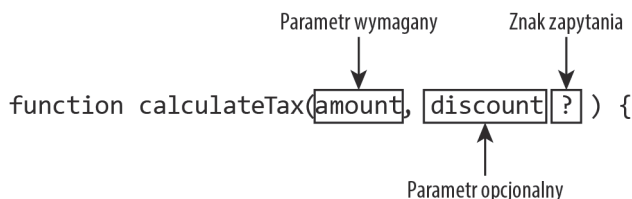
Domyślnie parametry funkcji są obowiązkowe, ale może to ulec zmianie przez użycie parametrów opcjonalnych, jak pokazałem na listingu 8.7. (Umieściłem znaki komentarza na początku poleceń zawierających zbyt wiele argumentów; powrócę do nich w dalszej części rozdziału).

Listing 8.7. Definiowanie parametru opcjonalnego w kodzie pliku `index.ts` w katalogu `src`

```
function calculateTax(amount, discount?) {
    return (amount * 1.2) - (discount || 0);
}
```

```
let taxValue = calculateTax(100, 0);
console.log(`Dwa argumenty: ${taxValue}`);
taxValue = calculateTax(100);
console.log(`Jeden argument: ${taxValue}`);
//taxValue = calculateTax(100, 10, 20);
//console.log(`Trzy argumenty: ${taxValue}`);
```

Parametry opcjonalne są definiowane przez umieszczenie znaku zapytania po nazwie parametru, jak pokazałem na rysunku 8.1.



Rysunek 8.1. Definiowanie parametru opcjonalnego

-
- **Uwaga** Parametry opcjonalne muszą być zdefiniowane po parametrach wymaganych. Dlatego nie można zamienić kolejnością np. parametrów `amount` i `discount` na listingu 8.7, ponieważ `amount` jest wymagany, a `discount` jest opcjonalny.
-

Komponent wywołujący funkcję `calculateTax()` może pominąć wartość parametru `discount`, który dostarczy funkcji wartość `undefined` dla parametru. Funkcja deklarująca parametry opcjonalne musi gwarantować możliwość działania w razie niedostarczenia wartości dla tych parametrów. W przypadku funkcji przedstawionej na listingu 8.7 odbywa się to za pomocą operatora logicznego lub (`||`) pozwalającego, aby niezdefiniowana wartość była uznana za zero w sytuacji, gdy parametr `discount` pozostaje niezdefiniowany:

```
...
return (amount * 1.2) - (discount || 0);
...
```

Parametr `discount` jest używany w dokładnie taki sam sposób jak parametr wymagany, a jedyna różnica polega na tym, że funkcja musi mieć możliwość obsługi sytuacji, w której otrzyma wartość `undefined`.

Użytkownik funkcji nie musi podejmować żadnych szczególnych kroków, aby pracować z parametrem opcjonalnym. W omawianym przykładzie oznacza to możliwość użycia funkcji z jednym lub dwoma argumentami. Po uruchomieniu kod z listingu 8.7 powoduje wygenerowanie następujących danych wyjściowych:

```
Dwa argumenty: 120
Jeden argument: 120
```

Używanie parametru z wartością domyślną

Jeżeli istnieje wartość awaryjna, która powinna być użyta dla parametru opcjonalnego, wówczas może być zastosowana podczas definiowania parametru, jak pokazałem na listingu 8.8.

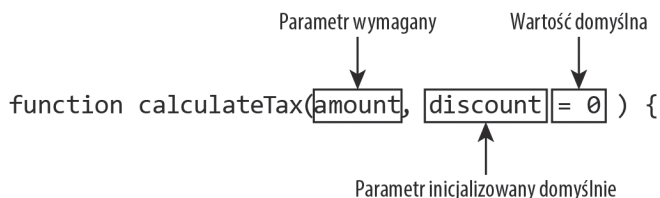
Listing 8.8. Użycie wartości domyślnej parametru w kodzie pliku `index.ts` w katalogu `src`

```
function calculateTax(amount, discount = 0) {
  return (amount * 1.2) - discount;
}
```

```
let taxValue = calculateTax(100, 0);
console.log(`Dwa argumenty: ${taxValue}`);
taxValue = calculateTax(100);
console.log(`Jeden argument: ${taxValue}`);
//taxValue = calculateTax(100, 10, 20);
//console.log(`Trzy argumenty: ${taxValue}`);
```

Parametr z wartością domyślną jest nazywany *parametrem inicjalizowanym domyślnie*. Po nazwie parametru znajduje się operator przypisania (jeden znak równości) i wartość, jak pokazałem na rysunku 8.2. Zwróć uwagę na brak znaku zapytania podczas definiowania parametru z wartością domyślną.

Używanie wartości domyślnej oznacza, że kod funkcji nie musi przeprowadzać sprawdzenia pod kątem wartości `undefined`, a wartość awaryjna może być zmieniona w jednym miejscu, co zostanie odzwierciedlone w całej funkcji.



Rysunek 8.2. Definiowanie wartości parametru domyślnego

-
- **Wskazówka** Parametry z wartościami domyślnymi nadal są parametrami opcjonalnymi, nawet pomimo braku użytego znaku zapytania, i muszą być definiowane po wymaganych parametrach funkcji.
-

Po skompilowaniu i uruchomieniu kodu przedstawionego na listingu 8.8 zostaną wygenerowane następujące dane wyjściowe:

```
Dwa argumenty: 120
Jeden argument: 120
```

Używanie parametru resztowego

Odpowiednikiem parametru opcjonalnego jest tzw. *parametr resztowy*, który pozwala funkcji na przyjęcie zmiennej liczby argumentów grupowanych ze sobą i przedstawianych razem. Funkcja może mieć tylko jeden parametr resztowy, który musi być jej ostatnim parametrem, jak pokazałem na listingu 8.9.

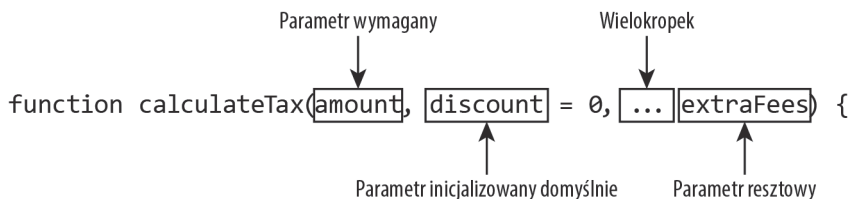
Listing 8.9. Definiowanie parametru resztowego w kodzie pliku `index.ts` w katalogu `src`

```
function calculateTax(amount, discount = 0, ...extraFees) {
    return (amount * 1.2) - discount
        + extraFees.reduce((total, val) => total + val, 0);
}
```

```
let taxValue = calculateTax(100, 0);
console.log(`Dwa argumenty: ${taxValue}`);
taxValue = calculateTax(100);
console.log(`Jeden argument: ${taxValue}`);
taxValue = calculateTax(100, 10, 20);
console.log(`Trzy argumenty: ${taxValue}`);
```

Parametr resztowy jest definiowany przez poprzedzenie jego nazwy wielokropkiem (trzy kropki), jak pokazałem na rysunku 8.3.

Każdy argument, dla którego nie ma odpowiadającego mu parametru, będzie przypisany parametrowi resztowemu zdefiniowanemu w postaci tablicy. Ta tablica zawsze będzie zainicjalizowana i pozostaje pusta, jeśli nie ma żadnych argumentów dodatkowych. Zdefiniowanie parametru resztowego oznacza, że funkcja `calculateTax()` może być wywołana z jednym lub większą liczbą argumentów: pierwszy jest przypisywany parametrowi `amount`, drugi (o ile istnieje) będzie przypisany parametrowi `discount`, natomiast każdy kolejny będzie dodany do tablicy parametrów `extraFees`.



Rysunek 8.3. Definiowanie parametru resztowego

Proces grupowania argumentów w tablicy parametrów resztowych odbywa się automatycznie i nie ma żadnych szczególnych wymagań podczas wywoływania funkcji. Użytkownik funkcji może zdefiniować argumenty dodatkowe i rozdzielić je przecinkami, jak pokazałem na listingu 8.10.

Listing 8.10. Używanie dodatkowych argumentów funkcji w kodzie pliku `index.ts` w katalogu `src`

```
function calculateTax(amount, discount = 0, ...extraFees) {
  return (amount * 1.2) - discount
    + extraFees.reduce((total, val) => total + val, 0);
}

let taxValue = calculateTax(100, 0);
console.log(`Dwa argumenty: ${taxValue}`);
taxValue = calculateTax(100);
console.log(`Jeden argument: ${taxValue}`);
taxValue = calculateTax(100, 10, 20);
console.log(`Trzy argumenty: ${taxValue}`);
taxValue = calculateTax(100, 10, 20, 1, 30, 7);
console.log(`Sześć argumentów: ${taxValue}`);
```

Po skompilowaniu i uruchomieniu kodu przedstawionego na listingu 8.10 zostaną wygenerowane następujące dane wyjściowe:

```
Dwa argumenty: 120
Jeden argument: 120
Trzy argumenty: 130
Sześć argumentów: 168
```

Stosowanie adnotacji typu dla parametrów funkcji

Domyślnie kompilator TypeScriptu przypisuje wszystkim parametrom funkcji typ `any`, ale konkretne typy mogą zostać zadeklarowane za pomocą adnotacji typów. W kodzie na listingu 8.11 przedstawiłem zastosowanie adnotacji typów dla funkcji `calculateTax()`, aby mieć gwarancję, że tylko wartości typu `number` będą używane dla jej parametrów.

Listing 8.11. Stosowanie adnotacji typu parametru w kodzie pliku `index.ts` w katalogu `src`

```
function calculateTax(amount: number, discount: number = 0, ...extraFees: number[]) {
  return (amount * 1.2) - discount
    + extraFees.reduce((total, val) => total + val, 0);
}
```

```
let taxValue = calculateTax(100, 0);
console.log(`Dwa argumenty: ${taxValue}`);
taxValue = calculateTax(100);
console.log(`Jeden argument: ${taxValue}`);
taxValue = calculateTax(100, 10, 20);
console.log(`Trzy argumenty: ${taxValue}`);
taxValue = calculateTax(100, 10, 20, 1, 30, 7);
console.log(`Sześć argumentów: ${taxValue}`);
```

W przypadku parametrów z wartościami domyślnymi adnotacja typu pojawia się przed przypisaniem wartości. Typem parametru resztowego zawsze jest tablica. Do tematu typu tablicy powrócę w rozdziale 9. W omawianym przykładzie adnotacja parametru `extraFees` informuje kompilator, że wszystkie argumenty dodatkowe muszą być liczbami. Kod przedstawiony na listingu 8.11 powoduje wygenerowanie następujących danych wyjściowych:

```
Dwa argumenty: 120
Jeden argument: 120
Trzy argumenty: 130
Sześć argumentów: 168
```

■ **Wskazówka** Adnotacje typów dla parametrów opcjonalnych są stosowane po znaku zapytania, np. `discount?: number`.

Kontrolowanie wartości null parametru

Jak wyjaśniłem w rozdziale 7., TypeScript domyślnie pozwala na używanie `null` i `undefined` jako wartości dla wszystkich typów. Oznacza to, że funkcja może otrzymywać wartości `null` dla wszystkich parametrów, jak pokazałem na listingu 8.12.

Listing 8.12. Przekazywanie wartości `null` do funkcji w kodzie pliku `index.ts` w katalogu `src`

```
function calculateTax(amount: number, discount: number = 0, ...extraFees: number[]) {
    return (amount * 1.2) - discount
        + extraFees.reduce((total, val) => total + val, 0);
}

let taxValue = calculateTax(null, 0);
console.log(`Wartość wraz z podatkiem: ${taxValue}`);
```

Jeżeli wartość `null` zostanie przekazana dla parametru inicjalizowanego domyślnie, wówczas będzie użyta jego wartość domyślna, jakby funkcja została wywołana bez argumentu. Natomiast w przypadku parametrów wymaganych funkcja otrzymuje wartość `null`, co może prowadzić do nieoczekiwanych wyników. W omawianym przykładzie funkcja `calculateTax()` otrzymuje wartość `null` dla parametru `amount`, co powoduje wygenerowanie następujących danych wyjściowych:

```
Wartość wraz z podatkiem: 0
```

Wartość `null` zostaje przez operator mnożenia zastąpiona liczbą 0. Z jednej strony w niektórych projektach będą to rozsądne dane wyjściowe, choć z drugiej strony jest to ten rodzaj danych wyjściowych, które „po cichu” mogą mieć wartość `null` i wprowadzać użytkownika w zakłopotanie podczas działania programu. Opcja kompilatora `strictNullChecks` nie pozwala na używanie wartości `null` i `undefined` dla wszystkich typów, jak to omówiłem w rozdziale 7., i wymaga parametrów, które mogą akceptować wartości `null` przeznaczone do użycia w unii typów. Włączenie wymienionej opcji kompilatora pokazałem na listingu 8.13.

Listing 8.13. Zmienianie opcji kompilatora w pliku `tsconfig.json` w katalogu `types`.

```
{
  "compilerOptions": {
    "target": "es2020",
    "outDir": "./dist",
    "rootDir": "./src",
    "declaration": true,
    "strictNullChecks": true
  }
}
```

Po zapisaniu pliku konfiguracyjnego kompilator zostanie uruchomiony i spowoduje wygenerowanie następującego komunikatu błędu wskazującego na użycie argumentu o wartości `null`:

```
src/index.ts(6,29): error TS2345: Argument of type 'null' is not assignable to parameter of type 'number'.
```

Gdy wartość `null` ma być dozwolona, parametr można zdefiniować za pomocą unii typów, jak pokazałem na listingu 8.14.

Listing 8.14. Zezwalanie na używanie wartości `null` parametru w kodzie pliku `index.ts` w katalogu `src`

```
function calculateTax(amount: number | null, discount: number = 0,
  ...extraFees: number[]) {
  if (amount != null) {
    return (amount * 1.2) - discount
      + extraFees.reduce((total, val) => total + val, 0);
  }
}
```

```
let taxValue = calculateTax(null, 0);
console.log(`Wartość wraz z podatkiem: ${taxValue}`);
```

Aby uniemożliwić używanie wartości `null` z operatorem mnożenia, konieczne jest wykorzystanie wartownika typu. Gdy dopiero rozpoczynasz programowanie w TypeScriptie, może się to wydawać uciążliwe, ale ograniczenie parametrów typu `null` może wyeliminować problemy, które w przeciwnym razie prowadziłyby do nieoczekiwanych wyników. Kod przedstawiony na listingu 8.14 powoduje wygenerowanie następujących danych wyjściowych:

```
Wartość wraz z podatkiem: undefined
```

Wynik działania funkcji

Kompilator TypeScriptu próbuje ustalić typ wyniku działania funkcji na podstawie jej kodu i automatycznie będzie używać unii, gdy funkcja może zwracać wiele typów. Najłatwiejszy sposób na sprawdzenie, jaki typ został przez kompilator określony dla wyniku działania funkcji, polega na włączeniu generowania plików deklaracji. Wymaga to ustawienia `declaration` w opcjach kompilatora, które zostało włączone w kodzie na listingu 8.2. Wymienione pliki deklaracji są używane w celu dostarczenia informacji typu, gdy pakiet jest wykorzystywany w innym projekcie TypeScriptu. Więcej informacji na ten temat znajdziesz w rozdziale 14.

Przeanalizuj zawartość pliku `index.d.ts` w katalogu `dist`, a poznasz szczegóły związane z typami, które kompilator określił lub ustalił na podstawie adnotacji typu.

```
declare function calculateTax(amount: number | null, discount?: number,
    ...extraFees: number[]): number | undefined;
declare let taxValue: number | undefined;
```

Pogrubione tutaj informacje typu dla funkcji `calculateTax()` pokazują typ określony przez kompilator dla wyniku działania funkcji.

Wyłączenie niejawnego zwracania wartości przez funkcję

JavaScript ma wiele nietypowo luźnych podejść do wyniku działania funkcji, np. zwrot wartości `undefined` dla każdej ścieżki kodu, która nie kończy się poleceniem zawierającym słowo kluczowe `return` — jest to określane mianem funkcjonalności *niejawnego zwracania wartości przez funkcję*.

W omawianym przykładzie, gdy wartownik typu jest używany do odfiltrowania wartości `null`, oznacza to istnienie w kodzie funkcji ścieżki, która nie kończy się poleceniem zawierającym słowo kluczowe `return`, więc funkcja zwróci wartość typu `number`, jeśli parametr jest inny niż `null`, lub `undefined`, jeśli parametr `amount` ma wartość `null`. Opcja kompilatora `strictNullChecks` została włączona w kodzie przedstawionym na listingu 8.14 i kompilator ustalił typ wyniku jako `number | undefined`.

Jeżeli chcesz uniemożliwić niejawne zwracanie wartości przez funkcję, musisz ustawić opcje kompilatora tak, jak pokazałem na listingu 8.15.

Listing 8.15. Zmiana konfiguracji kompilatora w pliku `tsconfig.json` w katalogu `types`

```
{
  "compilerOptions": {
    "target": "es2020",
    "outDir": "./dist",
    "rootDir": "./src",
    "declaration": true,
    "strictNullChecks": true,
    "noImplicitReturns": true
  }
}
```

Gdy ustawienie `noImplicitReturns` ma przypisaną wartość `true`, kompilator zgłosi błąd w przypadku istnienia ścieżki kodu niegenerującej jawnie wyniku za pomocą słowa

kluczowego result lub po wystąpieniu błędu. Zapisz zmiany wprowadzone w pliku *tsconfig.json*. Powinieneś zobaczyć następujące dane wyjściowe kompilatora kompilującego plik *index.ts* z użyciem nowej konfiguracji:

```
src/index.ts(1,10): error TS7030: Not all code paths return a value.
```

Teraz każda ścieżka wykonywania funkcji musi się zakończyć wygenerowaniem wyniku. Wprawdzie funkcja nadal może zwrócić wartość *undefined*, ale to musi odbywać się jawnie, jak pokazałem na listingu 8.16.

Listing 8.16. Zwracanie wyniku w kodzie pliku *index.ts* w katalogu *src*

```
function calculateTax(amount: number | null, discount: number = 0,
  ...extraFees: number[]) {
  if (amount != null) {
    return (amount * 1.2) - discount
      + extraFees.reduce((total, val) => total + val, 0);
  } else {
    return undefined;
  }
}
```

```
let taxValue = calculateTax(null, 0);
console.log(`Wartość wraz z podatkiem: ${taxValue}`);
```

Wyłączenie niejawnego zwracania wyniku przez funkcję gwarantuje, że funkcja będzie musiała działać jawnie w zakresie generowania wyniku. Zmiana wprowadzona na listingu 8.16 powoduje usunięcie błędu generowanego przez kod na listingu 8.14, a wynikiem działania są teraz następujące dane wyjściowe:

```
Wartość wraz z podatkiem: undefined
```

Używanie adnotacji typu dla wyniku działania funkcji

Kompilator określa typ wyniku działania funkcji na podstawie analizy kodu jej ścieżek i tworzy unię napotkanych typów. Preferuję używanie adnotacji typu, aby wyraźnie określać typ wyniku, ponieważ pozwala mi to na zadeklarowanie oczekiwanego wyniku działania funkcji, a nie tego, który będzie wygenerowany przez kod. W ten sposób mam pewność, że nie użyję przypadkowo nieprawidłowego typu. Adnotacje dla wyniku działania funkcji są umieszczane na końcu sygnatury funkcji, jak pokazałem na listingu 8.17.

Listing 8.17. Adnotacje typu wyniku działania funkcji umieszczone w kodzie pliku *index.ts* w katalogu *src*

```
function calculateTax(amount: number, discount: number = 0,
  ...extraFees: number[]): number {
  return (amount * 1.2) - discount
    + extraFees.reduce((total, val) => total + val, 0);
}
```

```
let taxValue = calculateTax(100, 0);
console.log(`Wartość wraz z podatkiem: ${taxValue}`);
```

Typ wyniku zdefiniowałem jako `number` i usunąłem typ `null` z parametru `amount`. Jawne zadeklarowanie typu oznacza, że kompilator zgłosi błąd, jeśli funkcja przypadkowo zwróci dane innego typu. Po skompilowaniu i uruchomieniu kodu przedstawionego na listingu 8.17 zostaną wygenerowane następujące dane wyjściowe:

Wartość wraz z podatkiem: 120

Definiowanie funkcji typu `void`

Funkcja, która nie generuje wyniku, jest deklarowana jako typu `void`. Przykład takiej funkcji przedstawiłem na listingu 8.18.

Listing 8.18. Definiowanie funkcji typu `void` w kodzie pliku `index.ts` w katalogu `src`

```
function calculateTax(amount: number, discount: number = 0,
    ...extraFees: number[]): number {
    return (amount * 1.2) - discount
        + extraFees.reduce((total, val) => total + val, 0);
}

function writeValue(label: string, value: number): void {
    console.log(`${label}: ${value}`);
}

writeValue("Wartość wraz z podatkiem", calculateTax(100, 0));
```

Funkcja `writeValue()` nie zwraca wyniku i otrzymała adnotację typu `void`. Zastosowanie typu `void` gwarantuje, że kompilator nie będzie generował ostrzeżenia, gdy użyte zostanie słowo kluczowe `return` lub jeśli funkcja jest używana do przypisania wartości.

■ **Uwaga** Typ `never` może zostać użyty jako typ wyniku funkcji, której działanie nigdy nie zostanie ukończone, np. takiej, która zawsze zgłasza wyjątek.

Kod przedstawiony na listingu 8.18 powoduje wygenerowanie następujących danych wyjściowych:

Wartość wraz z podatkiem: 120

Przeciążanie typu funkcji

Unia pozwala na zdefiniowanie zakresu typów dla parametrów i wyników funkcji, ale nie umożliwia istnienia między nimi relacji, którą można dokładnie wyrazić, jak pokazałem na listingu 8.19.

Listing 8.19. Definiowanie funkcji z unią w pliku `index.ts` w katalogu `src`

```
function calculateTax(amount: number | null): number | null {
    if (amount !== null) {
        return amount * 1.2;
    }
}
```

```

    return null;
}

function writeValue(label: string, value: number): void {
    console.log(`${label}: ${value}`);
}

let taxAmount: number | null = calculateTax(100);
if (typeof taxAmount === "number") {
    writeValue("Wartość wraz z podatkiem", taxAmount);
}

```

Adnotacja typu na listingu 8.19 opisuje typy akceptowane przez funkcję `calculateTax()` i informuje jej użytkowników o akceptacji wartości typu `number` lub `null` oraz o zwrocie wartości typu `number` lub `null`. Informacje dostarczone przez unię typów są prawidłowe, choć niekoniecznie w pełni przedstawiają sytuację. Brakuje relacji między typami parametru i wyniku — funkcja zawsze będzie zwracać wynik typu `number`, jeśli parametr `amount` również jest typu `number`, i zawsze zwróci wynik typu `null`, jeśli parametr `amount` jest typu `null`. Brakujące szczegóły związane z typami funkcji oznaczają, że użytkownik funkcji będzie musiał zastosować wartownika typu dla wyniku, aby usunąć wartości `null`, nawet jeśli wartość 100 jest typu `number` i zawsze będzie generowała wynik typu `number`.

W celu zapewnienia możliwości opisanego relacji między typami używanymi przez funkcję TypeScript pozwala na przeciążanie typu, jak pokazałem na listingu 8.20.

■ **Uwaga** To nie jest rodzaj przeciążania funkcji znany z innych języków programowania, takich jak C# i Java. Przeciążane są tylko informacje o typie na potrzeby operacji sprawdzania typu. Jak widać na listingu 8.20, istnieje tylko jedna implementacja funkcji, która nadal pozostaje odpowiedzialna za obsługę wszystkich typów używanych podczas przeciążania.

Listing 8.20. Przeciążenie typów funkcji w kodzie pliku `index.ts` w katalogu `src`

```

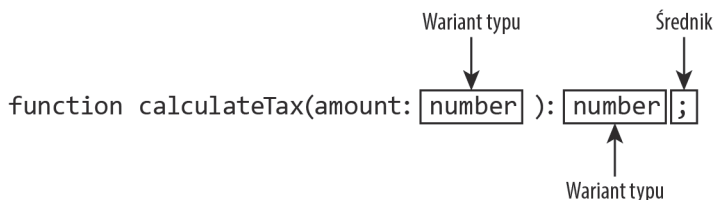
function calculateTax(amount: number): number;
function calculateTax(amount: null): null;
function calculateTax(amount: number | null): number | null {
    if (amount !== null) {
        return amount * 1.2;
    }
    return null;
}

function writeValue(label: string, value: number): void {
    console.log(`${label}: ${value}`);
}

let taxAmount: number = calculateTax(100);
//if (typeof taxAmount === "number") {
    writeValue("Wartość wraz z podatkiem", taxAmount);
//}

```

Każdy przeciążony typ definiuje połączenie typów obsługiwane przez funkcję oraz przedstawia mapowanie między parametrami i generowanymi przez nie wynikami, jak pokazałem na rysunku 8.4.



Rysunek 8.4. Przeciążenie typu funkcji

Przeciążenie typu zastępuje definicję funkcji jako informacje typu używane przez kompilator TypeScriptu. Oznacza to, że mogą być używane tylko wymienione warianty typu. Po wywołaniu funkcji kompilator ma możliwość ustalenia typu wyniku na podstawie typu dostarczonych argumentów. W omawianym przykładzie pozwala to na zdefiniowanie zmiennej `taxAmount` jako typu `number` i eliminuje potrzebę stosowania wartownika typu podczas przekazywania wyniku do funkcji `writeValue()`. Kompilator ustala, że zmienna `taxAmount` może mieć wartość jedynie typu `number`, więc nie ma konieczności zawężania jej typu. Po skompilowaniu i uruchomieniu kodu przedstawionego na listingu 8.20 zostaną wygenerowane następujące dane wyjściowe:

Wartość wraz z podatkiem: 120

- **Wskazówka** Relacje między parametrami i wynikami można wyrazić również za pomocą typów warunkowych, co dokładnie przedstawię w rozdziale 13.
-

Funkcje asercji

Funkcja asercji sprawdza wyrażenie warunkowe i zwykle zgłasza błąd, jeśli jego wynik jest inny niż `true`. Funkcja asercji jest czasami używana jako wartownik typu w czystym kodzie JavaScriptu, w którym są niedostępne typy statyczne TypeScriptu. Problem z funkcją asercji polega na tym, że kompilator TypeScriptu nie potrafi ustalić wpływu działania funkcji asercji dla danego typu, jak pokazałem na listingu 8.21.

Listing 8.21. Przykład użycia funkcji asercji w pliku `index.ts` w katalogu `src`

```

function check(expression: boolean) {
    if (!expression) {
        throw new Error("Wartością wyrażenia jest false");
    }
}

function calculateTax(amount: number | null): number {
    check(typeof amount == "number");
}

```



```

    return amount * 1.2;
}

let taxAmount: number | null = calculateTax(100);
console.log(`Wartość wraz z podatkiem: ${taxAmount}`)

```

Funkcja `check()` definiuje parametr typu `boolean` i zgłasza błąd, jeśli jego wartością jest `false`. Jest to podstawowy wzorzec działania funkcji asercji.

Funkcja `calculateTax()` akceptuje argument typu `number | null` i używa funkcji `check()` do zawężenia typu. Dzięki temu wartość typu `null` spowoduje wyświetlenie komunikatu błędu, a wartość typu `number` będzie stosowana do wygenerowania wyniku.

Jednak problem z omówionym przykładem polega na tym, że funkcja `check()` oznacza przetwarzanie tylko wartości typu `number`, o czym kompilator TypeScriptu nie wie. Po skompilowaniu tego kodu zostanie wygenerowany następujący komunikat błędu:

```
src/index.ts(9,12): error TS2531: Object is possibly 'null'.
```

Słowa kluczowego `asserts` można użyć do wskazania funkcji asercji, co pozwoli kompilatorowi TypeScriptu uwzględnić ją podczas kompilacji kodu źródłowego, jak pokazałem na listingu 8.22.

Listing 8.22. Wyraźne oznaczenie funkcji asercji w pliku `index.ts` w katalogu `src`

```

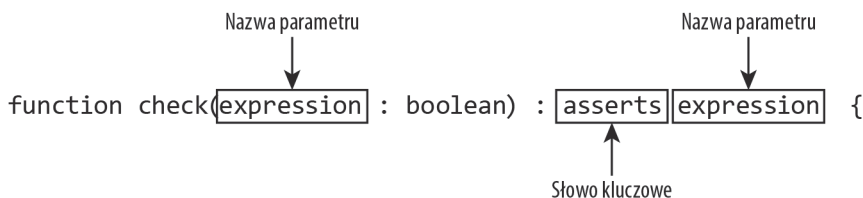
function check(expression: boolean) : asserts expression {
    if (!expression) {
        throw new Error("Wartością wyrażenia jest false");
    }
}

function calculateTax(amount: number | null): number {
    check(typeof amount == "number");
    return amount * 1.2;
}

let taxAmount: number | null = calculateTax(100);
console.log(`Wartość wraz z podatkiem: ${taxAmount}`)

```

Słowo kluczowe `asserts` zostało użyte podobnie jak typ wyniku. Po nim znajduje się nazwa parametru sprawdzanego przez funkcję asercji, jak pokazałem na rysunku 8.5.



Rysunek 8.5. Wyraźne oznaczenie funkcji asercji

Kompilator TypeScriptu może wziąć pod uwagę wynik wykonania funkcji `check()` i wie, że funkcja `calculateTax()` zawęży typ parametru `amount`, aby wykluczyć wartości `null`.

Istnieje wariant funkcji asercji bezpośrednio działający na typie, a nie sprawdzający wyrażenie. Przykład takiej funkcji pokazałem na listingu 8.23.

Listing 8.23. Bezpośrednie zawężenie typu w kodzie pliku `index.ts` w katalogu `src`

```
function checkNumber(val: any): asserts val is number {
    if (typeof val !== "number") {
        throw new Error("To nie jest wartość typu number");
    }
}

function calculateTax(amount: number | null): number {
    checkNumber(amount);
    return amount * 1.2;
}

let taxAmount: number = calculateTax(100);
console.log(`Wartość wraz z podatkiem: ${taxAmount}`)
```

W tym przykładzie po słowie kluczowym `asserts` mamy wyrażenie `val is number` wskazujące kompilatorowi TypeScriptu zadanie funkcji `checkNumber()` — zagwarantowanie, że parametr `val` ma wartość typu `number`.

Podsumowanie

W tym rozdziale omówiłem funkcjonalność TypeScriptu oferowaną do obsługi funkcji. Dowiedziałeś się, że nie można powielać definicji funkcji, poznałeś różne sposoby na opisanie parametrów i wyników działania funkcji, a także zobaczyłeś, jak nadpisywać typy funkcji, aby zdefiniować dokładniejsze mapowanie między typami parametrów i generowanymi przez nie wynikami. W następnym rozdziale pokażę, jak TypeScript obsługuje proste struktury danych.

ROZDZIAŁ 9.



Tablice, krotki i wyliczenia

Przykłady przedstawione dotychczas w książce koncentrowały się na prostych typach danych, co pozwoliło mi zapoznać Cię z podstawowymi funkcjonalnościami języka TypeScript. W rzeczywistych projektach powiązane ze sobą właściwości danych są grupowane i tworzą obiekty. W tym rozdziale zaprezentuję oferowaną przez TypeScript obsługę prostych struktur danych, a zacznę od tablic. Streszczenie materiału przedstawionego w rozdziale znajdziesz w tabeli 9.1.

Tabela 9.1. Streszczenie materiału przedstawionego w rozdziale

Problem	Rozwiązanie	Listing
Ograniczenie zakresu typów danych, które znajdują się w tablicy	Zastosuj adnotację typu lub zezwól kompilatorowi na ustalenie typu na podstawie wartości używanej do inicjalizacji tablicy	Od 4 do 9
Zdefiniowanie tablicy o stałej wielkości z określonymi typami dla każdej wartości	Użyj krotki	Od 10 do 14
Zdefiniowanie tablicy o zmiennej wielkości z określonymi typami dla każdej wartości	Użyj krotki z elementem resztowym	15
Używanie pojedynczej nazwy w celu odwoływania się do kolekcji powiązanych ze sobą wartości	Użyj typu wyliczeniowego	Od 16 do 25
Definiowanie typu powiązanego tylko z określoną wartością	Użyj literału wartości danego typu	Od 26 do 32
Unikanie powielania podczas opisywania typu złożonego	Użyj aliasu typu	33

W tabeli 9.2 wymieniałem opcje kompilatora TypeScriptu użyte w rozdziale.

Tabela 9.2. Opcje kompilatora TypeScriptu użyte w rozdziale

Opcja	Opis
target	Ta opcja określa docelową wersję języka JavaScript, dla której kompilator wygeneruje kod
outDir	Ta opcja określa katalog, w którym zostaną umieszczone wygenerowane pliki JavaScriptu
rootDir	Ta opcja określa katalog główny używany przez kompilator do wyszukiwania plików TypeScriptu
declaration	Ta opcja generuje pliki deklaracji typu, które pomagają w zrozumieniu, w jaki sposób zostały ustalone typy dla kodu JavaScriptu. Wspomniane pliki zostaną szczegółowo omówione w rozdziale 14.
strictNullChecks	Ta opcja uniemożliwia używanie null i undefined jako wartości dla innych typów

Przygotowanie projektu

W rozdziale będę kontynuował używanie projektu *types* utworzonego w rozdziale 7. Aby przygotować się do tego rozdziału, należy zastąpić zawartość pliku *index.ts* w katalogu *src* kodem przedstawionym na listingu 9.1.

■ **Wskazówka** Przykładowy projekt dla tego rozdziału, a także projekty dla pozostałych rozdziałów książki zostały umieszczone w archiwum dostępnym pod adresem <https://ftp.helion.pl/przyklady/tys4o2.zip>.

Listing 9.1. Nowa zawartość pliku *index.ts* w katalogu *src*

```
function calculateTax(amount: number): number {
    return amount * 1.2;
}

function writePrice(product: string, price: number): void {
    console.log(`Cena produktu ${product}: ${price.toFixed(2)} zł`);
}

let hatPrice = 100;
let glovesPrice = 75;
let umbrellaPrice = 42;

writePrice("czapka", calculateTax(hatPrice));
writePrice("rękawiczki", calculateTax(glovesPrice));
writePrice("parasol", calculateTax(umbrellaPrice));
```

Umieść znaki komentarza na początku opcji kompilatora pokazanych na listingu 9.2, aby w ten sposób wyzerować konfigurację kompilatora.

Listing 9.2. Wyłączanie opcji kompilatora w pliku *tsconfig.json* w katalogu *types*

```
{
  "compilerOptions": {
    "target": "es2020",
    "outDir": "./dist",
    "rootDir": "./src",
    "declaration": true,
    // "strictNullChecks": true,
    // "noImplicitReturns": true
  }
}
```

Otwórz nowe okno powłoki, przejdź do katalogu *types*, a następnie z jego poziomu wydaj polecenie przedstawione na listingu 9.3, uruchamiające kompilator TypeScriptu w trybie automatycznego kompilowania kodu po wykryciu zmiany w dowolnym pliku i wykonywania kodu po jego skompilowaniu.

Listing 9.3. Uruchamianie kompilatora TypeScriptu

```
$ npm start
```

Kompilator przeprowadzi kompilację kodu w pliku *index.ts*, wykona wygenerowany kod JavaScriptu, a następnie przejdzie do trybu monitorowania i wyświetli następujące dane wyjściowe:

```
6:58:20 AM - File change detected. Starting incremental compilation...
6:58:21 AM - Found 0 errors. Watching for file changes.
Cena produktu czapka: 120.00 zł
Cena produktu rękawiczki: 90.00 zł
Cena produktu parasol: 50.40 zł
```

Praca z tablicami

Jak wyjaśniłem w rozdziale 8., tablice JavaScriptu mogą zawierać połączenie różnych typów i mają zmienną wielkość, co oznacza możliwość dynamicznego dodawania i usuwania wartości bez potrzeby wyraźnej zmiany wielkości tablicy. TypeScript nie zmienia elastyczności w zakresie zmiany wielkości tablic, ale pozwala na ograniczanie za pomocą adnotacji typów danych znajdujących się w tablicy. Spójrz na przykładowy program przedstawiony na listingu 9.4.

Listing 9.4. Używanie tablicy w kodzie pliku *index.ts* w katalogu *src*

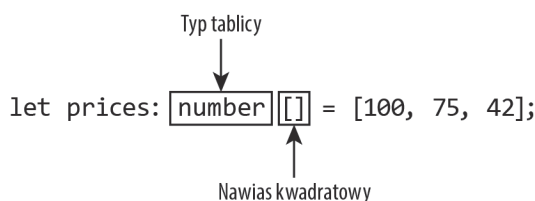
```
function calculateTax(amount: number): number {
  return amount * 1.2;
}

function writePrice(product: string, price: number): void {
  console.log(`Cena produktu ${product}: ${price.toFixed(2)} zł`);
}
```

```
let prices: number[] = [100, 75, 42];
let names: string[] = ["czapka", "rękawiczki", "parasol"];

writePrice(names[0], calculateTax(prices[0]));
writePrice(names[1], calculateTax(prices[1]));
writePrice(names[2], calculateTax(prices[2]));
```

Typ tablicy jest określany przez umieszczenie nawiasu kwadratowego po nazwie typu w adnotacji, jak pokazałem na rysunku 9.1.



Rysunek 9.1. Adnotacja typu tablicy

TypeScript używa adnotacji, aby ograniczyć do podanego typu operacje, które będą mogły być przeprowadzane w tablicy. Pierwsza tablica zdefiniowana na listingu 9.4 została ograniczona do wartości typu `number`, druga do wartości typu `string`. Na listingu 9.5 pokazałem przykład użycia metody JavaScriptu `forEach()` w tablicy — jak widać, funkcja odpowiedzialna za przetwarzanie wartości tablicy ma zdefiniowane typy odpowiadające typom tablicy.

-
- **Wskazówka** Istnieje możliwość użycia nawiasu podczas opisywania tablicy zawierającej wiele typów, np. podczas pracy z nią typów (rozdział 8.) lub ze złączeniami typów (rozdział 10.). Na przykład tablicę, której elementy mogą być wartościami typu `number` lub `string`, można określić adnotacją `(number | string)[]`, a nawias wokół unii typu uniemożliwia kompilatorowi przyjęcie założenia o istnieniu unii między pojedynczą liczbą lub tablicą ciągów tekstowych.
-

Listing 9.5. Przeprowadzanie operacji na tablicach typowanych w kodzie pliku `index.ts` w katalogu `src`

```
function calculateTax(amount: number): number {
    return amount * 1.2;
}

function writePrice(product: string, price: number): void {
    console.log(`Cena produktu ${product}: ${price.toFixed(2)} zł`);
}

let prices: number[] = [100, 75, 42];
let names: string[] = ["czapka", "rękawiczki", "parasol"];

prices.forEach((price: number, index: number) => {
    writePrice(names[index], calculateTax(price));
});
```

Pierwszy argument funkcji przekazanej metodzie `forEach()` otrzymuje wartość typu `number`, ponieważ jest to typ przetwarzanej tablicy. TypeScript zagwarantuje, że przez tę funkcję będą przeprowadzane tylko operacje dozwolone dla wartości typu `number`. Po skompilowaniu i uruchomieniu kodu przedstawionego na listingu 9.5 zostaną wygenerowane następujące dane wyjściowe:

```
Cena produktu czapka: 120.00 zł
Cena produktu rękawiczki: 90.00 zł
Cena produktu parasol: 50.40 zł
```

Składnia tablicy

Typ tablicy może być wyrażony również za pomocą składni nawiasu ostrego. Dlatego też polecenie:

```
...
let prices: number[] = [100, 75, 42];
...
```

jest odpowiednikiem następującego:

```
...
let prices: Array<number> = [100, 75, 42];
...
```

Problem związany z tą składnią polega na tym, że nie można jej stosować w plikach TSX łączących elementy HTML-a z kodem TypeScriptu, jak pokazałem w rozdziale 15. Dlatego też składnia nawiasu kwadratowego jest preferowanym sposobem na asercję typu tablicy.

Używanie automatycznie ustalonego typu tablicy

Na listingu 9.5 użyłem adnotacji typu, aby wyraźnie wskazać typ tablicy. Jednak kompilator TypeScriptu ma możliwość automatycznego ustalenia typu, więc pewne przykłady mogą być wyrażone bez adnotacji typu, jak pokazałem na listingu 9.6.

Listing 9.6. Używanie automatycznie ustalanych typów tablic w kodzie pliku `index.ts` w katalogu `src`

```
function calculateTax(amount: number): number {
    return amount * 1.2;
}

function writePrice(product: string, price: number): void {
    console.log(`Cena produktu ${product}: ${price.toFixed(2)} zł`);
}

let prices = [100, 75, 42];
let names = ["czapka", "rękawiczki", "parasol"];

prices.forEach((price, index) => {
    writePrice(names[index], calculateTax(price));
});
```

Kompilator ma możliwość określenia typu tablicy na podstawie zbioru wartości przypisywanych podczas inicjalizacji tablicy. Następnie ustalone typy są używane podczas wykonywania metody `forEach()`.

Wprawdzie kompilator potrafi automatycznie określać typy, ale jeśli nie otrzymasz oczekiwanych wyników, zawsze możesz sprawdzić wygenerowane przez niego pliki, gdy włączona jest opcja `declaration`. Ta opcja powoduje wygenerowanie plików deklaracji typu, które są używane w celu dostarczenia informacji typu, gdy pakiet jest wykorzystywany w innym projekcie TypeScriptu, co dokładnie przedstawię w rozdziale 14.

Spójrz na typy określone przez kompilator dla tablic przedstawionych na listingu 9.6. Ten fragment kodu pochodzi z pliku `index.d.ts` wygenerowanego w katalogu `dist`.

```
...
declare let prices: number[];
declare let names: string[];
...
```

Dokładne omówienie słowa kluczowego `declare` znajdziesz w rozdziale 14. W tym momencie wystarczy pamiętać, że kompilator prawidłowo określił typy tablic na podstawie ich wartości początkowych.

Unikanie problemów z automatycznie ustalaniem typu tablicy

Kompilator ustala typ tablicy na podstawie wartości umieszczanych w tablicy podczas jej tworzenia. Może to prowadzić do błędów, jeśli dojdzie do przypadkowego pomieszczenia wartości używanych do wypełnienia tablicy, jak pokazałem na listingu 9.7.

Listing 9.7. *Używanie w tablicy elementów różnych typów w kodzie pliku `index.ts` w katalogu `src`*

```
function calculateTax(amount: number): number {
    return amount * 1.2;
}

function writePrice(product: string, price: number): void {
    console.log(`Cena produktu ${product}: ${price.toFixed(2)} zł`);
}

let prices = [100, 75, 42, "20"];
let names = ["czapka", "rękawiczki", "parasol", "okulary"];

prices.forEach((price, index) => {
    writePrice(names[index], calculateTax(price));
});
```

Nowa wartość użyta do inicjalizacji tablicy `price` spowodowała wygenerowanie następującego błędu po skompilowaniu kodu:

```
src/index.ts(13,43): error TS2345: Argument of type 'string | number' is not assignable to
parameter of type 'number'.
```

Jeżeli przeanalizujesz zawartość pliku *index.d.ts* w katalogu *dist*, wówczas zobaczysz, że kompilator TypeScriptu wybrał najmniejszy zbiór typów, który może opisać wartości używane do inicjalizacji tablicy:

```
declare let prices: (string | number)[];
```

Zmiana w typie tablicy powoduje wygenerowanie komunikatu błędu, ponieważ funkcja przekazana metodzie `forEach()` traktuje wartości jako liczby (typ `number`), podczas gdy są one teraz częścią unii `string | number`. Bardzo łatwo jest dostrzec przyczynę problemu w tym prostym przykładzie, natomiast staje się to znacznie trudniejsze, gdy wartości początkowe tablicy pochodzą z różnych części aplikacji. Wyraźne zadeklarowanie typu tablicy uważam za dużo bardziej użyteczne rozwiązanie, choć oznacza to, że problemy takie jak pokazany w kodzie na listingu 9.7 spowodują wygenerowanie komunikatu błędu wskazującego na próbę dodania wartości typu `string` do tablicy elementów typu `number`.

Unikanie problemów z pustą tablicą

Kolejnym powodem używania adnotacji typu dla tablicy jest to, że kompilator określa typ *any* dla tablicy utworzonej jako pusta, jak pokazałem na listingu 9.8.

Listing 9.8. Tworzenie pustej tablicy w kodzie pliku *index.ts* w katalogu *src*

```
function calculateTax(amount: number): number {
    return amount * 1.2;
}

function writePrice(product: string, price: number): void {
    console.log(`Cena produktu ${product}: ${price.toFixed(2)} zł`);
}

let prices = [];
prices.push(...[100, 75, 42, "20"]);
let names = ["czapka", "rękawiczki", "parasol", "okulary"];

prices.forEach((price, index) => {
    writePrice(names[index], calculateTax(price));
});
```

Nie istnieje żadna wartość początkowa możliwa do użycia przez kompilator ustalający typ tablicy `prices`. Jedną możliwością dostępną dla kompilatora jest użycie typu `any`, ponieważ nie dysponuje żadnymi informacjami, o czym możesz się przekonać, analizując zawartość pliku *index.d.ts* w katalogu *dist*.

```
declare let prices: any[];
```

Mimo że wartości dodane do tablicy to połączenie elementów typu `number` i `string`, kod przedstawiony na listingu 9.8 zostanie skompilowany bez błędów i wygeneruje następujące dane wyjściowe:

```
Cena produktu czapka: 120.00 zł
Cena produktu rękawiczki: 90.00 zł
Cena produktu parasol: 50.40 zł
Cena produktu okulary: 24.00 zł
```

Efektem pozostawienia kompilatorowi zadania, jakim jest określenie typu pustej tablicy, będzie powstanie luki w procesie sprawdzania typu. Wymieniony kod działa, ponieważ operator mnożenia w JavaScriptcie automatycznie przeprowadza koercję wartości typu string na typ number. Wprawdzie może to być użyteczne zachowanie, ale prawdopodobnie będzie przypadkowe i dlatego też należy używać wyraźnie zdefiniowanych typów.

Problemy związane z tablicą typu never

TypeScript inaczej ustala typ dla pustej tablicy, gdy wartości null i undefined nie są możliwe do przypisania innym typom. Aby zobaczyć różnicę, ustawienia kompilatora zmodyfikuj w taki sposób, by odpowiadały przedstawionym na listingu 9.9.

Listing 9.9. Konfigurowanie kompilatora w pliku tsconfig.json w katalogu types

```
{
  "compilerOptions": {
    "target": "es2020",
    "outDir": "./dist",
    "rootDir": "./src",
    "declaration": true,
    "strictNullChecks": true
  }
}
```

Ustawienie strictNullChecks nakazuje kompilatorowi ograniczenie użycia wartości null i undefined, a także uniemożliwia mu zastosowanie typu any podczas ustalania typu pustej tablicy. Zamiast tego kompilator określa typ tablicy jako never, co oznacza brak możliwości dodawania elementów do tablicy. Po skompilowaniu i uruchomieniu kodu przedstawionego na listingu 9.9 zostanie wygenerowany następujący komunikat błędu:

```
src/index.ts(10,13): error TS2345: Argument of type 'string | number' is not assignable to
  ↳ parameter of type 'never'.
```

Określenie typu never gwarantuje, że tablica nie uniknie procesu sprawdzania typu, a kod nie zostanie skompilowany, póki nie będzie zdefiniowany typ dla tablicy lub tablica nie zostanie zainicjalizowana z wartościami pozwalającymi kompilatorowi na określenie mniej restrykcyjnego typu.

Krotka

Krotka to stałej wielkości tablica, której poszczególne elementy mogą być różnych typów. To struktura danych dostarczana przez kompilator TypeScriptu i implementowana za pomocą zwykłych tablic JavaScriptu w skompilowanym kodzie.

Na listingu 9.10 pokazałem przykład definiowania i używania krotki. (W dalszej części rozdziału przedstawię znacznie bardziej skomplikowany typ krotki).

Listing 9.10. *Używanie krotki w kodzie pliku `index.ts` w katalogu `src`*

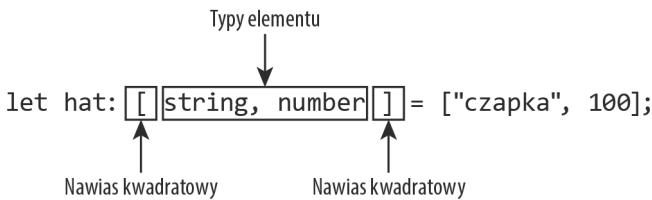
```
function calculateTax(amount: number): number {
    return amount * 1.2;
}

function writePrice(product: string, price: number): void {
    console.log(`Cena produktu ${product}: ${price.toFixed(2)} zł`);
}

let hat: [string, number] = ["czapka", 100];
let gloves: [string, number] = ["rękawiczki", 75];

writePrice(hat[0], hat[1]);
writePrice(gloves[0], gloves[1]);
```

Krotka jest definiowana za pomocą nawiasu kwadratowego zawierającego rozdzielone przecinkami typy dla poszczególnych elementów, jak pokazałem na rysunku 9.2.



Rysunek 9.2. *Definiowanie krotki*

Typ krotki `hat` przedstawionej na listingu 9.10 to `[string, number]` — definiuje on krotkę z dwoma elementami, z których pierwszy to ciąg tekstowy, a drugi to liczba. Dostęp do elementów krotki odbywa się za pomocą składni indeksu tablicy, więc pierwszy element omawianej krotki jest dostępny jako `hat[0]`.

Po skompilowaniu i uruchomieniu kodu przedstawionego na listingu 9.10 zostaną wygenerowane następujące dane wyjściowe:

```
Cena produktu czapka: 100.00 zł
Cena produktu rękawiczki: 75.00 zł
```

Krotka musi być zdefiniowana z adnotacjami typu, ponieważ w przeciwnym razie kompilator przyjmie założenie, że ma do czynienia ze zwykłą tablicą będącą unią poszczególnych wartości użytych podczas inicjalizacji. Bez adnotacji typu pokazanej na rysunku 9.2 kompilator uzna, że typem wartości przypisanej zmiennej `hat` jest `[string, number]`, co wskazuje na zmienną wielkości tablicę, w której każdy element może być typu `string` lub `number`.

Przetwarzanie krotki

Ograniczenia dotyczące liczby elementów i ich typów są narzucane w całości przez kompilator TypeScriptu, natomiast podczas działania aplikacji krotka jest implementowana jako zwykła tablica JavaScriptu. Oznacza to możliwość używania krotki ze standardowymi funkcjami tablicy JavaScriptu, jak pokazałem na listingu 9.11.

Listing 9.11. Przetwarzanie elementów krotki w kodzie pliku *index.ts* w katalogu *src*

```
function calculateTax(amount: number): number {
    return amount * 1.2;
}

function writePrice(product: string, price: number): void {
    console.log(`Cena produktu ${product}: ${price.toFixed(2)} zł`);
}

let hat: [string, number] = ["czapka", 100];
let gloves: [string, number] = ["rękawiczki", 75];

hat.forEach((h: string | number) => {
    if (typeof h === "string") {
        console.log(`Wartość typu string: ${h}`);
    } else {
        console.log(`Wartość typu number: ${h.toFixed(2)}`);
    }
});
```

Aby przetworzyć wszystkie wartości krotki, funkcja przekazana metodzie `forEach()` musi otrzymywać wartości typu `string | number`, które później są zawężane za pomocą wartownika typu. Adnotacje zastosowałem w celu zapewnienia większej przejrzystości kodu, ale kompilator prawidłowo określa unię typów na podstawie typu elementów krotki. Po skompilowaniu i uruchomieniu kodu przedstawionego na listingu 9.11 zostaną wygenerowane następujące dane wyjściowe:

```
Wartość typu string: czapka
Wartość typu number: 100.00
```

Skoro krotka jest tablicą, można przeprowadzić jej destrukuryzację w celu uzyskania dostępu do poszczególnych wartości. To ułatwi pracę z krotką, jak pokazałem na listingu 9.12.

Listing 9.12. Przykład destrukuryzacji krotki w kodzie pliku *index.ts* w katalogu *src*

```
function calculateTax(amount: number): number {
    return amount * 1.2;
}

function writePrice(product: string, price: number): void {
    console.log(`Cena produktu ${product}: ${price.toFixed(2)} zł`);
}

let hat: [string, number] = ["czapka", 100];
```

```
let gloves: [string, number] = ["rękawiczki", 75];
```

```
let [hatname, hatprice] = hat;
console.log(`Produkt: ${hatname}`);
console.log(`Cena: ${hatprice.toFixed(2)}`);
```

Krotka `hat` została poddana destrukuryzacji, a jej wartości zostały przypisane zmiennym `hatname` i `hatprice`, które następnie wyświetlono w konsoli. Ten przykład powoduje wygenerowanie dokładnie takich samych danych wyjściowych jak wcześniej. Jedyna różnica wiąże się ze sposobem uzyskania dostępu do wartości krotki.

Używanie typów krotki

Krotka to oddzielny typ, który może być używany dokładnie w taki sam sposób jak każdy inny typ. Dlatego też można tworzyć tablice krotek, podawać krotki w uniach typów, a także stosować wartowników typu w celu zawężenia wartości do określonego typu krotki. Przykłady tych wszystkich możliwości przedstawiłem na listingu 9.13.

Listing 9.13. Używanie typu krotki w kodzie pliku `index.ts` w katalogu `src`

```
function calculateTax(amount: number): number {
    return amount * 1.2;
}

function writePrice(product: string, price: number): void {
    console.log(`Cena produktu ${product}: ${price.toFixed(2)} zł`);
}

let hat: [string, number] = ["czapka", 100];
let gloves: [string, number] = ["rękawiczki", 75];

let products: [string, number][] = [["czapka", 100], ["rękawiczki", 75]];
let tupleUnion: ([string, number] | boolean)[] = [true, false, hat, ...products];

tupleUnion.forEach((elem: [string, number] | boolean) => {
    if (elem instanceof Array) {
        let [str, num] = elem;
        console.log(`Produkt: ${str}`);
        console.log(`Cena: ${num.toFixed(2)}`);
    } else if (typeof elem === "boolean") {
        console.log(`Wartość typu boolean: ${elem}`);
    }
});
```

Obfitość nawiasów kwadratowych może być dezorientująca i prawdopodobnie konieczne będzie przeprowadzenie kilku prób, zanim uda się prawidłowo opisać połączenie typów. Jednak ten przykład wyraźnie pokazuje, że typ krotki może być używany dokładnie tak samo jak każdy inny. Mimo to istnieje ważna różnica względem poprzednich przykładów przedstawionych w książce: w kodzie na listingu 9.13 nie można używać słowa kluczowego `typeof` w celu sprawdzenia, czy wartość jest krotką. Krotka jest implementowana za pomocą standardowej tablicy JavaScriptu i do sprawdzenia, czy dana wartość jest egzemplarzem typu tablicy,

konieczne jest użycie słowa kluczowego `instanceof`, jak to omówiłem w rozdziale 4. Po skompilowaniu i uruchomieniu kodu przedstawionego na listingu 9.13 zostaną wygenerowane następujące dane wyjściowe:

```
Wartość typu boolean: true
Wartość typu boolean: false
Wartość typu string: czapka
Wartość typu number: 100
Wartość typu string: czapka
Wartość typu number: 100
Wartość typu string: rękawiczki
Wartość typu number: 75
```

Używanie krotki z elementami opcjonalnymi

Krotka może zawierać elementy opcjonalne, wskazywane przez znak zapytania. Taka krotka wciąż ma stałą wielkość, a jeśli nie zostanie podana wartość elementu opcjonalnego, wówczas będzie nią `undefined`, jak przedstawiłem na listingu 9.14.

Listing 9.14. *Używanie krotki z elementem opcjonalnym w kodzie pliku `index.ts` w katalogu `src`*

```
function calculateTax(amount: number): number {
    return amount * 1.2;
}

function writePrice(product: string, price: number): void {
    console.log(`Cena produktu ${product}: ${price.toFixed(2)} zł`);
}

let hat: [string, number, number?] = ["czapka", 100];
let gloves: [string, number, number?] = ["rękawiczki", 75];

[hat, gloves].forEach(tuple => {
    let [name, price, taxRate] = tuple;
    if (taxRate != undefined) {
        price += price * (taxRate / 100);
    }
    writePrice(name, price);
});
```

Krotka przedstawiona na listingu 9.14 ma element opcjonalny `number`. (Krotka może mieć wiele elementów opcjonalnych, przy czym muszą być one zdefiniowane jako ostatnie).

Typem elementu opcjonalnego jest unia podanego typu i `undefined`, więc w omawianym przykładzie będzie to `number | undefined`. Jeżeli nie zostanie podana wartość elementu opcjonalnego, wówczas będzie nią `undefined`. Do kodu przetwarzającego krotkę należy zawężenie typu w celu wykluczenia wartości `undefined`.

Zdefiniowanie elementu opcjonalnego oznacza, że kompilator TypeScriptu nie będzie generował komunikatu błędu w przypadku braku wartości dla tego elementu, np. jak pokazałem w kolejnym fragmencie kodu.

```
...
let hat: [string, number, number?] = ["czapka", 100];
...
```

Wprawdzie nie została podana wartość dla trzeciego elementu krotki, ale kod i tak został skompilowany bezbłędnie. Po uruchomieniu kodu przedstawionego na listingu 9.14 zostaną wygenerowane następujące dane wyjściowe:

```
Cena produktu czapka: 100.00 zł
Cena produktu rękawiczki: 82.50 zł
```

Definiowanie krotki z elementem resztowym

Krotka może zawierać także element resztowy, pozwalający dopasować wiele wartości danego typu. Ta funkcjonalność powoduje wygenerowanie krotki o zmiennej wielkości i pozbawionej sztywnej struktury podstawowej krotki. Na listingu 9.15 pokazałem przykład krotki razem z elementem resztowym.

Listing 9.15. *Używanie elementu resztowego w kodzie pliku index.ts w katalogu src*

```
function calculateTax(amount: number): number {
    return amount * 1.2;
}

function writePrice(product: string, price: number): void {
    console.log(`Cena produktu ${product}: ${price.toFixed(2)} zł`);
}

let hat: [string, number, number?, ...number[]] = ["czapka", 100, 10, 1.20, 3, 0.95];
let gloves: [string, number, number?, ...number[]] = ["rękawiczki", 75, 10];

[hat, gloves].forEach(tuple => {
    let [name, price, taxRate, ...coupons] = tuple;
    if (taxRate != undefined) {
        price += price * (taxRate / 100);
    }
    coupons.forEach(c => price -= c);
    writePrice(name, price);
});
```

W omawianym przykładzie element resztowy krotki został zdestrukuryzowany na tablicę o nazwie coupons, przetwarzaną następnie przez pętlę forEach. Kod tego listingu powoduje wygenerowanie następujących danych wyjściowych:

```
Cena produktu czapka: 104.85 zł
Cena produktu rękawiczki: 82.50 zł
```

Nie lubię tej funkcjonalności, ponieważ zmienna wielkość krotki wprowadzana przez element resztowy narusza strukturę, dzięki której krotka jest tak użyteczna. Z tej funkcjonalności korzystam jedynie podczas opisywania kodu JavaScriptu, jak to przedstawię w rozdziale 14.

Wyliczenie

Wyliczenie pozwala na używanie kolekcji wartości za pomocą jednej nazwy, dzięki której kod staje się łatwiejszy w odczycie i można spójnie stosować pewny zbiór wartości. Podobnie jak w przypadku krotki, także wyliczenie to funkcjonalność oferowana przez kompilator TypeScriptu. Przykład zdefiniowania i użycia wyliczenia przedstawiłem na listingu 9.16.

Listing 9.16. Używanie wyliczenia w kodzie pliku *index.ts* w katalogu *src*

```
function calculateTax(amount: number): number {
    return amount * 1.2;
}

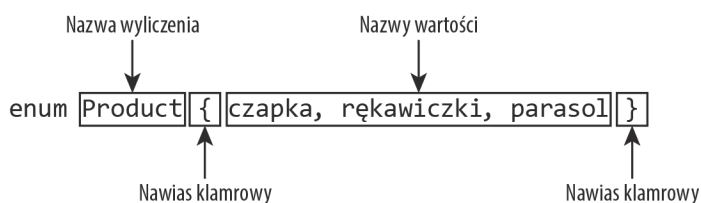
function writePrice(product: string, price: number): void {
    console.log(`Cena produktu ${product}: ${price.toFixed(2)} zł`);
}

enum Product { czapka, rękawiczki, parasol }

let products: [Product, number][] = [[Product.czapka, 100], [Product.rękawiczki, 75]];

products.forEach((prod: [Product, number]) => {
    switch (prod[0]) {
        case Product.czapka:
            writePrice("czapka", calculateTax(prod[1]));
            break;
        case Product.rękawiczki:
            writePrice("rękawiczki", calculateTax(prod[1]));
            break;
        case Product.parasol:
            writePrice("parasol", calculateTax(prod[1]));
            break;
    }
});
```

Wyliczenie jest definiowane za pomocą słowa kluczowego `enum`, po którym znajduje się nazwa i umieszczona w nawiasie klamrowym lista wartości, jak pokazałem na rysunku 9.3.



Rysunek 9.3. Definiowanie wyliczenia

Wartości wyliczenia są dostępne w postaci `<wyliczenie>.<wartość>`, więc wartość `czapka` zdefiniowana przez wyliczenie `Product` jest dostępna jako `Product.czapka`:

```
...
case Product.czapka:
...
```


Wyliczenie działa podobnie jak każdy inny typ. Omawiany przykład pokazuje wyliczenie `Product` użyte w krotce oraz w konstrukcji `switch`. Po skompilowaniu i uruchomieniu kodu przedstawionego na listingu 9.16 zostaną wygenerowane następujące dane wyjściowe:

```
Cena produktu czapka: 120.00 zł
Cena produktu rękawiczki: 90.00 zł
```

Sposób działania wyliczenia

Wyliczenie jest całkowicie implementowane przez kompilator TypeScriptu, opiera się na sprawdzeniu typu podczas kompilacji oraz na standardowych funkcjach JavaScriptu w trakcie działania aplikacji. Każdej wartości wyliczenia odpowiada wartość typu `number` przypisywana automatycznie przez kompilator i domyślnie rozpoczynająca się od zera. Dlatego też liczby przypisane elementom `czapka`, `rękawiczki` i `parasol` wyliczenia `Product` mają wartości odpowiednio 0, 1 i 2, jak pokazałem na listingu 9.17.

Listing 9.17. *Używanie wartości liczbowej wyliczenia w kodzie pliku `index.ts` w katalogu `src`*

```
function calculateTax(amount: number): number {
    return amount * 1.2;
}

function writePrice(product: string, price: number): void {
    console.log(`Cena produktu ${product}: ${price.toFixed(2)} zł`);
}

enum Product { czapka, rękawiczki, parasol }

[Product.czapka, Product.rękawiczki, Product.parasol].forEach(val => {
    console.log(`Wartość typu number: ${val}`);
});
```

Pogrubiony na listingu 9.17 fragment kodu przekazuje poszczególne wartości wyliczenia `Product` wywołaniu `console.log()`. Każda wartość wyliczenia jest liczbą, a kod tego listingu powoduje wygenerowanie następujących danych wyjściowych:

```
Wartość typu number: 0
Wartość typu number: 1
Wartość typu number: 2
```

Skoro wyliczenie jest implementowane za pomocą wartości JavaScriptu typu `number`, wyliczeniu można przypisać liczbę, która będzie później wyświetlona jako wartość typu `number`, jak pokazałem na listingu 9.18.

Listing 9.18. *Używanie wyliczenia i wartości liczbowej w kodzie pliku `index.ts` w katalogu `src`*

```
function calculateTax(amount: number): number {
    return amount * 1.2;
}

function writePrice(product: string, price: number): void {
```

```
    console.log(`Cena produktu ${product}: ${price.toFixed(2)} zł`);
  }
```

```
enum Product { czapka, rękawiczki, parasol }
```

```
let productValue: Product = 0;
let productName: string = Product[productValue];
console.log(`Wartość: ${productValue}, Nazwa: ${productName}`);
```

Kompilator wymusza sprawdzanie typu wyliczenia, co oznacza możliwość otrzymania komunikatu błędu w przypadku próby porównania wartości różnych wyliczeń, nawet jeśli mają tę samą wartość typu number. Wyliczenie oferuje składnię w stylu indeksu tablicy przeznaczoną do pobierania nazwy wartości, np.:

```
...
let productName: string = Product[productValue];
...
```

Wynikiem tej operacji jest ciąg tekstowy zawierający nazwę wartości wyliczenia, którą w omawianym przykładzie jest czapka. Kod przedstawiony na listingu 9.18 powoduje wygenerowanie następujących danych wyjściowych:

```
Wartość: 0, Nazwa: czapka
```

Używanie wartości wyliczenia

Domyślnie kompilator TypeScriptu rozpoczyna przypisywanie wartości liczbowych wyliczenia od zera i oblicza je przez inkrementację poprzedniej wartości. W przypadku wyliczenia Product przedstawionego na listingu 9.18 kompilator zaczyna od przypisania wartości 0 elementowi czapka, wartości 1 elementowi rękawiczki i wartości 2 elementowi parasol. Jeżeli chcesz poznać wartości przypisane elementom wyliczenia, wówczas możesz przeanalizować pliki deklaracji typu generowane przez kompilator, gdy ustawienie `declarations` ma wartość `true`. Jeżeli przeanalizujesz plik `index.d.ts` w katalogu `dist`, wówczas zobaczysz wartości obliczone przez kompilator dla wyliczenia Product.

```
...
declare enum Product {
  czapka = 0,
  rękawiczki = 1,
  parasol = 2
}
...
```

Wyliczenie może być zdefiniowane również z literałami i wówczas będą użyte podane wartości, jak pokazałem na listingu 9.19. To użyteczna możliwość w sytuacji, w której wyliczenie przedstawia rzeczywisty zbiór wartości.

Listing 9.19. Używanie stałej wartości wyliczenia w kodzie pliku `index.ts` w katalogu `src`

```
function calculateTax(amount: number): number {
  return amount * 1.2;
}
```

```
function writePrice(product: string, price: number): void {
    console.log(`Cena produktu ${product}: ${price.toFixed(2)} zł`);
}
```

```
enum Product { czapka, rękawiczki = 20, parasol }
```

```
let productValue: Product = 0;
let productName: string = Product[productValue];
console.log(`Wartość: ${productValue}, Nazwa: ${productName}`);
```

Elementowi rękawiczki wyliczenia przypisałem wartość 20. Kompilator wygeneruje pozostałe wartości niezbędne wyliczeniu. Po przeanalizowaniu zawartości pliku *index.d.ts* zobaczysz wyliczenia obliczone przez kompilator dla elementów czapka i parasol.

```
...
declare enum Product {
    czapka = 0,
    rękawiczki = 20,
    parasol = 21
}
...
```

Wartość poprzednia jest używana do wygenerowania wartości wyliczenia niezależnie od tego, czy została wybrana przez programistę, czy wygenerowana przez kompilator. W przypadku wyliczenia przedstawionego na listingu 9.19 kompilator użył wartości przypisanej elementowi rękawiczki do wygenerowania wartości dla elementu parasol. Kod przedstawiony na listingu 9.19 powoduje wygenerowanie następujących danych wyjściowych:

```
Wartość: 0, Nazwa: czapka
```

■ **Ostrzeżenie** Kompilator sprawdza poprzednią wartość jedynie podczas generowania wartości typu `number`, ale nie sprawdza, czy dana wartość została już użyta, co może prowadzić do powtarzających się wartości wyliczenia.

Kompilator będzie wykonywał proste wyrażenia dla wartości wyliczenia, jak pokazałem na listingu 9.20. Oznacza to, że wartości mogą być oparte na innych wartościach w tym samym wyliczeniu, w innym wyliczeniu bądź też na zupełnie innej wartości.

Listing 9.20. Używanie wyrażenia wyliczenia w kodzie pliku *index.ts* w katalogu *src*

```
function calculateTax(amount: number): number {
    return amount * 1.2;
}

function writePrice(product: string, price: number): void {
    console.log(`Cena produktu ${product}: ${price.toFixed(2)} zł`);
}

enum OtherEnum { First = 10, Two = 20 }
enum Product { czapka = OtherEnum.First + 1, rękawiczki = 20, parasol = czapka + rękawiczki }
```

```
let productValue: Product = 0;
let productName: string = Product[productValue];
console.log(`Wartość: ${productValue}, Nazwa: ${productName}`);
```

Wartość elementu czapka została obliczona na podstawie wyrażenia obejmującego wartość innego wyliczenia (OtherEnum) i operatora dodawania, natomiast wartością elementu parasol jest suma wartości elementów czapka i rękawiczki. Analiza pliku *index.d.ts* w katalogu *dist* pokazuje, że kompilator wykonał te wyrażenia podczas obliczania wartości elementów wyliczenia Product.

```
...
declare enum Product {
    czapka = 11,
    rękawiczki = 20,
    parasol = 31
}
...
```

Taka funkcjonalność jest użyteczna, ale należy zwrócić baczną uwagę, aby uniknąć przypadkowego utworzenia powtarzających się wartości lub nieoczekiwanych wyników. Najlepszym rozwiązaniem będzie pozostawienie kompilatorowi zadania, jakim jest generowanie wartości elementów wyliczenia, o ile to możliwe. Kod przedstawiony na listingu 9.20 powoduje wygenerowanie następujących danych wyjściowych:

```
Wartość: 0, Nazwa: undefined
```

Używanie wyliczenia w postaci ciągu tekstowego

Domyślna implementacja wyliczenia powoduje przedstawianie wartości poszczególnych elementów jako liczb. Kompilator ma możliwość użycia również wartości w postaci ciągu tekstowego, jak pokazałem na listingu 9.21.

-
- **Wskazówka** Wyliczenie może zawierać jednocześnie wartości typów string i number, choć takie rozwiązanie nie należy do często stosowanych.
-

Listing 9.21. Używanie ciągów tekstowych jako wartości wyliczenia w kodzie pliku *index.ts* w katalogu *src*

```
function calculateTax(amount: number): number {
    return amount * 1.2;
}

function writePrice(product: string, price: number): void {
    console.log(`Cena produktu ${product}: ${price.toFixed(2)} zł`);
}

enum OtherEnum { First = 10, Two = 20 }
enum Product { czapka = OtherEnum.First + 1, rękawiczki = 20, parasol = czapka + rękawiczki }
```

```
let productValue: Product = 0;
let productName: string = Product[productValue];
console.log(`Wartość: ${productValue}, Nazwa: ${productName}`);
```

```
enum City { Londyn = "Londyn", Paryż = "Paryż", NY = "Nowy Jork" }
console.log(`Miasto: ${City.Londyn}`);
```

Wartość w postaci ciągu tekstowego musi być dostarczona dla każdego elementu wyliczenia. Zaletą wartości tekstowych jest to, że są łatwiejsze do rozpoznania podczas procesu debugowania lub w plikach dzienników zdarzeń, jak pokazałem w danych wyjściowych wygenerowanych przez kod przedstawiony na listingu 9.21.

```
Wartość: 0, Nazwa: undefined
Miasto: Londyn
```

Ograniczenia typu wyliczeniowego

Wyliczenie może być użyteczne, ale istnieją pewne ograniczenia, ponieważ jest to funkcjonalność w pełni implementowana przez kompilator TypeScriptu, a następnie konwertowana na czysty kod JavaScriptu.

Ograniczenia związane ze sprawdzaniem typu

Kompilator doskonale radzi sobie ze sprawdzaniem typu dla wyliczenia, choć nie podejmuje żadnych kroków mających na celu zagwarantowanie użycia prawidłowej wartości typu `number`. W kodzie przedstawionym na listingu 9.21 wybrałem określone wartości dla elementów wyliczenia `Product`, więc następujące polecenie będzie problematyczne:

```
...
let productValue: Product = 0;
...
```

Kompilator nie chroni przed przypisaniem liczby zmiennej, której typ jest wyliczeniem, gdy ta liczba nie odpowiada jednej z wartości wyliczenia. Dlatego też dane wyjściowe kodu przedstawionego na listingu 9.21 zawierają wartość `undefined` — nie udało się znaleźć odpowiedniej nazwy elementu wyliczenia `Product` dla podanej wartości liczbowej. Ten sam problem pojawia się, gdy funkcja używa wyliczenia jako typu wyniku, ponieważ wówczas kompilator pozwoli jej na zwrot dowolnej wartości typu `number`.

■ **Wskazówka** Nie stanowi to problemu w przypadku wyliczenia zawierającego ciągi tekstowe, które w tle są implementowane inaczej i mogą mieć przypisane tylko wartości wyliczenia.

Ograniczenia wartownika typu

Podobny problem pojawia się podczas stosowania wartownika typu. Sprawdzanie typu odbywa się za pomocą słowa kluczowego `typeof` w JavaScriptcie, a skoro wyliczenie jest implementowane z wykorzystaniem wartości JavaScriptu `number`, słowo kluczowe `typeof` nie może być używane do rozróżniania między wartościami typów `enum` i `number`, jak pokazałem na listingu 9.22.

Listing 9.22. Używanie wartownika typu w pliku *index.ts* w katalogu *src*

```
function calculateTax(amount: number): number {
    return amount * 1.2;
}

function writePrice(product: string, price: number): void {
    console.log(`Cena produktu ${product}: ${price.toFixed(2)} zł`);
}

enum OtherEnum { First = 10, Two = 20 }
enum Product { czapka = OtherEnum.First + 1 , rękawiczki = 20, parasol = czapka + rękawiczki }

let productValue: Product = Product.czapka;
if (typeof productValue === "number") {
    console.log("Wartość jest typu number.");
}

let unionValue: number | Product = Product.czapka;
if (typeof unionValue === "number") {
    console.log("Wartość jest typu number.");
}
```

Po skompilowaniu i uruchomieniu kodu przedstawionego na listingu 9.22 zostaną wygenerowane następujące dane wyjściowe:

```
Wartość jest typu number.
Wartość jest typu number.
```

Używanie wyliczenia w postaci stałej

Kompilator TypeScriptu tworzy obiekt dostarczający implementację wyliczenia. W niektórych aplikacjach wpływ tego obiektu na wydajność działania może stanowić problem i dlatego trzeba będzie zastosować inne podejście.

■ **Wskazówka** Jest to funkcja zaawansowana, rzadko wymagana w większości projektów.

Aby pokazać, jak kompilator używa obiektu do implementacji wyliczenia, na listingu 9.23 przedstawiłem uproszczoną wersję kodu pliku *index.ts*, w której zostało zdefiniowane wyliczenie i znalazło się jedno polecenie przypisujące to wyliczenie zmiennej.

Listing 9.23. Uproszczenie kodu w pliku *index.ts* w katalogu *src*

```
enum Product { czapka, rękawiczki, parasol }
let productValue = Product.czapka;
```

Jeżeli chcesz zobaczyć, jak wyliczenie jest implementowane, przeanalizuj zawartość pliku *index.ts* w katalogu *dist*, a znajdziesz następujący fragment kodu:

```
...
var Product;
(function (Product) {
    Product[Product["czapka"] = 0] = "czapka";
```

```

    Product[Product["rękawiczki"] = 1] = "rękawiczki";
    Product[Product["parasol"] = 2] = "parasol";
  })(Product || (Product = {}));
  let productValue = Product.czapka;
  ...

```

Nie musisz rozumieć sposobu działania tego kodu. Najważniejsze jest tutaj utworzenie obiektu `Product` i jego użycie podczas przypisywania wartości zmiennej `productValue`.

Aby uniemożliwić kompilatorowi używanie obiektu podczas implementacji wyliczenia, można wykorzystać słowo kluczowe `const` w definicji wyliczenia znajdującej się w pliku TypeScriptu, jak pokazałem na listingu 9.24.

■ **Uwaga** Wyliczenie w postaci stałej jest znacznie bardziej ograniczone niż zwykle, a jego wszystkie wartości muszą być wyrażeniami stałych. Aby spełnić ten warunek, należy pozostawić zadanie przypisywania wartości kompilatorowi lub wyraźnie przypisywać je samodzielnie.

Listing 9.24. Definiowanie wyliczenia w postaci stałej w kodzie pliku *index.ts* w katalogu *src*

```

const enum Product { czapka, rękawiczki, parasol }
let productValue = Product.czapka;

```

Po skompilowaniu tego kodu kompilator będzie zmieniał na literal każde odwołanie do wyliczenia, co oznacza bezpośrednie używanie wartości liczbowych. Jeżeli po zakończeniu kompilacji przeanalizujesz zawartość pliku *index.d.ts* w katalogu *dist*, znajdziesz w nim następujący fragment:

```

...
let productValue = 0 /* czapka */;
...

```

Komentarz został dołączony przez kompilator i ma pomóc we wskazaniu relacji między wartością typu `number` a wyliczeniem. Obiekt, który wcześniej przedstawiał wyliczenie, nie jest już umieszczany w skompilowanym kodzie.

Wyliczenie w postaci stałej oferuje nieco większą wydajność działania, choć odbywa się to kosztem wyłączenia funkcjonalności w postaci wyszukiwania elementu za pomocą wartości, jak pokazałem na listingu 9.25.

Listing 9.25. Wyszukiwanie elementu wyliczenia za pomocą wartości w pliku *index.ts* w katalogu *src*

```

const enum Product { czapka, rękawiczki, parasol }
let productValue = Product.czapka;
let productName = Product[0];

```

Kod przedstawiony na listingu 9.25 powoduje wygenerowanie następujących danych wyjściowych:

```

src/index.ts(11,27): error TS2476: A const enum member can only be accessed using a string
↳ literal

```

Obiekt używany do przedstawienia zwykłego wyliczenia jest odpowiedzialny za dostarczanie funkcjonalności wyszukiwania, która będzie niedostępna dla wyliczenia w postaci stałej.

- **Wskazówka** Istnieje opcja kompilatora o nazwie `preserveConstEnums` nakazująca kompilatorowi wygenerowanie obiektu nawet w przypadku zdefiniowania wyliczenia w postaci stałej. Ta funkcjonalność jest dostępna jedynie dla celów debugowania i nie przywraca funkcji wyszukiwania.

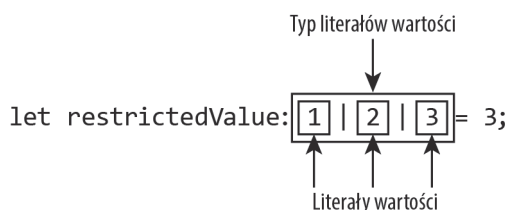
Używanie typu literału wartości

Typ literałów wartości określa pewien zbiór wartości i pozwala na stosowanie tylko tych wartości. W efekcie ten zbiór wartości jest traktowany jako oddzielny typ, co wprawdzie jest użyteczną funkcjonalnością, ale jednocześnie może powodować trudności, ponieważ zaciera granice między typami a wartościami. Tę funkcjonalność najłatwiej zrozumieć na przykładzie, jak pokazałem na listingu 9.26.

Listing 9.26. Używanie typu literału wartości w kodzie pliku `index.ts` w katalogu `src`

```
let restrictedValue: 1 | 2 | 3 = 3;
console.log(`Wartość: ${restrictedValue}`);
```

Typ literałów wartości przedstawia się podobnie do unii typów, przy czym używane są literały wartości zamiast typów danych, jak pokazałem na rysunku 9.4.



Rysunek 9.4. Typ literałów wartości

Typ literałów wartości na listingu 9.26 wskazuje kompilatorowi, że zmienna `restrictedValue` może otrzymać jedynie wartości 1, 2 lub 3. Kompilator wygeneruje komunikat błędu w przypadku próby przypisania innej wartości tej zmiennej, np. innej liczby, jak pokazałem na listingu 9.27.

Listing 9.27. Przypisywanie wartości innej niż wymieniona w typie literałów wartości w kodzie pliku `index.ts` w katalogu `src`

```
let restrictedValue: 1 | 2 | 3 = 100;
console.log(`Wartość: ${restrictedValue}`);
```

Kompilator ustala, że 100 nie jest jedną z dozwolonych wartości, więc generuje następujący komunikat błędu:

```
src/index.ts(1,5): error TS2322: Type '100' is not assignable to type '1 | 2 | 3'.
```

Połączenie wartości jest traktowane jako odmienny typ, a poszczególne połączenia literałów wartości są uznawane za odmienny typ, jak pokazałem na listingu 9.28. Jednak wartość jednego typu może być przypisana innemu typowi, o ile wartość jest jedną z dozwolonych.

Listing 9.28. Definiowanie drugiego typu literałów wartości w kodzie pliku *index.ts* w katalogu *src*

```
let restrictedValue: 1 | 2 | 3 = 1;

let secondValue: 1 | 10 | 100 = 1;

restrictedValue = secondValue;
secondValue = 100;
restrictedValue = secondValue;

console.log(`Wartość: ${restrictedValue}`);
```

Pierwsze polecenie przypisujące `restrictedValue` wartość `secondValue` jest dozwolone, ponieważ wartość `secondValue` jest jednym z literałów wartości `restrictedValue`. Natomiast drugie polecenie przypisania jest niedozwolone, ponieważ wartość wykracza poza dozwolony zbiór. Dlatego po skompilowaniu kodu zostaje wyświetlony następujący komunikat błędu:

```
src/index.ts(7,1): error TS2322: Type '100' is not assignable to type '1 | 2 | 3'
```

Używanie w funkcji typu literałów wartości

Typ literałów wartości okazuje się najbardziej użyteczny w funkcjach, ponieważ pozwala na ograniczenie parametrów lub wyników jedynie do zbioru określonych wartości, jak pokazałem na listingu 9.29.

Listing 9.29. Ograniczanie funkcji w kodzie pliku *index.ts* w katalogu *src*

```
function calculatePrice(quantity: 1 | 2, price: number): number {
    return quantity * price;
}

let total = calculatePrice(2, 19.99);
console.log(`Cena: ${total}`);
```

Parametr `quantity` funkcji będzie akceptował jedynie wartości 1 lub 2, a użycie jakiegokolwiek innej wartości — nawet liczbowej — spowoduje wygenerowanie błędu przez kompilator. Po skompilowaniu i uruchomieniu kodu przedstawionego na listingu 9.29 zostaną wygenerowane następujące dane wyjściowe:

```
Cena: 39.98
```

Łączenie typów wartości w typie literałów wartości

Typ literałów wartości może się składać z dowolnego połączenia wartości, które można wyrazić literalnie (dotyczy to również wyliczeń). Na listingu 9.30 przedstawiłem połączenie różnych wartości w typie literałów wartości.

Listing 9.30. Łączenie wartości w typie literałów wartości w pliku *index.ts* w katalogu *src*

```
function calculatePrice(quantity: 1 | 2, price: number): number {
    return quantity * price;
```

```

}

let total = calculatePrice(2, 19.99);
console.log(`Cena: ${total}`);

function getRandomValue(): 1 | 2 | 3 | 4 {
    return Math.floor(Math.random() * 4) + 1 as 1 | 2 | 3 | 4;
}

enum City { Londyn = "LON", Paryż = "PAR", Chicago = "CHI" }

function getMixedValue(): 1 | "Witaj" | true | City.Londyn {
    switch (getRandomValue()) {
        case 1:
            return 1;
        case 2:
            return "Witaj";
        case 3:
            return true;
        case 4:
            return City.Londyn;
    }
}

console.log(`Wartość: ${getMixedValue()}`);

```

Funkcja `getRandomValue()` zwraca jedną z czterech wartości, które są wykorzystywane przez funkcję `getMixedValue()` do wygenerowania wyniku działania. Funkcja `getMixedValue()` pokazuje, jak typ literałów wartości łączy ze sobą wartości zwykle uznawane za oddzielne typy — używając wartości typu `number`, `string`, `boolean` i wyliczenia. Po skompilowaniu i uruchomieniu kodu przedstawionego na listingu 9.30 zostaną wygenerowane następujące dane wyjściowe (możesz otrzymać nieco inne, ponieważ funkcja `getMixedValues()` używa generatora liczb losowych):

```

Cena: 39.98
Wartość: true

```

- **Wskazówka** Typ literału wartości może być stosowany w unii typów, tworząc tym samym połączenie pozwalające na powiązanie wartości jednego typu z dozwoloną wartością innego typu. Na przykład unia typów `string | true | 3` może otrzymać dowolną wartość w postaci ciągu tekstowego, wartość boolowską lub liczbę 3.
-

Nadpisywanie za pomocą typu literałów wartości

W rozdziale 8. wyjaśniłem, jak relacja między typem parametru a wynikiem działania funkcji może zostać wyrażona poprzez nadpisanie typu, co ogranicza efekt stosowania unii typów. Nadpisywanie typu można zastosować również dla typów literałów wartości, jak pokazałem na listingu 9.31, co w praktyce dostarcza unie używane do przedstawienia poszczególnych wartości.

Listing 9.31. *Nadpisywanie typu literalów wartości w kodzie pliku `index.ts` w katalogu `src`*

```

function calculatePrice(quantity: 1 | 2, price: number): number {
    return quantity * price;
}

let total = calculatePrice(2, 19.99);
console.log(`Cena: ${total}`);

function getRandomValue(): 1 | 2 | 3 | 4 {
    return Math.floor(Math.random() * 4) + 1 as 1 | 2 | 3 | 4;
}

enum City { Londyn = "LON", Paryż = "PAR", Chicago = "CHI" }

function getMixedValue(input: 1): 1;
function getMixedValue(input: 2 | 3): "Witaj" | true;
function getMixedValue(input: 4): City.Londyn;
function getMixedValue(input: number): 1 | "Witaj" | true | City.Londyn {
    switch (input) {
        case 1:
            return 1;
        case 2:
            return "Witaj";
        case 3:
            return true;
        case 4:
            return City.Londyn;
        default:
            return City.Londyn;
    }
}

let first = getMixedValue(1);
let second = getMixedValue(2);
let third = getMixedValue(4);
console.log(`${first}, ${second}, ${third}`);

```

Każde mapowanie tworzy relację między parametrem a wynikiem, którą można wyrazić za pomocą jednej lub więcej wartości. Kompilator TypeScriptu potrafi obsługiwać nadpisywanie i tym samym ustalać typ zmiennych `first`, `second` i `third`, co możesz sprawdzić poprzez analizę zawartości pliku `index.d.ts` w katalogu `dist`.

```

...
declare let first: 1;
declare let second: true | "Witaj";
declare let third: City.Londyn;
...

```

Wprawdzie jest to funkcjonalność niepotrzebna w większości projektów, ale zdecydowałem się na jej przedstawienie tutaj, aby pokazać, że typy literalów wartości są obsługiwane tak samo jak zwykłe typy. Poza tym to doskonale pokazuje wewnętrzny sposób działania kompilatora TypeScriptu. Kod przedstawiony na listingu 9.31 powoduje wygenerowanie następujących danych wyjściowych:

```

Cena: 39.98
1, Witaj, LON

```

Używanie szablonów typu literałów tekstowych

Typy literałów tekstowych można stosować w połączeniu z funkcjonalnością szablonów ciągów tekstowych w JavaScriptcie w celu tworzenia szablonów ciągów tekstowych akceptujących jedynie określone wartości. Może to być zwięzły sposób na wyrażenie skomplikowanych połączeń wartości. W kodzie na listingu 9.32 pokazałem przykład utworzenia szablonu ciągu tekstowego, który wykorzystuje typ literału wartości.

Listing 9.32. Przykład użycia typu literału tekstowego w szablonie ciągu tekstowego w kodzie pliku *index.ts* w katalogu *src*

```
function calculatePrice(quantity: 1 | 2, price: number): number {
    return quantity * price;
}

let total = calculatePrice(2, 19.99);
console.log(`Cena: ${total}`);

function getRandomValue(): 1 | 2 | 3 | 4 {
    return Math.floor(Math.random() * 4) + 1 as 1 | 2 | 3 | 4;
}

function getCityString(city: "Londyn" | "Paryż" | "Chicago")
    : `Miasto: ${"Londyn" | "Paryż" | "Chicago"}` {
    return `Miasto: ${city}` as `Miasto: ${"Londyn" | "Paryż" | "Chicago"}`;
}

let str = getCityString("Londyn");
console.log(str);
```

Funkcja `getCityString()` definiuje parametr, którego wartość jest ograniczona do trzech ciągów tekstowych typu literału wartości. Wynik działania tej funkcji jest wyrażony za pomocą szablonu ciągu tekstowego używającego typu literału wartości, np.:

```
...
`Miasto: ${"Londyn" | "Paryż" | "Chicago"}`
...
```

Aby zobaczyć, dlaczego takie rozwiązanie jest użyteczne, zajrzyj do pliku *index.d.ts* w katalogu *dist* i sprawdź, jak kompilator TypeScriptu zdefiniował typ zmiennej `str`:

```
...
declare let str: "Miasto: Londyn" | "Miasto: Paryż" | "Miasto: Chicago";
...
```

Kompilator wykorzystał typ literału wartości w celu rozbudowy szablonu ciągu tekstowego na postać pełnego zbioru ciągów tekstowych, które mogą być przypisywane zmiennej `str`. Kod przedstawiony na listingu 9.32 powoduje wygenerowanie następujących danych wyjściowych:

```
Cena: 39.98
Miasto: Londyn
```

Używanie aliasu typu

Aby uniknąć powtórzeń, TypeScript oferuje funkcję aliasu typu pozwalającą na tworzenie własnych połączeń typów i przypisywanie im nazw, do których później można się odwoływać, gdy zajdzie potrzeba. Przykład takiego rozwiązania pokazałem na listingu 9.33.

Listing 9.33. Używanie aliasu typu w kodzie pliku *index.ts* w katalogu *src*

```
function calculatePrice(quantity: 1 | 2, price: number): number {
    return quantity * price;
}

let total = calculatePrice(2, 19.99);
console.log(`Cena: ${total}`);

type numVals = 1 | 2 | 3 | 4;

function getRandomValue(): numVals {
    return Math.floor(Math.random() * 4) + 1 as numVals;
}

type cities = "Londyn" | "Paryż" | "Chicago";
type cityResponse = `Miasto: ${ cities }`;

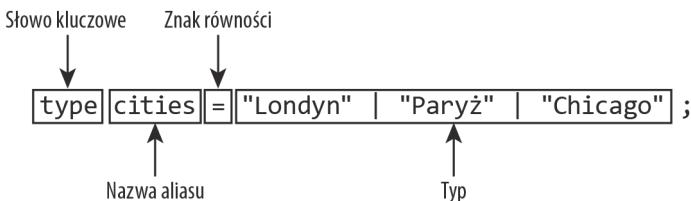
function getCityString(city: cities): cityResponse {
    return `Miasto: ${city}` as cityResponse;
}

let str = getCityString("Londyn");
console.log(str);
```

Zastosowanie aliasu typu pozwala uporządkować kod TypeScriptu przez wyeliminowanie powtórzeń. Zamiast definiować ten sam zbiór miast dla parametru funkcji `getCityString()` i wyniku jej działania, można utworzyć alias typu, który później będzie używany w parametrze funkcji i szablonie ciągu tekstowego.

```
...
type cities = "Londyn" | "Paryż" | "Chicago";
type cityResponse = `City: ${ cities }`;
...
```

Alias typu jest definiowany przez słowo kluczowe `type`, po którym znajduje się nazwa aliasu, znak równości i typ, jak pokazałem na rysunku 9.5.



Rysunek 9.5. Definiowanie aliasu typu

Nazwa przypisana aliasowi jest używana zamiast pełnej definicji typu. Stosowanie aliasu typu pozwala na znacznie łatwiejsze odwoływanie się do skomplikowanych typów lub połączenia typów. Alias jednak nie zmienia sposobu, w jaki kompilator TypeScriptu obsługuje dany typ. Dlatego też alias może być w zwykły sposób używany w adnotacjach typów lub w asercjach. Po skompilowaniu i uruchomieniu kodu przedstawionego na listingu 9.33 zostaną wygenerowane następujące dane wyjściowe:

Cena: 39.98
Miasto: Londyn

Podsumowanie

W tym rozdziale wyjaśniłem, jak używać TypeScriptu z tablicami, ponadto przedstawiłem krotki i wyliczenia, które są implementowane przez kompilator TypeScriptu. Dowiedziałeś się również, jak definiować typ literałów wartości i jak używać aliasów do spójnego opisywania typów. W następnym rozdziale przejdę do omówienia funkcjonalności, które TypeScript oferuje do pracy z obiektami.

ROZDZIAŁ 10.



Praca z obiektami

W tym rozdziale zamierzam pokazać, w jaki sposób przebiega praca z obiektami w TypeScriptie. Jak już wyjaśniłem w rozdziałach 3. i 4., JavaScript oferuje elastyczne podejście w zakresie pracy z obiektami, natomiast TypeScript dąży do zachowania równowagi między ochroną użytkownika przed najczęściej popełnianymi błędami a umożliwieniem stosowania najużyteczniejszych funkcjonalności. Ten temat będę kontynuował w rozdziale 11., w którym przedstawię oferowaną przez TypeScript obsługę klas. Streszczenie materiału przedstawionego w rozdziale znajdziesz w tabeli 10.1.

Tabela 10.1. Streszczenie materiału przedstawionego w rozdziale

Problem	Rozwiązanie	Listing
Opisanie obiektu w kompilatorze TypeScriptu	Użyj kształtu typu	Od 4 do 6 i 8
Opisanie nieregularnych kształtów typu	Użyj właściwości opcjonalnych	7, 9 i 10
Użycie tego samego kształtu do opisanie wielu obiektów	Użyj aliasu typu	11
Zapobieganie błędom kompilatora, gdy typ zawiera nadzbiór właściwości w kształcie	Włącz opcję kompilatora <code>suppressExcessPropertyErrors</code>	12 i 13
Łączenie kształtów typu	Użyj unii typów lub złączeń typów	14, 15, od 19 do 25
Używanie wartownika typu dla typów obiektu	Za pomocą słowa kluczowego <code>in</code> sprawdź właściwości zdefiniowane przez obiekt	16 i 17
Ponowne użycie wartownika typu	Zdefiniuj funkcję predykatu	18

W tabeli 10.2 wymieniłem opcje kompilatora TypeScriptu użyte w rozdziale.

Tabela 10.2. Opcje kompilatora TypeScriptu użyte w rozdziale

Opcja	Opis
target	Ta opcja określa docelową wersję języka JavaScript, dla której kompilator wygeneruje kod
outDir	Ta opcja określa katalog, w którym zostaną umieszczone wygenerowane pliki JavaScriptu
rootDir	Ta opcja określa katalog główny używany przez kompilator do wyszukiwania plików TypeScriptu
declaration	Ta opcja generuje pliki deklaracji typu, które pomagają zrozumieć, w jaki sposób zostały ustalone typy dla kodu JavaScriptu. Wspomniane pliki zostaną szczegółowo omówione w rozdziale 14.
strictNullChecks	Ta opcja uniemożliwia używanie null i undefined jako wartości dla innych typów
suppressExcessPropertyErrors	Ta opcja uniemożliwia kompilatorowi generowanie błędów w przypadku obiektów definiujących właściwości, które nie są w określonym kształcie

Przygotowanie projektu

W rozdziale będę kontynuował używanie projektu *types* utworzonego w rozdziale 7. i uaktualnionego w kolejnych. Aby przygotować się do tego rozdziału, należy zastąpić zawartość pliku *index.ts* w katalogu *src* kodem przedstawionym na listingu 10.1.

Listing 10.1. Nowa zawartość pliku *index.ts* w katalogu *src*

```
let hat = { name: "czapka", price: 100 };
let gloves = { name: "rękawiczki", price: 75 };

let products = [hat, gloves];

products.forEach(prod => console.log(`${prod.name}: ${prod.price}`));
```

Wyzeruj konfigurację kompilatora przez zastąpienie zawartości pliku *tsconfig.json* kodem przedstawionym na listingu 10.2.

Listing 10.2. Konfigurowanie kompilatora w pliku *tsconfig.json* w katalogu *types*

```
{
  "compilerOptions": {
    "target": "es2020",
    "outDir": "./dist",
    "rootDir": "./src",
    "declaration": true,
    // "strictNullChecks": true,
  }
}
```


Konfiguracja kompilatora zawiera ustawienie `declaration` oznaczające, że kompilator będzie oprócz plików JavaScriptu tworzył również pliki deklaracji typów. Rzeczywiste przeznaczenie tych plików przedstawię w rozdziale 14., natomiast w tym rozdziale będą one używane do wyjaśnienia, jak kompilator radzi sobie z typami danych.

Otwórz nowe okno powłoki, przejdź do katalogu `types`, a następnie z jego poziomu wydaj polecenie przedstawione na listingu 10.3, uruchamiające kompilator TypeScriptu w trybie automatycznego kompilowania kodu po wykryciu zmiany w dowolnym pliku i wykonywania kodu po jego skompilowaniu.

■ **Wskazówka** Przykładowy projekt dla tego rozdziału, a także projekty dla pozostałych rozdziałów książki zostały umieszczone w archiwum dostępnym pod adresem <https://ftp.helion.pl/przyklady/tys4o2.zip>.

Listing 10.3. Uruchamianie kompilatora TypeScriptu

```
$ npm start
```

Kompilator przeprowadzi kompilację kodu w pliku `index.ts`, wykona wygenerowany kod JavaScriptu, a następnie przejdzie do trybu monitorowania i wyświetli następujące dane wyjściowe:

```
7:10:34 AM - Starting compilation in watch mode...
7:10:35 AM - Found 0 errors. Watching for file changes.
czapka: 100
rękawiczki: 75
```

Praca z obiektami

Obiekt JavaScriptu to kolekcja właściwości, które mogą być utworzone za pomocą składni literału, funkcji konstruktora lub klas. Niezależnie od sposobu utworzenia, gotowy obiekt może być modyfikowany, co oznacza możliwość dodawania lub usuwania właściwości, a także otrzymywania wartości różnych typów. Aby dostarczyć obiektowi funkcje typu, TypeScript koncentruje się na tzw. kształcie obiektu, czyli połączeniu nazw właściwości i typów.

Kompilator JavaScriptu próbuje zagwarantować spójny sposób używania obiektów przez sprawdzenie wspólnych cech charakterystycznych kształtu. Najlepszym sposobem na zobaczenie, jak to działa, jest analiza plików deklaracji wygenerowanych przez kompilator po włączeniu opcji `declarations`. Jeżeli przeanalizujesz plik `index.d.ts` w katalogu `dist`, zobaczysz, że kompilator użył kształtu każdego obiektu zdefiniowanego na listingu 10.1 jako jego typu, jak pokazałem w kolejnym fragmencie kodu:

```
declare let hat:      { name: string; price: number; };
declare let gloves:  { name: string; price: number; };
declare let products: { name: string; price: number; }[];
```

Zawartość pliku deklaracji sformatowałem w taki sposób, aby można było łatwiej dostrzec, jak kompilator zidentyfikował typy poszczególnych obiektów na podstawie ich kształtu. Po umieszczeniu obiektów w tablicy kompilator używa ich kształtów do zdefiniowania typu tablicy do dopasowania.

Wprowadzić takie podejście może nie wydawać się użyteczne, ale chroni przed wieloma najczęściej popełnianymi błędami. Na listingu 10.4 pokazałem przykład dodania obiektu z innym kształtem.

Listing 10.4. Dodawanie obiektu w kodzie pliku *index.ts* w katalogu *src*

```
let hat = { name: "czapka", price: 100 };
let gloves = { name: "rękawiczki", price: 75 };
let umbrella = { name: "parasol" };

let products = [hat, gloves, umbrella];

products.forEach(prod => console.log(`${prod.name}: ${prod.price}`));
```

Mimo że obiekty przedstawione na listingu 10.1 zostały zdefiniowane za pomocą składni literału, kompilator TypeScriptu i tak generuje komunikat ostrzeżenia w przypadku niespójnego używania obiektów. Obiekt *umbrella* nie ma właściwości *price*, więc po skompilowaniu pliku kompilator spowoduje wygenerowanie następującego błędu:

```
src/index.ts(9,60): error TS2339: Property 'price' does not exist on type '{ name: string; }'.
```

Funkcja strzałki użyta z metodą *forEach()* odczytuje właściwość *price* nieistniejącą we wszystkich obiektach tablicy *products*, co prowadzi do błędu. W omawianym przykładzie kompilator prawidłowo określa kształt obiektów, który można sprawdzić w pliku *index.d.ts* w katalogu *dist*:

```
declare let hat:      { name: string; price: number; };
declare let gloves:  { name: string; price: number; };
declare let umbrella: { name: string; };
declare let products: { name: string; }[];
```

Zwróć uwagę na zmianę typu tablicy *products*. Gdy obiekty o różnych kształtach są używane razem, np. w tablicy, wówczas kompilator tworzy typ z właściwościami istniejącymi we wszystkich obiektach, ponieważ praca z tylko tymi właściwościami jest uznawana za bezpieczną. W omawianym przykładzie jedyną właściwością zdefiniowaną we wszystkich obiektach tablicy jest właściwość *name* typu *string* i dlatego kompilator generuje komunikat błędu podczas próby odczytania właściwości *price*.

Używanie adnotacji kształtu typu obiektu

W przypadku literału obiektu kompilator TypeScriptu ustala typ poszczególnych właściwości na podstawie przypisanej jej wartości. Typy mogą być również określone jawnie za pomocą adnotacji typu, które są stosowane dla poszczególnych właściwości, jak pokazałem na listingu 10.5.

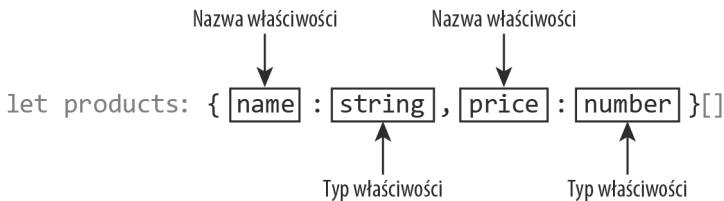
Listing 10.5. Używanie adnotacji kształtu typu obiektu w kodzie pliku *index.ts* w katalogu *src*

```
let hat = { name: "czapka", price: 100 };
let gloves = { name: "rękawiczki", price: 75 };
let umbrella = { name: "parasol" };

let products: { name: string, price: number }[] = [hat, gloves, umbrella];

products.forEach(prod => console.log(`${prod.name}: ${prod.price}`));
```

Adnotacja typu ogranicza zawartość tablicy `products` do obiektów zawierających właściwości o nazwach `name` i `price`, którym zostały przypisane wartości typu odpowiednio `string` i `number`, jak pokazałem na rysunku 10.1.

**Rysunek 10.1.** Typ kształtu obiektu

Kompilator nadal generuje błąd dla kodu przedstawionego na listingu 10.5, ale obecnie problem polega na tym, że obiekt `umbrella` jest niezgodny z kształtem wskazywanym przez adnotację typu dla tablicy `products`, a sam komunikat błędu zawiera znacznie użyteczniejszy opis problemu:

```
src/index.ts(5,64): error TS2741: Property 'price' is missing in type '{ name: string; }' but required in type '{ name: string; price: number; }'.
```

Dopasowanie kształtu typu obiektu

Aby dopasować typ, obiekt musi definiować wszystkie właściwości wymienione w jego kształcie. Kompilator nadal dopasuje obiekt zawierający właściwości dodatkowe, które nie zostały zdefiniowane przez kształt typu, jak pokazałem na listingu 10.6.

Listing 10.6. Dodawanie właściwości w kodzie pliku *index.ts* w katalogu *src*

```
let hat = { name: "czapka", price: 100 };
let gloves = { name: "rękawiczki", price: 75 };
let umbrella = { name: "parasol", price: 30, waterproof: true };

let products: { name: string, price?: number }[] = [hat, gloves, umbrella];

products.forEach(prod => console.log(`${prod.name}: ${prod.price}`));
```

Nowa właściwość pozwala na dopasowanie obiektu `umbrella` do kształtu typu tablicy, gdyż obiekt definiuje właściwości `name` i `price`. Właściwość `waterproof` jest ignorowana, nie stanowi bowiem fragmentu kształtu typu. Po skompilowaniu i uruchomieniu kodu przedstawionego na listingu 10.6 zostaną wygenerowane następujące dane wyjściowe:

```
czapka: 100
rękawiczki: 75
parasol: 30
```

Zwróć uwagę na pewien fakt: adnotacje typu nie są wymagane do wskazywania, że poszczególne obiekty mają określony kształt. Kompilator TypeScriptu automatycznie ustala, czy obiekt jest zgodny z kształtem, co odbywa się przez analizę jego właściwości i wartości.

Używanie właściwości opcjonalnych dla nieregularnych kształtów

Właściwości opcjonalne zapewniają większą elastyczność kształtu typu i pozwalają na dopasowanie obiektów niezawierających tych właściwości, jak pokazałem na listingu 10.7. Może to mieć ważne znaczenie podczas pracy ze zbiorem obiektów, które nie współdzielą tego samego kształtu, ale zachodzi potrzeba zastosowania właściwości, gdy jest ona dostępna.

Listing 10.7. Używanie właściwości opcjonalnej w kodzie pliku *index.ts* w katalogu *src*

```
let hat = { name: "czapka", price: 100 };
let gloves = { name: "rękawiczki", price: 75 };
let umbrella = { name: "parasol", price: 30, waterproof: true };

let products: { name: string, price?: number, waterproof?: boolean }[]
    = [hat, gloves, umbrella];

products.forEach(prod =>
    console.log(`${prod.name}: ${prod.price} nieprzemakalność: ${ prod.waterproof }`));
```

Właściwość opcjonalna jest definiowana za pomocą tej samej składni co opcjonalny parametr funkcji. Oznacza to umieszczenie znaku zapytania po nazwie właściwości, jak pokazałem na rysunku 10.2.

Właściwość opcjonalna
↓

```
let products: { name: string, price?: number, waterproof?: boolean }[]
```

Rysunek 10.2. Właściwość opcjonalna w kształcie typu obiektu

Typ kształtu z właściwościami opcjonalnymi ma możliwość dopasowania obiektów niedefiniujących tych właściwości, o ile zostały zdefiniowane właściwości wymagane. Gdy jest używana właściwość opcjonalna, np. w funkcji `forEach()` na listingu 10.7, wówczas wartością właściwości opcjonalnej będzie wartość `undefined` lub zdefiniowana przez obiekt, jak widać na przykładowych danych wyjściowych wygenerowanych po skompilowaniu i wykonaniu omawianego fragmentu kodu:

```
czapka: 100 nieprzemakalność: undefined
rękawiczki: 75 nieprzemakalność: undefined
parasol: 30 nieprzemakalność: true
```

Obiekty `hat` i `gloves` nie definiują właściwości opcjonalnej `waterproof`, więc wartością otrzymaną w funkcji `forEach()` jest `undefined`. Obiekt `umbrella` definiuje tę właściwość, więc zostaje wyświetlona jej wartość.

Dołączanie metod w kształcie typu

Typ kształtu może zawierać metody i właściwości, co zapewnia większą kontrolę nad sposobem dopasowywania obiektów przez typ, jak pokazałem na listingu 10.8.

Listing 10.8. Dołączenie metody w kształcie typu w kodzie pliku `index.ts` w katalogu `src`

```
enum Feature { Waterproof, Insulated }

let hat = { name: "czapka", price: 100 };
let gloves = { name: "rękawiczki", price: 75 };
let umbrella = { name: "parasol", price: 30,
  hasFeature: (feature) => feature === Feature.Waterproof };

let products: { name: string, price?: number,
  hasFeature?(Feature): boolean }[]
  = [hat, gloves, umbrella];

products.forEach(prod => console.log(`${prod.name}: ${prod.price} `
  + `nieprzemakalność: ${prod.hasFeature(Feature.Waterproof)}`));
```

Adnotacja typu dla tablicy `products` zawiera właściwość opcjonalną o nazwie `hasFeature` przedstawiającą metodę. Właściwość metody jest podobna do zwykłej właściwości, przy czym zawiera nawias z opisem typów parametrów. Po nawiasie znajduje się dwukropek i typ wyniku, jak pokazałem na rysunku 10.3.

```
let products: { name: string, price?: number,
  hasFeature?(Feature): boolean }[]
```

Nazwa metody
Typ parametru
Typ wyniku

Rysunek 10.3. Metoda w kształcie typu

Metoda umieszczona w kształcie typu na listingu 10.8 określa metodę o nazwie `hasFeature()` z jednym parametrem, którym musi być wartość typu wyliczeniowego `Feature` (również zdefiniowanego na listingu 10.8), i zwracającą wynik typu `boolean`.

-
- **Wskazówka** Metody w kształcie typu nie muszą być opcjonalne, ale jeśli są, jak w przykładzie przedstawionym na listingu 10.8, to znak zapytania należy umieścić po nazwie metody, a przed nawiasem zawierającym typy parametrów.
-

Obiekt `umbrella` definiuje metodę `hasFeature()` z odpowiednimi typami. Skoro ta metoda jest opcjonalna, obiekty `hat` i `gloves` również zostaną dopasowane przez kształt typu. Podobnie jak w przypadku zwykłych właściwości, metody opcjonalne otrzymują wartość

undefined, jeśli nie zostały zdefiniowane w obiekcie. Dlatego też przedstawiony na listingu 10.8 kod po skompilowaniu i uruchomieniu powoduje wygenerowanie następującego komunikatu błędu:

```
C:\types\dist\index.js:12  + `nieprzemakalność:
${prod.hasFeature(Feature.Waterproof)}`));
TypeError: prod.hasFeature is not a function
```

Podobnie jak w przypadku zwykłej właściwości, przed wywołaniem metody trzeba zagwarantować, że została ona zaimplementowana.

Wymuszenie ścisłego sprawdzania metod

Aby chronić się przed błędami takimi jak przedstawiony w poprzedniej sekcji, kompilator TypeScriptu może zgłaszać błąd, gdy metoda opcjonalna wskazana przez kształt typu będzie użyta bez sprawdzenia pod kątem wartości undefined. Takie sprawdzenie zostaje włączone za pomocą ustawienia strictNullChecks, z którego już korzystałem we wcześniejszych rozdziałach. Zmień konfigurację kompilatora i włącz ustawienie przedstawione pogrubioną czcionką na listingu 10.9.

Listing 10.9. Konfigurowanie kompilatora w pliku tsconfig.json w katalogu types

```
{
  "compilerOptions": {
    "target": "es2020",
    "outDir": "./dist",
    "rootDir": "./src",
    "declaration": true,
    "strictNullChecks": true
  }
}
```

Po zapisaniu pliku konfiguracyjnego kompilator przeprowadzi ponowną kompilację projektu i wygeneruje następujący komunikat błędu:

```
src/index.ts(13,22): error TS2722: Cannot invoke an object which is possibly 'undefined'.
```

Ten błąd uniemożliwia użycie metod opcjonalnych, dopóki nie zostanie przeprowadzona operacja sprawdzenia mająca na celu zagwarantowanie, że dana metoda istnieje w obiekcie. Spójrz na przykład przedstawiony na listingu 10.10.

Listing 10.10. Sprawdzanie metody opcjonalnej w kodzie pliku index.ts w katalogu src

```
enum Feature { Waterproof, Insulated }

let hat = { name: "czapka", price: 100 };
let gloves = { name: "rękawiczki", price: 75 };
let umbrella = { name: "parasol", price: 30,
  hasFeature: (feature) => feature === Feature.Waterproof };

let products: { name: string, price?: number, hasFeature?(Feature): boolean }[]
  = [hat, gloves, umbrella];
```

```
products.forEach(prod => console.log(`${prod.name}: ${prod.price} `
  + `${ prod.hasFeature ? prod.hasFeature(Feature.Waterproof) : "false" }`));
```

Metoda `hasFeature()` jest wywoływana tylko wtedy, gdy została zdefiniowana. Po skompilowaniu i uruchomieniu kodu przedstawionego na listingu 10.10 zostaną wygenerowane następujące dane wyjściowe:

```
czapka: 100 false
rękawiczki: 75 false
parasol: 30 true
```

Używanie aliasu typu dla kształtu typu

Alias typu może być używany do nadania nazwy określonego kształtowi, co pomaga w spójnym odwoływaniu się do tego kształtu w kodzie, jak pokazałem w przykładzie na listingu 10.11.

Listing 10.11. Używanie aliasu typu dla kształtu typu w kodzie pliku `index.ts` w katalogu `src`

```
enum Feature { Waterproof, Insulated }
```

```
type Product = {
  name: string,
  price?: number,
  hasFeature?(Feature): boolean
};
```

```
let hat = { name: "czapka", price: 100 };
let gloves = { name: "rękawiczki", price: 75 };
let umbrella = { name: "parasol", price: 30,
  hasFeature: (feature) => feature === Feature.Waterproof };
```

```
let products: Product[] = [hat, gloves, umbrella];
```

```
products.forEach(prod => console.log(`${prod.name}: ${prod.price} `
  + `${ prod.hasFeature ? prod.hasFeature(Feature.Waterproof) : "false" }`));
```

Alias powoduje przypisanie kształtowi nazwy, która następnie może być stosowana w adnotacjach typu. Na omawianym listingu alias o nazwie `Product` został utworzony jako typ dla tablicy. Używanie aliasu nie powoduje zmiany danych wyjściowych kodu po jego skompilowaniu i wykonaniu:

```
czapka: 100 false
rękawiczki: 75 false
parasol: 30 true
```

Radzenie sobie z nadmiarem właściwości

Kompilator TypeScriptu świetnie radzi sobie z określaniem typów, co oznacza, że często można pomijać adnotacje typu. Jednak zdarzają się sytuacje, w których dostarczenie kompilatorowi informacji o typach może zmienić sposób jego zachowania, jak pokazałem na listingu 10.12.

Listing 10.12. Definiowanie obiektów w kodzie pliku *index.ts* w katalogu *src*

```
enum Feature { Waterproof, Insulated }

type Product = {
  name: string,
  price?: number,
  hasFeature?(Feature): boolean
};

let hat = { name: "czapka", price: 100 };
let gloves = { name: "rękawiczki", price: 75 };
let umbrella = { name: "parasol", price: 30,
  hasFeature: (feature) => feature === Feature.Waterproof };

let mirrorShades = { name: "okulary", price: 54, finish: "lustrzane"};
let darkShades: Product = { name: "okulary", price: 54, finish: "matowe"};

let products: Product[] = [hat, gloves, umbrella, mirrorShades, darkShades];

products.forEach(prod => console.log(`${prod.name}: ${prod.price} `
  + `${ prod.hasFeature ? prod.hasFeature(Feature.Waterproof) : "false" }`));
```

Po skompilowaniu tego kodu kompilator wyświetli następujący komunikat błędu:

```
src/index.ts(16,60): error TS2322: Type '{ name: string; price: number; finish: string; }'
is not assignable to type 'Product'
  Object literal may only specify known properties, and 'finish' does not exist in type
  'Product'.
```

Kompilator odmiennie traktuje obiekty *mirrorShades* i *darkShades*, mimo że mają one ten sam kształt. Kompilator zgłasza błąd, gdy literal obiektu z adnotacją typu definiuje właściwości dodatkowe, ponieważ to może być błędem. W omawianym przykładzie obiekt *darkShades* ma adnotację typu *Product*. Właściwość *finish* nie jest częścią kształtu *Product* i stanowi *właściwość nadmiarową*, więc kompilator powoduje wygenerowanie komunikatu błędu. Właściwość nadmiarowa nie prowadzi do błędu w przypadku zdefiniowania obiektu bez adnotacji typu, co oznacza możliwość użycia obiektu *darkShades* jako *Product*.

Przedstawionego tutaj błędu można uniknąć przez usunięcie właściwości nadmiarowej lub adnotacji typu. Ja wolę jednak całkowicie wyłączyć sprawdzanie pod kątem właściwości nadmiarowych, co uważam za nieintuicyjne. W konfiguracji kompilatora wprowadź zmiany przedstawione na listingu 10.13, a następnie zapisz plik.

Listing 10.13. Konfigurowanie kompilatora w pliku *tsconfig.json* w katalogu *types*

```
{
  "compilerOptions": {
    "target": "es2020",
    "outDir": "./dist",
    "rootDir": "./src",
    "declaration": true,
    "strictNullChecks": true,
    "suppressExcessPropertyErrors": true
  }
}
```


Jeśli właściwości `suppressExcessPropertyErrors` została przypisana wartość `true`, kompilator nie będzie generował błędów w sytuacji, w której literal obiektu zdefiniuje właściwości niebędące częścią typu zadeklarowanego przez adnotację. Po zapisaniu zmodyfikowanego pliku konfiguracyjnego kompilatora kod zostanie skompilowany i uruchomiony, co spowoduje wygenerowanie następujących danych wyjściowych:

```
czapka: 100 false
rękawiczki: 75 false
parasol: 30 true
okulary: 54 false
okulary: 54 false
```

Używanie unii kształtu typu

W rozdziale 7. przedstawiłem funkcjonalność w postaci unii pozwalającej na wyrażenie wielu typów jako jedności, co pozwala np. tablicy lub parametrom funkcji na akceptowanie wielu typów. Wyjaśniłem również, że unie typów są pełnoprawnymi typami i zawierają właściwości definiowane przez wszystkie tworzące je typy. To nie jest szczególnie użyteczna funkcjonalność podczas pracy z uniami prostych typów danych, ponieważ zawierają one niewiele właściwości. Sytuacja zmienia się podczas pracy z obiektami, o czym możesz się przekonać, analizując przykład zaprezentowany na listingu 10.14.

Listing 10.14. Używanie unii typów w kodzie pliku `index.ts` w katalogu `src`

```
type Product = {
  id: number,
  name: string,
  price?: number
};

type Person = {
  id: string,
  name: string,
  city: string
};

let hat = { id: 1, name: "czapka", price: 100 };
let gloves = { id: 2, name: "rękawiczki", price: 75 };
let umbrella = { id: 3, name: "parasol", price: 30 };
let bob = { id: "bnowak", name: "Bartek", city: "Londyn" };

let dataItems: (Product | Person)[] = [hat, gloves, umbrella, bob];

dataItems.forEach(item => console.log(`ID: ${item.id}, imię: ${item.name}`));
```

Tablica `dataItems` w tym przykładzie zawiera adnotację z unią typów `Product` i `Person`. Te typy mają dwie wspólne właściwości, `id` i `name`, co oznacza, że te właściwości mogą być używane podczas przetwarzania tablicy bez konieczności zawężania do pojedynczego typu:

```
...
dataItems.forEach(item => console.log(`ID: ${item.id}, imię: ${item.name}`));
...
```

Nie są to jedyne dostępne właściwości, ale jedyne współdzielone przez wszystkie typy tworzące unię. Każda próba uzyskania dostępu do właściwości `price` zdefiniowanej przez typ `Product` lub do właściwości `city` zdefiniowanej przez typ `Person` spowoduje wygenerowanie błędu, ponieważ te właściwości nie są częścią unii `Product | Person`. Kod przedstawiony na listingu 10.14 powoduje wygenerowanie następujących danych wyjściowych:

```
ID: 1, imię: czapka
ID: 2, imię: rękawiczki
ID: 3, imię: parasol
ID: bnowak, imię: Bartek
```

Typy właściwości unii

Gdy tworzona jest unia kształtu typu, typy poszczególnych właściwości występujące we wszystkich typach są łączone również za pomocą unii. Ten efekt można znacznie łatwiej zrozumieć przez utworzenie typu będącego odpowiednikiem unii, jak pokazałem na listingu 10.15.

Listing 10.15. Tworzenie odpowiednika typu w kodzie pliku `index.ts` w katalogu `src`

```
type Product = {
    id: number,
    name: string,
    price?: number
};

type Person = {
    id: string,
    name: string,
    city: string
};

type UnionType = {
    id: number | string,
    name: string
};

let hat = { id: 1, name: "czapka", price: 100 };
let gloves = { id: 2, name: "rękawiczki", price: 75 };
let umbrella = { id: 3, name: "parasol", price: 30 };
let bob = { id: "bnowak", name: "Bartek", city: "Londyn" };

let dataItems: UnionType[] = [hat, gloves, umbrella, bob];

dataItems.forEach(item => console.log(`ID: ${item.id}, nazwa lub imię: ${item.name}`));
```

Typ `UnionType` pokazuje efekt zastosowania unii między typami `Product` i `Person`. Typem właściwości `id` jest `number | string`, ponieważ właściwość `id` w typie `Product` jest liczbą (`number`), natomiast właściwość `id` w typie `Person` jest ciągiem tekstowym (`string`). W obu typach właściwość `name` jest ciągiem tekstowym (`string`), więc to jest typ właściwości `name` w unii. Po skompilowaniu i uruchomieniu kodu przedstawionego na listingu 10.15 zostaną wygenerowane następujące dane wyjściowe:

```
ID: 1, nazwa lub imię: czapka
ID: 2, imię: rękawiczki
ID: 3, imię: parasol
ID: bnowak, nazwa lub imię: Bartek
```

Używanie wartownika typu dla obiektu

W poprzedniej sekcji pokazałem, jak użyteczna może być unia kształtów, ponieważ jest ona oddzielnym typem. Mimo to wartownik typu nadal jest wymagany, aby określony typ mógł uzyskać dostęp do całej funkcjonalności definiowanej przez unię.

W rozdziale 7. pokazałem, jak wykorzystać słowo kluczowe `typeof` podczas tworzenia wartownika typu. To słowo kluczowe jest standardową funkcjonalnością JavaScriptu, którą kompilator TypeScriptu rozpoznaje i której używa podczas procesu sprawdzania typu. Jednak słowo kluczowe `typeof` nie może być używane z obiektami, ponieważ zawsze zwraca ten sam wynik, jak pokazałem w przykładzie na listingu 10.16.

Listing 10.16. *Używanie wartownika typu w kodzie pliku `index.ts` w katalogu `src`*

```
type Product = {
  id: number,
  name: string,
  price?: number
};

type Person = {
  id: string,
  name: string,
  city: string
};

let hat = { id: 1, name: "czapka", price: 100 };
let gloves = { id: 2, name: "rękawiczki", price: 75 };
let umbrella = { id: 3, name: "parasol", price: 30 };
let bob = { id: "bnowak", name: "Bartek", city: "Londyn" };

let dataItems: (Product | Person)[] = [hat, gloves, umbrella, bob];

dataItems.forEach(item => console.log(`ID: ${item.id}, Typ: ${typeof item}`));
```

Kod na omawianym listingu zeruje typ tablicy jako unię typów `Product` i `Person`, a następnie używa słowa kluczowego `typeof` w funkcji `forEach()` w celu ustalenia typu poszczególnych elementów tablicy. Po skompilowaniu i uruchomieniu kodu przedstawionego na listingu 10.16 zostaną wygenerowane następujące dane wyjściowe:

```
ID: 1, Typ: object
ID: 2, Typ: object
ID: 3, Typ: object
ID: bnowak, Typ: object
```

Kształt typu to funkcjonalność dostarczana całkowicie przez TypeScript. Wszystkie obiekty mają w JavaScriptcie typ `object`, więc słowo kluczowe `typeof` nie przedstawia żadnej użyteczności podczas ustalania typu obiektu `Product` lub `Person`.

Wartownik typu poprzez sprawdzanie właściwości

Najprostszym sposobem na rozróżnianie kształtów typu jest skorzystanie ze słowa kluczowego `in` w JavaScriptcie i użycie go do sprawdzenia właściwości, jak pokazałem na listingu 10.17.

Listing 10.17. Używanie wartownika typu w kodzie pliku `index.ts` w katalogu `src`

```
type Product = {
  id: number,
  name: string,
  price?: number
};

type Person = {
  id: string,
  name: string,
  city: string
};

let hat = { id: 1, name: "czapka", price: 100 };
let gloves = { id: 2, name: "rękawiczki", price: 75 };
let umbrella = { id: 3, name: "parasol", price: 30 };
let bob = { id: "bnowak", name: "Bartek", city: "Londyn" };

let dataItems: (Product | Person)[] = [hat, gloves, umbrella, bob];

dataItems.forEach(item => {
  if ("city" in item) {
    console.log(`Typ Person: ${item.name}: ${item.city}`);
  } else {
    console.log(`Typ Product: ${item.name}: ${item.price}`);
  }
});
```

Celem jest możliwość ustalenia, z którym typem (`Product` lub `Person`) jest zgodny każdy obiekt tablicy. Wiadomo, że tablica może zawierać tylko te typy, ponieważ jej adnotacja typu ma postać `(Product | Person)[]`.

Kształt to połączenie właściwości, a wartownik typu musi sprawdzać każdą lub większość właściwości dołączonych w jednym kształcie, choć nie w pozostałych kształtach. W przypadku kodu przedstawionego na listingu 10.7 każdy obiekt zawierający właściwość `city` musi być zgodny z kształtem `Person`, ponieważ wymieniona właściwość nie jest częścią kształtu `Product`. W celu zastosowania wartownika typu przeprowadzającego sprawdzenie pod kątem właściwości nazwa właściwości jest wyrażona w postaci ciągu tekstowego, następnie znajduje się słowo kluczowe `in` i dalej sprawdzany obiekt, jak pokazałem na rysunku 10.4.

```

if ( "city" in item ) {
  console.log(`Typ Person...`)
} else {
  console.log(`Typ Product...`)
}

```

Diagram illustrating the use of the `in` keyword. The code snippet shows a conditional check: `if ("city" in item)`. The compiler determines the type of `item` based on the context. In the first block, the type is `Person`, and in the second block, it is `Product`.

Rysunek 10.4. Używanie słowa kluczowego `in`

Wyrażenie `in` zwraca wartość `true` dla obiektów definiujących określoną właściwość, natomiast w przypadku pozostałych obiektów wartością zwrótną jest `false`. Kompilator TypeScript rozpoznaje znaczenie operacji sprawdzenia pod kątem właściwości oraz określa typ w blokach kodu konstrukcji `if-else`. Po skompilowaniu i uruchomieniu kodu przedstawionego na listingu 10.17 zostaną wygenerowane następujące dane wyjściowe:

```

Typ Product: czapka: 100
Typ Product: rękawiczki: 75
Typ Product: parasol: 30
Typ Person: Bartek: Londyn

```

Unikanie najczęściej spotykanych problemów podczas używania wartownika typu

Bardzo ważne znaczenie ma tworzenie testów wartownika typu, które będą pozwalały na ostateczne i dokładne rozróżnianie między typami. Jeżeli kompilator wygeneruje nieoczekiwane błędy podczas używania wartownika typu, wówczas przyczyną prawdopodobnie może być nieodpowiednie wyrażenie warunkowe.

Są dwa najczęściej spotykane problemy, których należy unikać. Pierwszy polega na utworzeniu niedokładnego wyrażenia warunkowego, co uniemożliwia niezawodne rozróżnianie typów. Spójrz na przykładowe wyrażenie warunkowe:

```

dataItems.forEach(item => {
  if ("id" in item && "name" in item) {
    console.log(`Typ Person: ${item.name}: ${item.city}`);
  } else {
    console.log(`Typ Product: ${item.name}: ${item.price}`);
  }
});

```

W tym wyrażeniu warunkowym sprawdzane są właściwości `id` i `name`, przy czym są one definiowane w obu typach, `Person` i `Product`, więc wynik wykonania tego wyrażenia nie daje kompilatorowi wystarczającej ilości informacji pozwalających na ustalenie typu. Typem określonym w bloku `if` jest `Product` | `Person`, co oznacza, że próba użycia właściwości `city` zakończy się wygenerowaniem błędu. Typem określonym w bloku `else` jest `never`, ponieważ wszystkie możliwe typy zostały już ustalone i kompilator będzie generował błędy w przypadku użycia właściwości `name` i `price`.

Podobny problem pojawia się podczas operacji sprawdzania pod kątem właściwości opcjonalnej, np.:

```
dataItems.forEach(item => {
  if ("price" in item) {
    console.log(`Typ Product: ${item.name}: ${item.price}`);
  } else {
    console.log(`Typ Person: ${item.name}: ${item.city}`);
  }
});
```

To wyrażenie warunkowe spowoduje dopasowanie obiektów definiujących właściwość `price`, więc typem ustalonym w bloku `if` będzie `Product`, zgodnie z oczekiwaniami (zwróć uwagę na to, że polecenia w blokach kodu są w tym przykładzie odwrócone). Problem polega na niebezpieczeństwie dopasowania obiektu do kształtu typu `Product` nawet w przypadku braku właściwości `price`. Dlatego też typem określonym w bloku `else` jest `Product | Person` i kompilator zgłosi błąd w przypadku użycia właściwości `city`.

Tworzenie efektywnych wyrażeń warunkowych dla typów może wymagać starannej analizy i dokładnego testowania, choć wraz ze wzrostem doświadczenia ten proces staje się coraz łatwiejszy.

Wartownik typu z funkcją predykatu

Słowo kluczowe `in` to użyteczny sposób na ustalenie zgodności obiektu z kształtem, choć jednocześnie wymaga tych samych operacji sprawdzenia za każdym razem, gdy zachodzi potrzeba utworzenia egzemplarza typu. TypeScript obsługuje również wartowników typu wykorzystujących funkcje, jak pokazałem na listingu 10.18.

Listing 10.18. Wartownik typu z funkcją w kodzie pliku `index.ts` w katalogu `src`

```
type Product = {
  id: number,
  name: string,
  price?: number
};

type Person = {
  id: string,
  name: string,
  city: string
};

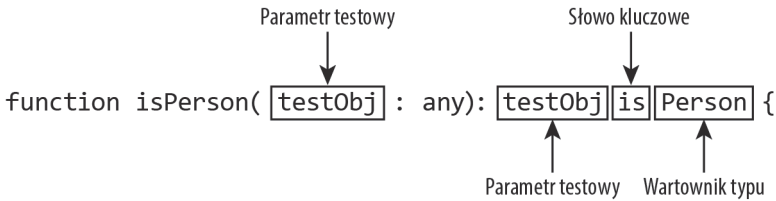
let hat = { id: 1, name: "czapka", price: 100 };
let gloves = { id: 2, name: "rękawiczki", price: 75 };
let umbrella = { id: 3, name: "parasol", price: 30 };
let bob = { id: "bnowak", name: "Bartek", city: "Londyn" };

let dataItems: (Product | Person)[] = [hat, gloves, umbrella, bob];

function isPerson(testObj: any): testObj is Person {
  return testObj.city !== undefined;
}
```

```
dataItems.forEach(item => {
  if (isPerson(item)) {
    console.log(`Typ Person: ${item.name}: ${item.city}`);
  } else {
    console.log(`Typ Product: ${item.name}: ${item.price}`);
  }
});
```

Wartownik typu dla tego obiektu korzysta z funkcji opierającej działanie na słowie kluczowym `in`, jak pokazałem na rysunku 10.5.



Rysunek 10.5. Wartownik typu z funkcją

Wynik działania funkcji *typu predykatu* wskazuje kompilatorowi sprawdzane parametry funkcji i typ, pod kątem którego odbywa się operacja sprawdzenia. W omawianym przykładzie funkcja `isPerson()` sprawdza parametr `testObj` pod kątem typu `Person`. Jeżeli wynikiem działania funkcji jest `true`, wówczas kompilator TypeScriptu będzie traktował obiekt jako `Person`.

Użycie funkcji dla wartownika typu może być znacznie elastyczniejszym rozwiązaniem, ponieważ typem parametru jest `any`, co pozwala na sprawdzanie właściwości pod kątem stosowania literałów w postaci ciągów tekstowych oraz słowa kluczowego `in`.

■ **Wskazówka** Wprawdzie nie ma żadnych ograniczeń związanych z funkcją wartownika, ale zgodnie z konwencją stosowany jest prefiks `is`. Dlatego też funkcja sprawdzająca pod kątem typu `Person` ma nazwę `isPerson()`, natomiast sprawdzająca pod kątem typu `Product` ma nazwę `isProduct()`.

Po skompilowaniu i uruchomieniu kodu przedstawionego na listingu 10.18 zostaną wygenerowane następujące dane wyjściowe potwierdzające, że użycie funkcji predykatu ma taki sam efekt jak zastosowanie słowa kluczowego `in`:

```
Typ Product: czapka: 100
Typ Product: rękawiczki: 75
Typ Product: parasol: 30
Typ Person: Bartek: Londyn
```

Używanie złączenia typów

Złączenie typów oznacza połączenie w sobie funkcjonalności wielu typów i pozwala na wykorzystanie całej dostępnej funkcjonalności. To przeciwieństwo unii typów, w której są używane części wspólne typów. Na listingu 10.19 pokazałem przykład zdefiniowania i użycia części złączenia typów.

Listing 10.19. Definiowanie złączenia typów w kodzie pliku *index.ts* w katalogu *src*

```
type Person = {
  id: string,
  name: string,
  city: string
};

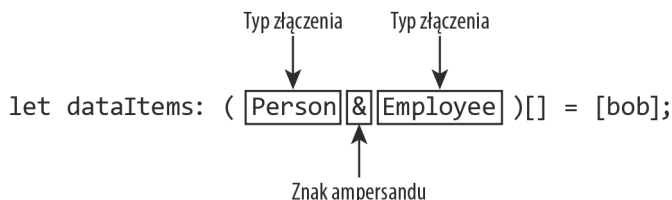
type Employee = {
  company: string,
  dept: string
};

let bob = { id: "bnowak", name: "Bartek", city: "Londyn",
  company: "Acme Co", dept: "handlowiec" };

let dataItems: (Person & Employee)[] = [bob];

dataItems.forEach(item => {
  console.log(`Typ Person: ${item.id}, ${item.name}, ${item.city}`);
  console.log(`Typ Employee: ${item.id}, ${item.company}, ${item.dept}`);
});
```

Typ tablicy `dataItems` został zdefiniowany jako złączenie typów `Person` i `Employee`. Złączenie jest definiowane za pomocą znaku `&` umieszczanego między dwoma (lub więcej) typami, jak pokazałem na rysunku 10.6.



```
let dataItems: ( Person & Employee )[] = [bob];
```

Znak ampersandu

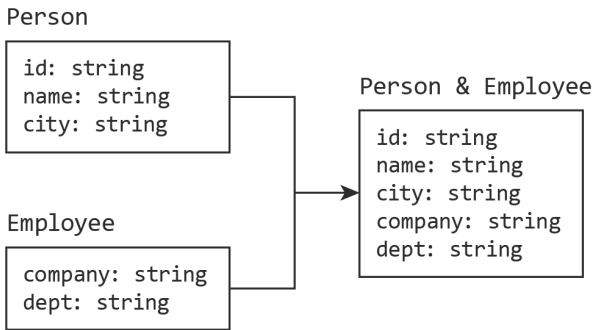
Rysunek 10.6. Definiowanie złączenia typów

Obiekt jest zgodny z kształtem typu złączenia tylko wtedy, gdy definiuje właściwości otrzymane po połączeniu wszystkich wiązanych ze sobą typów, jak pokazałem na rysunku 10.7.

Na listingu 10.19 złączenie typów `Person` i `Employee` ma następujący efekt: tablica `dataItems` może zawierać jedynie obiekty definiujące właściwości `id`, `name`, `city`, `company` i `dept`.

Zawartość tablicy jest przetwarzana za pomocą metody `forEach()`, która pokazuje, że mogą być wykorzystywane właściwości pochodzące z obu złączonych ze sobą typów. Po skompilowaniu i uruchomieniu kodu przedstawionego na listingu 10.19 zostaną wygenerowane następujące dane wyjściowe:

```
Typ Person: bnowak, Bartek, Londyn
Typ Employee: bnowak, Acme Co, handlowiec
```



Rysunek 10.7. Efekt złączenia typów

Używanie złączenia do korelacji danych

Złączenie jest użyteczne w sytuacji, gdy otrzymujesz obiekty z jednego źródła i musisz wprowadzić nową funkcjonalność, aby mogły być wykorzystane w innym miejscu aplikacji, lub gdy zachodzi potrzeba połączenia bądź korelacji obiektów z dwóch źródeł danych. JavaScript niezwykle ułatwia wprowadzenie w obiekcie funkcjonalności pochodzącej z innego obiektu, natomiast złączenie pozwala na czytelne opisywanie typów, aby mogły być sprawdzane przez kompilator TypeScriptu. Na listingu 10.20 przedstawiłem funkcję przeprowadzającą korelację dwóch tablic danych.

Listing 10.20. Korelowanie danych w kodzie pliku *index.ts* w katalogu *src*

```

type Person = {
  id: string,
  name: string,
  city: string
};

type Employee = {
  id: string,
  company: string,
  dept: string
};

type EmployedPerson = Person & Employee;

function correlateData(peopleData: Person[], staff: Employee[]): EmployedPerson[] {
  const defaults = { company: "brak", dept: "brak" };
  return peopleData.map(p => ({ ...p,
    ...staff.find(e => e.id === p.id) || { ...defaults, id: p.id } }));
}

let people: Person[] =
  [{ id: "bnowak", name: "Bartek Nowak", city: "Londyn" },
  { id: "ajanowska", name: "Alicja Janowska", city: "Paryż"},
  { id: "dpetecka", name: "Dorota Petecka", city: "Nowy Jork"}];

let employees: Employee[] =

```

```
[{ id: "bnowak", company: "Acme Co", dept: "handlowiec" },
 { id: "dpeteka", company: "Acme Co", dept: "programista" }];
```

```
let dataItems: EmployedPerson[] = correlateData(people, employees);
```

```
dataItems.forEach(item => {
  console.log(`Typ Person: ${item.id}, ${item.name}, ${item.city}`);
  console.log(`Typ Employee: ${item.id}, ${item.company}, ${item.dept}`);
});
```

W tym przykładzie funkcja `correlateData()` otrzymuje tablicę obiektów `Person` i tablicę obiektów `Employee`, a następnie używa współdzielonej przez nie właściwości `id` do wygenerowania obiektów łączących właściwości obu kształtów typu. Podczas przetwarzania obiektu `Person` przez metodę `map()` metoda `find()` jest używana do odszukania obiektu `Employee` o takiej samej wartości `id`. Z kolei operator rozwinęcia pozwala na utworzenie obiektów dopasowanych do kształtu złączenia. Skoro wyniki działania funkcji `correlateData()` mają definiować wszystkie właściwości złączenia, zdecydowałem się na wykorzystanie wartości domyślnych, gdy nie znajduje się dopasowania w postaci obiektu `Employee`.

```
...
const defaults = { company: "brak", dept: "brak"};
return peopleData.map(p => ({ ...p,
  ...staff.find(e => e.id === p.id) || { ...defaults, id: p.id } }));
...
```

Adnotacje typu na listingu 10.20 zostały użyte w celu wyraźnego wskazania przeznaczenia kodu, który powinien działać także bez nich. Kompilator TypeScriptu potrafi właściwie ustalić efekt wykonania poleceń i tworzy obiekty zgodne z kształtem typu złączenia.

Po skompilowaniu i uruchomieniu kodu przedstawionego na listingu 10.20 zostaną wygenerowane następujące dane wyjściowe:

```
Typ Person: bnowak, Bartek Nowak, Londyn
Typ Employee: bnowak, Acme Co, handlowiec
Typ Person: ajanowska, Alicja Janowska, Paryż
Typ Employee: ajanowska, brak, brak
Typ Person: dpeteka, Dorota Petecka, Nowy Jork
Typ Employee: dpeteka, Acme Co, programista
```

Łączenie złączeń

Ponieważ złączenie łączy w sobie funkcjonalność pochodzącą z wielu typów, obiekt zgodny z kształtem złączenia pozostaje także zgodny z poszczególnymi typami złączeń. Na przykład obiekt zgodny ze złączeniem `Person & Employee` może być stosowany wszędzie tam, gdzie oczekuje się obiektu typu `Person` lub `Product`, jak pokazałem w przykładzie na listingu 10.21.

Listing 10.21. Używanie typów złączenia w kodzie pliku `index.ts` w katalogu `src`

```
type Person = {
  id: string,
  name: string,
  city: string
```

```

};

type Employee = {
  id: string,
  company: string,
  dept: string
};

type EmployedPerson = Person & Employee;

function correlateData(peopleData: Person[], staff: Employee[]): EmployedPerson[] {
  const defaults = { company: "brak", dept: "brak" };
  return peopleData.map(p => ({ ...p,
    ...staff.find(e => e.id === p.id) || { ...defaults, id: p.id } }));
}

let people: Person[] =
  [{ id: "bnowak", name: "Bartek Nowak", city: "Londyn" },
    { id: "ajanowska", name: "Alicja Janowska", city: "Paryż" },
    { id: "dpeteca", name: "Dorota Petecka", city: "Nowy Jork" }];

let employees: Employee[] =
  [{ id: "bnowak", company: "Acme Co", dept: "handlowiec" },
    { id: "dpeteca", company: "Acme Co", dept: "programista" }];

let dataItems: EmployedPerson[] = correlateData(people, employees);

function writePerson(per: Person): void {
  console.log(`Typ Person: ${per.id}, ${per.name}, ${per.city}`);
}

function writeEmployee(emp: Employee): void {
  console.log(`Typ Employee: ${emp.id}, ${emp.company}, ${emp.dept}`);
}

dataItems.forEach(item => {
  writePerson(item);
  writeEmployee(item);
});

```

Kompilator dopasowuje obiekt do kształtu przez zagwarantowanie, że definiuje on wszystkie właściwości kształtu i nie przejmuje się nadmiarowymi właściwościami (z wyjątkiem sytuacji, w której obiekt jest definiowany jako literał, co wyjaśniłem we wcześniejszej części rozdziału). Obiekty zgodne z typem `EmployedPerson` mogą być używane przez funkcje `writePerson()` i `writeEmployee()`, ponieważ są zgodne z typami wymienionymi w parametrach funkcji. Po skompilowaniu i uruchomieniu kodu przedstawionego na listingu 10.21 zostaną wygenerowane następujące dane wyjściowe:

```

Typ Person: bnowak, Bartek Nowak, Londyn
Typ Employee: bnowak, Acme Co, handlowiec
Typ Person: ajanowska, Alicja Janowska, Paryż
Typ Employee: ajanowska, brak, brak
Typ Person: dpeteca, Dorota Petecka, Nowy Jork
Typ Employee: dpeteca, Acme Co, programista

```

Może się wydawać oczywiste, że typ złączenia jest zgodny ze wszystkimi jego elementami składowymi, ale ma to ważny wpływ w sytuacji, gdy typy złączenia definiują właściwości o takich samych nazwach: typ właściwości w złączeniu jest złączeniem poszczególnych typów właściwości. Być może trudno jest zrozumieć poprzednie zdanie, więc w kolejnych sekcjach przedstawię znacznie bardziej użyteczne wyjaśnienie jego sensu.

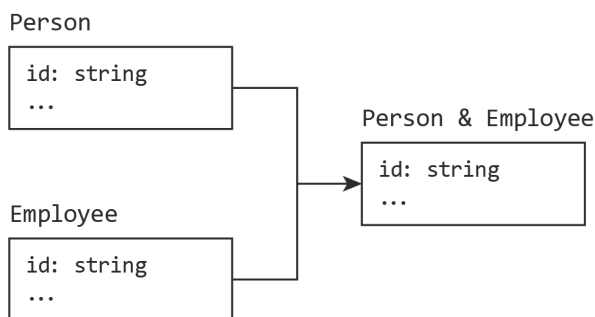
Złączanie właściwości tego samego typu

Najprostsza sytuacja zachodzi wtedy, gdy właściwości mają takie same nazwy i typy, np. właściwość `id` jest definiowana przez typy `Person` i `Employee` — w takim przypadku są złączane bez żadnych zmian, jak pokazałem na rysunku 10.8.

W przedstawionej sytuacji nie ma żadnych skutków ubocznych, ponieważ dowolna wartość przypisywana właściwości `id` jest ciągiem tekstowym i pozostaje zgodna z wymaganiami zarówno obiektu, jak i typu złączenia.

Złączanie właściwości różnych typów

Jeżeli istnieją właściwości o takiej samej nazwie, ale o różnych typach, wówczas kompilator zachowuje nazwę i łączy typy. Aby zaprezentować takie rozwiązanie, w kodzie przedstawionym na listingu 10.22 usunąłem funkcje i dodałem właściwość `contact` do typów `Person` i `Employee`.



Rysunek 10.8. Złączanie właściwości tego samego typu

Listing 10.22. Dodawanie właściwości różnych typów w kodzie pliku `index.ts` w katalogu `src`

```

type Person = {
  id: string,
  name: string,
  city: string,
  contact: number
};

type Employee = {
  id: string,
  company: string,
  dept: string,
  contact: string
};
  
```

```
};

type EmployedPerson = Person & Employee;

let typeTest = ({ as EmployedPerson }).contact;
```

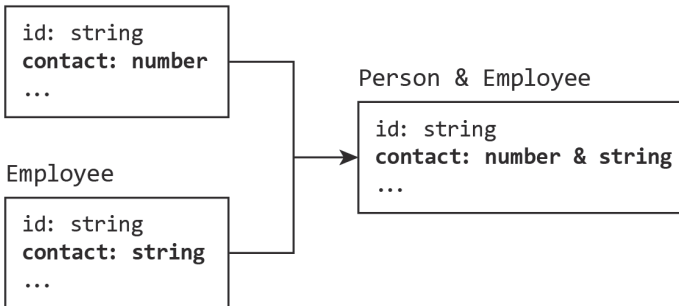
Ostatnie polecenie na listingu 10.22 prezentuje użyteczną sztuczkę pozwalającą sprawdzić, który typ kompilator przypisał właściwości złączenia — w tym celu wystarczy spojrzeć na plik deklaracji utworzony w katalogu *dist*, gdy opcja konfiguracyjna kompilatora *declaration* ma przypisaną wartość *true*. Polecenie używa asercji typu do wskazania kompilatorowi, że pusty obiekt jest zgodny z typem *EmployedPerson*, co prowadzi do przypisania właściwości *contact* zmiennej *typeTest*. Po zapisaniu zmian w pliku *index.ts* kompilator przeprowadzi kompilację kodu, a w pliku *index.d.ts* w katalogu *dist* będzie można sprawdzić typ określony dla właściwości *contact* w złączeniu:

```
declare let typeTest: number & string;
```

Kompilator utworzył złączenie między typem właściwości *contact* zdefiniowanej przez *Person* a typem właściwości *contact* zdefiniowanej przez *Employee*, jak pokazałem na rysunku 10.9.

Tworzenie złączeń typów to tylko jeden ze sposobów, w jaki kompilator może łączyć właściwości. Jednak nie spowoduje to wygenerowania użytecznego wyniku, ponieważ nie istnieją wartości, które mogą być przypisywane złączeniom typów prostych *number* i *string*, jak pokazałem na listingu 10.23.

Person



Rysunek 10.9. Złączanie właściwości różnych typów

Listing 10.23. Przypisywanie wartości złączeniom typów prostych w kodzie pliku *index.ts* w katalogu *src*

```
type Person = {
  id: string,
  name: string,
  city: string,
  contact: number
};

type Employee = {
  id: string,
  company: string,
  dept: string,
```

```

    contact: string
  };

type EmployedPerson = Person & Employee;

let typeTest = ({} as EmployedPerson).contact;

let person1: EmployedPerson = {
  id: "bnowak", name: "Bartek Nowak", city: "Londyn",
  company: "Acme Co", dept: "handlowiec", contact: "Alicja"
};

let person2: EmployedPerson = {
  id: "dpeteccka", name: "Dorota Peteccka", city: "Nowy Jork",
  company: "Acme Co", dept: "programista", contact: 6512346543
};

```

Obiekt musi przypisać wartość właściwości `contact`, aby był zgodny z kształtem, choć prowadzi to do wygenerowania następujących błędów:

```

src/index.ts(21,40): error TS2322: Type 'string' is not assignable to type 'never'.
src/index.ts(26,46): error TS2322: Type 'number' is not assignable to type 'never'.

```

Złączenie typów `number` i `string` prowadzi do powstania typu niemożliwego. Nie istnieje rozwiązanie dla tego problemu dla typów podstawowych i można jedynie dostosować typy wykorzystane w złączeniu, aby były używane te kształty typów zamiast typów podstawowych, jak pokazałem na listingu 10.24.

■ **Uwaga** Wprawdzie może wydawać się dziwne, że kompilator TypeScriptu pozwala na definiowanie typów niemożliwych, ale wybrane funkcje zaawansowane TypeScriptu (omówione w późniejszych rozdziałach) utrudniają kompilatorowi właściwe działanie w takich sytuacjach. Tworzący TypeScript zespół w Microsoftzie zdecydował się na prostotę zamiast dokładniejszego sprawdzania pod kątem każdego typu niemożliwego.

Listing 10.24. Używanie kształtu typów w złączeniu w kodzie pliku `index.ts` w katalogu `src`

```

type Person = {
  id: string,
  name: string,
  city: string,
  contact: { phone: number }
};

type Employee = {
  id: string,
  company: string,
  dept: string,
  contact: { name: string }
};

type EmployedPerson = Person & Employee;

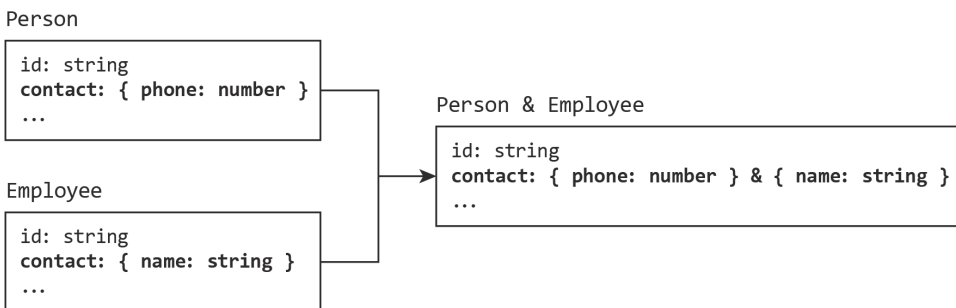
let typeTest = ({} as EmployedPerson).contact;

```

```
let person1: EmployedPerson = {
  id: "bnowak", name: "Bartek Nowak", city: "Londyn",
  company: "Acme Co", dept: "handlowiec",
  contact: { name: "Alicja", phone: 6512346543 }
};

let person2: EmployedPerson = {
  id: "dpetecka", name: "Dorota Petecka", city: "Nowy Jork",
  company: "Acme Co", dept: "programista",
  contact: { name: "Alicja", phone: 6512346543 }
};
```

Kompilator obsługuje połączenie właściwości w taki sam sposób, ale wynikiem złączenia jest kształt zawierający właściwości `name` i `phone`, jak pokazałem na rysunku 10.10.



Rysunek 10.10. Łączenie właściwości za pomocą kształtu typów

Złączenie obiektu z właściwością `phone` i obiektu z właściwością `name` prowadzi do powstania obiektu zawierającego właściwości `phone` i `name`, co pozwala na przypisanie wartości właściwości `contact`, a to z kolei oznacza zgodność z typami `Person` i `Employee` oraz z ich złączeniem.

Łączenie metod

Jeżeli typy w złączeniu definiują metody o takiej samej nazwie, wówczas kompilator utworzy funkcję, której sygnatura jest złączeniem, jak pokazałem na listingu 10.25.

Listing 10.25. Łączenie metod w kodzie pliku `index.ts` w katalogu `src`

```
type Person = {
  id: string,
  name: string,
  city: string,
  getContact(field: string): string
};

type Employee = {
  id: string,
  company: string,
  dept: string,
  getContact(field: number): number
};
```

```
type EmployedPerson = Person & Employee;

let person: EmployedPerson = {
  id: "bnowak", name: "Bartek Nowak", city: "Londyn",
  company: "Acme Co", dept: "handlowiec",
  getContact(field: string | number): any {
    return typeof field === "string" ? "Alicja" : 6512346543;
  }
};

let typeTest = person.getContact;
let stringParamTypeTest = person.getContact("Alicja");
let numberParamTypeTest = person.getContact(123);

console.log(`Typ Contact: ${person.getContact("Alicja")}`);
console.log(`Typ Contact: ${person.getContact(123)}`);
```

Kompilator łączy funkcje przez utworzenie złączenia ich sygnatur, co może doprowadzić do powstania typów niemożliwych lub funkcji, które nie będą mogły zostać zaimplementowane w użyteczny sposób. W omawianym przykładzie metody `getContact()` w typach `Person` i `Employee` są przedmiotem złożenia, jak pokazałem na rysunku 10.11.

Person

```
id: string
getContact(field: string): string
...
```

Employee

```
id: string
getContact(field: number): string
...
```

Person & Employee

```
id: string
getContact(field: string) => string
  & (field: number) => string
...
```

Rysunek 10.11. Łączenie metod

Ustalenie konsekwencji łączenia metod w złączeniu może być trudne, ale ogólny efekt jest podobny do przeciążenia, które omówiłem w rozdziale 8. Bardzo często opieram się na pliku deklaracji typu, aby sprawdzić, czy otrzymałem oczekiwane złączenie. W kodzie przedstawionym na listingu 10.25 znajdują się trzy polecenia pomagające w sprawdzeniu, jak odbyło się łączenie metod:

```
...
let typeTest = person.getContact;
let stringParamTypeTest = person.getContact("Alicja");
let numberParamTypeTest = person.getContact(123);
...
```

Po zapisaniu i skompilowaniu pliku `index.ts` plik `index.d.ts` w katalogu `dist` będzie zawierał polecenia pokazujące typ, który kompilator przypisał poszczególnym zmiennym:

```
declare let typeTest: ((field: string) => string) & ((field: number) => number);
declare let stringParamTypeTest: string;
declare let numberParamTypeTest: number;
```

Pierwsze polecenie pokazuje typ złączonych metod, natomiast pozostałe pokazują typ wartości zwrotnej, gdy przekazywane są argumenty typów `string` i `number`. (Przeznaczenie pliku *index.d.ts* wyjaśnię w rozdziale 14., ale często okazuje się on użyteczny do sprawdzenia sposobu działania kompilatora).

Implementacja złączenia metod musi zachowywać zgodność z metodami złączenia. Parametry zwykle nie sprawiają trudności; w przykładzie przedstawionym na listingu 10.25 wykorzystałem unię typów do utworzenia metody otrzymującej wartości typów `string` i `number`. Wynik działania metody może sprawić nieco problemów, ponieważ trudno jest znaleźć typ zapewniający zachowanie zgodności. Przekonałem się, że najbardziej niezawodne podejście polega na używaniu `any` jako typu wyniku działania metody i wartowników typu do przygotowania mapowania między parametrami a typami wyników.

```
...
getContact(field: string | number): any {
    return typeof field === "string" ? "Alicja" : 6512346543;
}
...
```

Staram się unikać stosowania typu `any`, jeśli tylko to możliwe. Jednak nie istnieje żaden typ, który w omawianym przykładzie pozwoliłby na używanie obiektu `EmployedPerson` jako obiektów `Person` i `Employee`. Po skompilowaniu i uruchomieniu kodu przedstawionego na listingu 10.25 zostaną wygenerowane następujące dane wyjściowe:

```
Typ Contact: Alicja
Typ Contact: 6512346543
```

Podsumowanie

W tym rozdziale pokazałem, w jaki sposób TypeScript używa kształtu obiektu do sprawdzania typu. Wyjaśniłem również temat porównywania kształtów, używania kształtów do definiowania aliasów, a także do łączenia kształtów na postać unii i złączeń. W następnym rozdziale dowiesz się, jak używać funkcjonalności kształtu w celu zapewnienia obsługi typów w klasach.

ROZDZIAŁ 11.



Praca z klasami i interfejsami

W tym rozdziale przedstawię funkcje oferowane przez TypeScript do pracy z klasami oraz wprowadzę funkcjonalność interfejsu, która zapewnia alternatywne podejście w zakresie opisywania kształtu obiektów. Streszczenie materiału przedstawionego w rozdziale znajdziesz w tabeli 11.1.

Tabela 11.1. Streszczenie materiału przedstawionego w rozdziale

Problem	Rozwiązanie	Listing
Spójne tworzenie obiektów	Użyj funkcji konstruktora lub zdefiniuj klasę	Od 4 do 6 i od 13 do 15
Uniemożliwienie dostępu do właściwości i metod	Użyj słów kluczowych kontroli dostępu w TypeScriptie lub właściwości prywatnych JavaScriptu	Od 7 do 10
Uniemożliwienie modyfikowania właściwości	Użyj słowa kluczowego <code>readonly</code>	11
Otrzymywanie parametru konstruktora i tworzenie egzemplarza właściwości w jednym kroku	Użyj zwięzłej składni konstruktora	12
Definiowanie częściowej funkcjonalności dziedziczonej przez podklasy	Zdefiniuj klasę abstrakcyjną	16 i 17
Definiowanie kształtu, który może być implementowany przez klasę	Zdefiniuj interfejs	Od 18 do 23
Dynamiczne definiowanie właściwości	Użyj sygnatury indeksu	Od 24 do 28

W tabeli 11.2 wymieniłem opcje kompilatora TypeScriptu użyte w rozdziale.

Tabela 11.2. Opcje kompilatora TypeScriptu użyte w rozdziale

Opcja	Opis
target	Ta opcja określa docelową wersję języka JavaScript, dla której kompilator wygeneruje kod
outDir	Ta opcja określa katalog, w którym zostaną umieszczone wygenerowane pliki JavaScriptu
rootDir	Ta opcja określa katalog główny używany przez kompilator do wyszukiwania plików TypeScriptu
declaration	Ta opcja generuje pliki deklaracji typu, które pomagają zrozumieć, w jaki sposób zostały ustalone typy dla kodu JavaScriptu. Wspomniane pliki zostaną szczegółowo omówione w rozdziale 14.
noUnchecked ↳ IndexedAccess	Ta opcja uniemożliwia uzyskiwanie dostępu do właściwości za pomocą sygnatury indeksu, o ile nie będą one chronione przed wartościami undefined

Przygotowanie projektu

W rozdziale będę kontynuował używanie projektu *types* utworzonego w rozdziale 7. i uaktualnionego w kolejnych. Aby przygotować się do tego rozdziału, należy zastąpić zawartość pliku *index.ts* w katalogu *src* kodem przedstawionym na listingu 11.1.

Listing 11.1. Nowa zawartość pliku *index.ts* w katalogu *src*

```
type Person = {
  id: string,
  name: string,
  city: string
};

let data: Person[] =
  [{ id: "bnowak", name: "Bartek Nowak", city: "Londyn" },
  { id: "ajanowska", name: "Alicja Janowska", city: "Paryż"},
  { id: "dpetecka", name: "Dorota Petecka", city: "Nowy Jork"}];

data.forEach(item => {
  console.log(`${item.id} ${item.name}, ${item.city}`);
});
```

Wyzeruj konfigurację kompilatora przez zastąpienie zawartości pliku *tsconfig.json* kodem przedstawionym na listingu 11.2.

Listing 11.2. Konfigurowanie kompilatora w pliku *tsconfig.json* w katalogu *types*

```
{
  "compilerOptions": {
    "target": "es2020",
    "outDir": "./dist",
    "rootDir": "./src",
```

```

    "declaration": true,
    // "strictNullChecks": true,
    // "suppressExcessPropertyErrors": true
  }
}

```

Konfiguracja kompilatora zawiera ustawienie `declaration` oznaczające, że kompilator będzie oprócz plików JavaScriptu tworzył również pliki deklaracji typów. Rzeczywiste przeznaczenie tych plików przedstawię w rozdziale 14., natomiast w tym rozdziale będą one używane do wyjaśnienia, jak kompilator radzi sobie z typami danych.

Otwórz nowe okno powłoki, przejdź do katalogu *types*, a następnie z jego poziomu wydaj polecenie przedstawione na listingu 11.3, uruchamiające kompilator TypeScriptu w trybie automatycznego kompilowania kodu po wykryciu zmiany w dowolnym pliku i wykonywania kodu po jego skompilowaniu.

■ **Wskazówka** Przykładowy projekt dla tego rozdziału, a także projekty dla pozostałych rozdziałów książki zostały umieszczone w archiwum dostępnym pod adresem <https://ftp.helion.pl/przyklady/tys4o2.zip>.

Listing 11.3. Uruchamianie kompilatora TypeScriptu

```
$ npm start
```

Kompilator przeprowadzi kompilację kodu w pliku *index.ts*, wykona wygenerowany kod JavaScriptu, a następnie przejdzie do trybu monitorowania i wyświetli następujące dane wyjściowe:

```

7:16:33 AM - Starting compilation in watch mode...
7:16:35 AM - Found 0 errors. Watching for file changes.
bnowak Bartek Nowak, Londyn
ajanowska Alicja Janowska, Paryż
dpeteccka Dorota Peteccka, Nowy Jork

```

Używanie funkcji konstruktora

Jak wyjaśniłem w rozdziale 4., obiekt może być tworzony za pomocą funkcji konstruktora i zapewnienia dostępu do systemu prototypowania w JavaScriptcie. Funkcje konstruktora mogą być używane w kodzie TypeScriptu, choć sposób ich obsługi jest nieintuicyjny i nie tak elegancki jak sposób obsługi klas, o czym się przekonasz w dalszej części rozdziału. Na listingu 11.4 pokazałem dodanie funkcji konstruktora do przykładowego fragmentu kodu.

*Listing 11.4. Używanie funkcji konstruktora w kodzie pliku *index.ts* w katalogu *src**

```

type Person = {
  id: string,
  name: string,
  city: string
};

```

```

let Employee = function(id: string, name: string, dept: string, city: string) {
    this.id = id;
    this.name = name;
    this.dept = dept;
    this.city = city;
};
Employee.prototype.writeDept = function() {
    console.log(`${this.name} pracuje w dziale ${this.dept}`);
};

let salesEmployee = new Employee("fvega", "Fidel Vega", "sprzedaży", "Paryż");

let data: (Person | Employee)[] =
    [{ id: "bnowak", name: "Bartek Nowak", city: "Londyn" },
    { id: "ajanowska", name: "Alicja Janowska", city: "Paryż"},
    { id: "dpetecka", name: "Dorota Petecka", city: "Nowy Jork"},
    salesEmployee];

data.forEach(item => {
    if (item instanceof Employee) {
        item.writeDept();
    } else {
        console.log(`${item.id} ${item.name}, ${item.city}`);
    }
});

```

Funkcja konstruktora `Employee` tworzy obiekt z właściwościami `id`, `name`, `dept` i `city`. W prototypie `Employee` została zdefiniowana metoda o nazwie `writeDept()`. Tablica `data` jest uaktualniana w taki sposób, aby zawierała obiekty `Person` i `Employee`, a funkcja przekazana metodzie `forEach()` używa operatora `instanceof` do zawężenia typu poszczególnych obiektów tablicy. Kod przedstawiony na listingu 11.4 powoduje wygenerowanie następujących błędów:

```

src/index.ts(17,21): error TS2304: Cannot find name 'Employee'.
src/index.ts(17,21): error TS4025: Exported variable 'data' has or is using private name
↳ 'Employee'.
src/index.ts(25,14): error TS2339: Property 'writeDept' does not exist on type '{}'.

```

TypeScript traktuje funkcję konstruktora `Employee` jak każdą inną, szuka typów jej parametrów i wartości zwrótej, aby opisać kształt. Gdy funkcja `Employee` jest używana ze słowem kluczowym `new`, wówczas kompilator użyje typu `any` dla obiektu przypisanego zmiennej `salesEmployee`. Wynikiem jest seria błędów, ponieważ kompilator zmaga się z sensownym sposobem użycia funkcji konstruktora.

Najprostsze rozwiązanie problemu polega na dostarczeniu kompilatorowi dodatkowych informacji o kształtach używanych obiektów. Na listingu 11.5 pokazałem dodanie aliasu typu opisującego obiekty tworzone przez funkcję konstruktora `Employee`.

Listing 11.5. Dodawanie aliasu typu w kodzie pliku `index.ts` w katalogu `src`

```

type Person = {
    id: string,
    name: string,
    city: string
}

```

```

};

type Employee = {
  id: string,
  name: string,
  dept: string,
  city: string,
  writeDept: () => void
};

let Employee = function(id: string, name: string, dept: string, city: string) {
  this.id = id;
  this.name = name;
  this.dept = dept;
  this.city = city;
};

Employee.prototype.writeDept = function() {
  console.log(`${this.name} pracuje w dziale ${this.dept}`);
};

let salesEmployee = new Employee("fvega", "Fidel Vega", "sprzedaży", "Paryż");

let data: (Person | Employee)[] =
  [{ id: "bnowak", name: "Bartek Nowak", city: "Londyn" },
    { id: "ajanowska", name: "Alicja Janowska", city: "Paryż"},
    { id: "dpetecka", name: "Dorota Petecka", city: "Nowy Jork"},
    salesEmployee];

data.forEach(item => {
  if ("dept" in item) {
    item.writeDept();
  } else {
    console.log(`${item.id} ${item.name}, ${item.city}`);
  }
});

```

Wprawdzie kompilator TypeScriptu może nie znać znaczenia funkcji konstruktora, ale ma możliwość dopasowywania obiektów tworzonych za pomocą kształtu. W kodzie na omawianym listingu został dodany kształt typu odpowiadający typowi utworzonemu przez funkcję konstruktora i zawierający m.in. metodę dostępną poprzez prototyp. Dla wygody programisty kształt typu otrzymał alias dopasowujący nazwę funkcji konstruktora, choć jest to opcjonalne, ponieważ kompilator oddzielnie monitoruje nazwy zmiennych i typów.

Zwróć uwagę na zmianę wartownika typu w kodzie na listingu 11.5 w celu zawężenia typu przez sprawdzenie pod kątem właściwości. Kompilator TypeScriptu nie ma możliwości używania operatora `instanceof` jako wartownika typu dla obiektów tworzonych przez funkcję konstruktora, więc wykorzystałem jedną z technik omówionych w rozdziale 10. Dlatego też kompilator ma możliwość dopasowania kształtu obiektów tworzonych przez funkcję konstruktora do kształtu zdefiniowanego przez typ `Employee` i rozróżniania między obiektami na podstawie istnienia właściwości `dept`. Po skompilowaniu i uruchomieniu kodu przedstawionego na listingu 11.5 zostaną wygenerowane następujące dane wyjściowe:

bnowak Bartek Nowak, Londyn
 ajanowska Alicja Janowska, Paryż
 dpetecka Dorota Petecka, Nowy Jork
 Fidel Vega pracuje w dziale sprzedaży

Używanie klas

TypeScript nie zapewnia dobrej obsługi funkcji konstruktora, ponieważ język został skoncentrowany na klasach i funkcje oferowane przez JavaScript wykorzystuje do tego, aby klasy były podobne do stosowanych w językach takich jak C#. Na listingu 11.6 pokazałem zastąpienie funkcji fabryki klasą.

Listing 11.6. *Używanie klasy w kodzie pliku index.ts w katalogu src*

```
type Person = {
  id: string,
  name: string,
  city: string
};

class Employee {
  id: string;
  name: string;
  dept: string;
  city: string;

  constructor(id: string, name: string, dept: string, city: string) {
    this.id = id;
    this.name = name;
    this.dept = dept;
    this.city = city;
  }

  writeDept() {
    console.log(`${this.name} pracuje w dziale ${this.dept}`);
  }
}

let salesEmployee = new Employee("fvega", "Fidel Vega", "sprzedaży", "Paryż");

let data: (Person | Employee)[] =
  [{ id: "bnowak", name: "Bartek Nowak", city: "Londyn" },
    { id: "ajanowska", name: "Alicja Janowska", city: "Paryż"},
    { id: "dpetecka", name: "Dorota Petecka", city: "Nowy Jork"},
    salesEmployee];

data.forEach(item => {
  if (item instanceof Employee) {
    item.writeDept();
  } else {
    console.log(`${item.id} ${item.name}, ${item.city}`);
  }
});
```

Składnia klasy w TypeScriptie wymaga deklaracji właściwości egzemplarzy i ich typów. Prowadzi to do powstawania znacznie bardziej rozbudowanych klas — choć wkrótce zaprezentuję funkcję pozwalającą skrócić kod — ale ma pewną zaletę: możliwość zdefiniowania innego typu parametru konstruktora niż typ właściwości egzemplarza, do którego jest on przypisywany. Obiekt jest tworzony na podstawie klasy za pomocą standardowego słowa kluczowego `new`, a kompilator potrafi wykorzystać słowo kluczowe `instanceof` do zawężenia typu, gdy są używane klasy.

Jak zobaczysz w kolejnych sekcjach, TypeScript oferuje potężne funkcje przeznaczone do obsługi klas, a klasa TypeScriptu może prezentować się inaczej niż standardowa klasa JavaScriptu przedstawiona w rozdziale 4. Trzeba jednak zrozumieć, że kompilator generuje klasy standardowe zależne od funkcji konstruktora JavaScriptu i funkcjonalności prototypu w trakcie działania programu. Klasę wygenerowaną przez kod przedstawiony na listingu 11.6 możesz zobaczyć, analizując znajdujący się w katalogu *dist* plik *index.js*, który będzie zawierał m.in. następujący fragment kodu:

```
...
class Employee {
  constructor(id, name, dept, city) {
    this.id = id;
    this.name = name;
    this.dept = dept;
    this.city = city;
  }
  writeDept() {
    console.log(`${this.name} pracuje w dziale ${this.dept}`);
  }
}
...
```

Gdy zaczniesz używać znacznie bardziej zaawansowanych funkcji klas, wówczas dobrze jest mieć możliwość przeanalizowania klas wygenerowanych przez kompilator, aby zobaczyć, jak funkcje TypeScriptu są konwertowane na czysty kod JavaScriptu. Po skompilowaniu i uruchomieniu kodu przedstawionego na listingu 11.6 zostaną wygenerowane następujące dane wyjściowe:

```
bnowak Bartek Nowak, Londyn
ajanowska Alicja Janowska, Paryż
dpetecka Dorota Petecka, Nowy Jork
Fidel Vega pracuje w dziale sprzedaży
```

Używanie słów kluczowych kontroli dostępu

JavaScript nie zapewnia kontroli dostępu, co oznacza, że wszystkie właściwości egzemplarza obiektu są dostępne, więc klasę — lub utworzony na jej podstawie obiekt — można łatwo zmieniać lub utworzyć zależność od implementacji funkcji. W czystym kodzie JavaScriptu konwencje nazw właściwości są używane do wskazania, które z właściwości nie będą wykorzystywane. Natomiast TypeScript idzie o krok dalej i zapewnia obsługę słów kluczowych (tabela 11.3)

pozwalających na zarządzanie dostępem do właściwości klasy. (Zapewniona jest również obsługa funkcjonalności, która prawdopodobnie będzie dodana do specyfikacji JavaScriptu — do tego tematu jeszcze powrócę w dalszej części rozdziału).

Tabela 11.3. Słowa kluczowe kontroli dostępu TypeScriptu

Słowo kluczowe	Opis
<code>public</code>	To słowo kluczowe umożliwia nieograniczony dostęp do właściwości lub metody. Taki rodzaj dostępu jest stosowany domyślnie w przypadku pominięcia słowa kluczowego kontroli dostępu
<code>private</code>	To słowo kluczowe ogranicza dostęp do klasy definiującej właściwość lub metodę, w której jest stosowane
<code>protected</code>	To słowo kluczowe ogranicza dostęp do klasy (i jej podklas) definiującej właściwość lub metodę, w której jest stosowane

W przypadku pominięcia słowa kluczowego kontroli dostępu TypeScript domyślnie stosuje słowo kluczowe `public`, choć istnieje możliwość jego jawnego stosowania, co powinno ułatwić zrozumienie przeznaczenia kodu źródłowego. Na listingu 11.7 pokazałem przykład zastosowania słów kluczowych dla właściwości definiowanych w klasie `Employee`.

Listing 11.7. Stosowanie słów kluczowych kontroli dostępu w kodzie pliku `index.ts` w katalogu `src`

```
type Person = {
  id: string,
  name: string,
  city: string
};

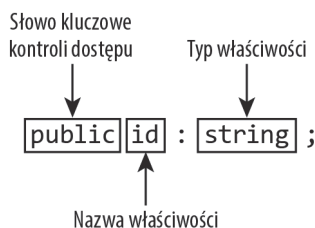
class Employee {
  public id: string;
  public name: string;
  private dept: string;
  public city: string;

  constructor(id: string, name: string, dept: string, city: string) {
    this.id = id;
    this.name = name;
    this.dept = dept;
    this.city = city;
  }

  writeDept() {
    console.log(`${this.name} pracuje w dziale ${this.dept}`);
  }
}

let salesEmployee = new Employee("fvega", "Fidel Vega", "sprzedaży", "Paryż");
console.log(`Wartość właściwości dept: ${salesEmployee.dept}`);
```

Słowa kluczowe kontroli dostępu są stosowane przed nazwą właściwości, jak pokazałem na rysunku 11.1.



Rysunek 11.1. Słowo kluczowe kontroli dostępu

W kodzie przedstawionym na listingu 11.7 słowo kluczowe `public` zastosowałem dla wszystkich właściwości egzemplarza z wyjątkiem `dept`, która została poprzedzona słowem kluczowym `private`. Efektem zastosowania słowa kluczowego `private` jest ograniczenie dostępu do klasy `Employee`, a kompilator wygeneruje następujący komunikat błędu, gdy jakiegokolwiek polecenie z zewnątrz klasy podejmie próbę odczytania wartości właściwości `dept`:

```
src/index.ts(27,42): error TS2341: Property 'dept' is private and only accessible within
↳ class 'Employee'.
```

Jedyny sposób na uzyskanie dostępu do właściwości `dept` to wykorzystanie metody `writeDept()`, jak pokazałem w kodzie na listingu 11.8. Ta metoda jest częścią klasy `Employee` i może uzyskać dostęp do właściwości oznaczonej słowem kluczowym `private`.

-
- **Ostrzeżenie** Funkcje kontroli dostępu są wymuszane przez kompilator TypeScriptu i nie trafiają do kodu JavaScriptu wygenerowanego przez kompilator. Nie opieraj się na słowach kluczowych `private` lub `protected` do ochrony danych wrażliwych, ponieważ w trakcie działania aplikacji pozostaną one dostępne dla jej pozostałej części.
-

Listing 11.8. Używanie metody w kodzie pliku `index.ts` w katalogu `src`

```
type Person = {
  id: string,
  name: string,
  city: string
};

class Employee {
  public id: string;
  public name: string;
  private dept: string;
  public city: string;

  constructor(id: string, name: string, dept: string, city: string) {
    this.id = id;
    this.name = name;
    this.dept = dept;
    this.city = city;
  }
}
```

```

    writeDept() {
        console.log(`${this.name} pracuje w dziale ${this.dept}`);
    }
}

let salesEmployee = new Employee("fvega", "Fidel Vega", "sprzedaży", "Paryż");
salesEmployee.writeDept();

```

Po skompilowaniu i uruchomieniu kodu przedstawionego na listingu 11.8 zostaną wygenerowane następujące dane wyjściowe:

```
Fidel Vega pracuje w dziale sprzedaży
```

Gwarancja inicjalizacji właściwości egzemplarza

Gdy opcja konfiguracyjna `strictPropertyInitialization` ma przypisaną wartość `true`, kompilator TypeScriptu zgłasza błąd, jeśli klasa definiuje właściwość, której nie została przypisana wartość w trakcie definicji lub przez konstruktor. Opcja `strictNullChecks` musi być włączona, aby zapewnić działanie wymienionej funkcjonalności.

Używanie właściwości prywatnych JavaScriptu

TypeScript obsługuje funkcjonalność JavaScriptu, która znajduje się obecnie w trakcie procesu standaryzacji i prawdopodobnie zostanie dodana do specyfikacji języka. Tą funkcjonalnością jest obsługa właściwości prywatnych, zapewniająca alternatywę dla słowa kluczowego `private`, jak pokazałem na listingu 11.9.

-
- **Uwaga** Zalecam stosowanie słowa kluczowego `private` TypeScriptu, przynajmniej do chwili gdy właściwość prywatna stanie się częścią specyfikacji JavaScriptu.
-

Listing 11.9. Przykład użycia właściwości prywatnej w kodzie pliku `index.ts` w katalogu `src`

```

type Person = {
    id: string,
    name: string,
    city: string
};

class Employee {
    public readonly id: string;
    public name: string;
    # dept: string;
    public city: string;

    constructor(id: string, name: string, dept: string, city: string) {
        this.id = id;
        this.name = name;
    }
}

```

```

        this.#dept = dept;
        this.city = city;
    }

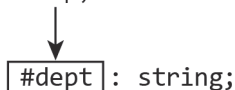
    writeDept() {
        console.log(`${this.name} pracuje w dziale ${this.#dept}`);
    }
}

let salesEmployee = new Employee("fvega", "Fidel Vega", "sprzedaży", "Paryż");
salesEmployee.writeDept();

```

Do oznaczenia właściwości prywatnej używa się znaku #, jak pokazałem na rysunku 11.2.

Właściwość prywatna



```

#dept: string;

```

Rysunek 11.2. Właściwość prywatna

Dodanie prefiksu w postaci znaku # do nazwy zmiennej dept powoduje ograniczenie dostępu do niej jedynie do klasy, w której została zdefiniowana. Znak # jest również wymagany podczas przypisywania wartości właściwości prywatnej lub pobierania jej wartości, jak pokazałem w kolejnym fragmencie kodu:

```

...
this.#dept = dept;
...

```

W porównaniu do słowa kluczowego private TypeScriptu kluczową zaletą znaku # jest to, że pozostaje on w kodzie podczas kompilacji, a tym samym w środowisku uruchomieniowym JavaScriptu wymuszane jest zastosowanie kontroli dostępu. Podobnie jak większość funkcjonalności TypeScriptu, także słowo kluczowe private nie jest umieszczane w kodzie JavaScript wygenerowanym przez kompilator, więc w kodzie JavaScript kontrola dostępu nie będzie wymuszana.

Właściwość prywatna w JavaScriptcie ma również pewne ograniczenia. Elementem prywatnym może być tylko właściwość — nie istnieją metody prywatne. Wprawdzie istnieje propozycja dodania metod prywatnych do specyfikacji JavaScriptu, ale ta funkcjonalność nie jest obsługiwana przez kompilator TypeScriptu. Skoro ta funkcjonalność nie jest częścią specyfikacji języka JavaScript, kompilator TypeScriptu implementuje ją pośrednio przez dodanie kodu do wygenerowanych danych wyjściowych JavaScriptu. Jeżeli przeanalizujesz zawartość pliku *index.js* w katalogu *dist*, możesz zobaczyć, w jaki sposób kompilator obsługuje właściwości prywatne:

```

var __classPrivateFieldSet = (this && this.__classPrivateFieldSet) || function (receiver,
privateMap, value) {
    if (!privateMap.has(receiver)) {
        throw new TypeError("attempted to set private field on non-instance");
    }
    privateMap.set(receiver, value);
    return value;
};

```

```

};
var __classPrivateFieldGet = (this && this.__classPrivateFieldGet) || function (receiver,
privateMap) {
    if (!privateMap.has(receiver)) {
        throw new TypeError("attempted to get private field on non-instance");
    }
    return privateMap.get(receiver);
};
var _dept;
class Employee {
    constructor(id, name, dept, city) {
        _dept.set(this, void 0);
        this.id = id;
        this.name = name;
        __classPrivateFieldSet(this, _dept, dept);
        this.city = city;
    }
    writeDept() {
        console.log(`${this.name} pracuje w dziale ${__classPrivateFieldGet(this, _dept)}`);
    }
}
_dept = new WeakMap();
let salesEmployee = new Employee("fvega", "Fidel Vega", "sprzedaży", "Paryż");
salesEmployee.writeDept();

```

Bardzo trudno jest zaimplementować kontrolę dostępu, gdy ten mechanizm jest niedostępny w środowisku uruchomieniowym. W najlepszym przypadku wynik będzie jedynie zbliżony do oczekiwanego. Jeżeli masz pewność, że docelowe środowisko uruchomieniowe zapewnia obsługę proponowanych właściwości prywatnych, możesz je wykorzystać w skompilowanym kodzie JavaScriptu. W tym celu opcji `target` w pliku konfiguracyjnym `tsconfig.json` przypisz wartość `ESNext`, jak pokazałem na listingu 11.10.

Listing 11.10. Zmiana wersji docelowej JavaScriptu w pliku konfiguracyjnym `tsconfig.json` w katalogu `types`

```

{
  "compilerOptions": {
    "target": "esNext",
    "outDir": "./dist",
    "rootDir": "./src",
    "declaration": true,
    // "strictNullChecks": true,
    // "suppressExcessPropertyErrors": true
  }
}

```

Żadna opcja konfiguracyjna nie włącza jedynie funkcjonalności właściwości prywatnych, więc musisz się upewnić, że środowisko uruchomieniowe JavaScriptu będzie zapewniało obsługę wszystkich funkcji dostarczanych przez wersję określaną jako `ESNext`. Konkretna funkcjonalność wersji `ESNext` ulega zmianie w zależności od wydania języka. Po skompilowaniu pliku TypeScriptu właściwości prywatne zostaną przekazane bezpośrednio i zostanie wygenerowany kod w następującej postaci:

```
class Employee {
    constructor(id, name, dept, city) {
        this.id = id;
        this.name = name;
        this.#dept = dept;
        this.city = city;
    }
    #dept;
    writeDept() {
        console.log(`${this.name} pracuje w dziale ${this.#dept}`);
    }
}

let salesEmployee = new Employee("fvega", "Fidel Vega", "sprzedaży", "Paryż");
salesEmployee.writeDept();
```

Definiowanie właściwości tylko do odczytu

Słowo kluczowe `readonly` może być używane do tworzenia właściwości egzemplarza, której wartość jest przypisywana przez konstruktor i nie może być zmieniona, jak pokazałem na listingu 11.11.

Listing 11.11. Tworzenie właściwości tylko do odczytu w kodzie pliku `index.ts` w katalogu `src`

```
type Person = {
    id: string,
    name: string,
    city: string
};

class Employee {
    public readonly id: string;
    public name: string;
    #dept: string;
    public city: string;

    constructor(id: string, name: string, dept: string, city: string) {
        this.id = id;
        this.name = name;
        this.#dept = dept;
        this.city = city;
    }

    writeDept() {
        console.log(`${this.name} pracuje w dziale ${this.#dept}`);
    }
}

let salesEmployee = new Employee("fvega", "Fidel Vega", "sprzedaży", "Paryż");
salesEmployee.writeDept();
salesEmployee.id = "fidel";
```

Słowo kluczowe `readonly` musi być umieszczone po słowie kluczowym kontroli dostępu, o ile takie jest używane, jak pokazałem na rysunku 11.3.

Słowo kluczowe
↓

```
public readonly id: string;
```

Rysunek 11.3. Właściwość tylko do odczytu

Zastosowanie słowa kluczowego `readonly` dla właściwości `id` na listingu 11.11 oznacza brak możliwości zmiany wartości przypisanej przez konstruktor. Polecenie, które spróbuje przypisać nową wartość właściwości `id`, spowoduje wygenerowanie następującego komunikatu błędu:

```
src/index.ts(27,15): error TS2540: Cannot assign to 'id' because it is a read-only property.
```

-
- **Ostrzeżenie** Słowo kluczowe `readonly` jest wymuszane przez kompilator TypeScriptu i nie ma wpływu na kod JavaScriptu generowany przez kompilator. Nie używaj tej funkcjonalności do ochrony operacji lub danych wrażliwych.
-

Upraszczenie klasy konstruktora

Czyste klasy JavaScriptu używają konstruktorów dynamicznie tworzących właściwości egzemplarza, natomiast TypeScript wymaga wyraźnego zdefiniowania właściwości. Podejście stosowane przez TypeScript jest jednym z najlepiej znanych programistom, choć jednocześnie może być rozwlekłe i powtarzające się, zwłaszcza jeśli większość parametrów konstruktora jest przypisywana właściwościom o takiej samej nazwie. TypeScript obsługuje zwięźlejszą składnię konstruktora pozwalającą na uniknięcie wzorca „definiuj i przypisuj”, jak pokazałem w przykładzie na listingu 11.12.

Listing 11.12. Uproszczenie konstruktora w kodzie pliku `index.ts` w katalogu `src`

```
type Person = {
  id: string,
  name: string,
  city: string
};

class Employee {
  constructor(public readonly id: string, public name: string,
    private dept: string, public city: string) {
    // Polecenia nie są wymagane.
  }

  writeDept() {
    console.log(`${this.name} pracuje w dziale ${this.dept}`);
  }
}
```

```
let salesEmployee = new Employee("fvega", "Fidel Vega", "sprzedaży", "Paryż");
salesEmployee.writeDept();
//salesEmployee.id = "fidel";
```

W celu uproszczenia konstruktora słowa kluczowe związane z kontrolą dostępu są stosowane dla parametrów, jak pokazałem na rysunku 11.4.

Słowo kluczowe kontroli dostępu



```
constructor(public readonly id: string, public name: string,
    private dept: string, public city: string) {
```

Rysunek 11.4. Stosowanie słów kluczowych kontroli dostępu dla parametrów konstruktora

Kompilator automatycznie tworzy właściwość egzemplarza dla każdego argumentu konstruktora, dla którego zostało zastosowane słowo kluczowe kontroli dostępu, i przypisuje wartość parametru. Stosowanie słów kluczowych kontroli dostępu nie zmienia sposobu wywoływania konstruktora i jest konieczne tylko po to, aby wskazać kompilatorowi, że podane zmienne egzemplarza są wymagane. Jeżeli zachodzi potrzeba, to spójną składnię można łączyć z parametrami konwencjonalnymi, a słowo kluczowe `readonly` jest przekazywane właściwości egzemplarza utworzonej przez kompilator. Kod przedstawiony na listingu 11.12 powoduje wygenerowanie następujących danych wyjściowych:

```
Fidel Vega pracuje w dziale sprzedaży
```

Używanie dziedziczenia klas

TypeScript opiera się na standardowej funkcjonalności dziedziczenia klas, które dzięki temu pozostają bardziej spójne, oraz stosuje pewne użyteczne dodatki dla najczęściej wykonywanych zadań. Ponadto ogranicza te cechy charakterystyczne JavaScriptu, które mogą spowodować problemy. Na listingu 11.13 pokazałem zastąpienie aliasu typu `Person` klasą dostarczającą taką samą funkcjonalność i używaną jako klasa nadrzędna dla `Employee`.

-
- **Uwaga** Wprawdzie w jednym pliku kodu źródłowego umieściłem wiele klas, ale powszechnie stosowaną praktyką jest definiowanie poszczególnych klas w oddzielnych plikach, co ułatwia poruszanie się po projekcie. Bardziej rzeczywiste przykłady przedstawię w trzeciej części książki, w której zajmę się tworzeniem serii aplikacji internetowych.
-

Listing 11.13. Dodawanie klasy w kodzie pliku `index.ts` w katalogu `src`

```
class Person {

    constructor(public id: string, public name: string,
        public city: string) { }
}
```



```

class Employee extends Person {
    constructor(public readonly id: string, public name: string,
        private dept: string, public city: string) {
        super(id, name, city);
    }

    writeDept() {
        console.log(`${this.name} pracuje w dziale ${this.dept}`);
    }
}

let data = [new Person("bnowak", "Bartek Nowak", "Londyn"),
    new Employee("fvega", "Fidel Vega", "sprzedaży", "Paryż")];

data.forEach(item => {
    console.log(`Person: ${item.name}, ${item.city}`);
    if (item instanceof Employee) {
        item.writeDept();
    }
});

```

Gdy używane jest słowo kluczowe `extends`, TypeScript wymaga wywołania konstruktora klasy nadrzędnej za pomocą słowa kluczowego `super`, co gwarantuje zainicjalizowanie jego właściwości. Po skompilowaniu i uruchomieniu kodu przedstawionego na listingu 11.13 zostaną wygenerowane następujące dane wyjściowe:

```

Person: Bartek Nowak, Londyn
Person: Fidel Vega, Paryż
Fidel Vega pracuje w dziale sprzedaży

```

Automatyczne określanie typu podklasy

Należy zachować ostrożność podczas zezwalania kompilatorowi na określanie typu na podstawie klas, ponieważ bardzo łatwo można doprowadzić do nieoczekiwanych wyników, np. przez przyjęcie założenia, że kompilator ma wgląd w hierarchię klas.

Zdefiniowana na listingu 11.13 tablica danych zawiera obiekty `Person` i `Employee`, a jeśli przeanalizujesz plik *index.d.ts* w katalogu *dist*, wówczas zobaczysz, że kompilator określił `Person[]` jako typ tablicy:

```

...
declare let data: Person[];
...

```

Jeśli masz doświadczenie w programowaniu za pomocą innych języków, możesz przyjąć założenie, że kompilator określił `Employee` jako podklasę `Person`, a wszystkie obiekty tablicy mogą być traktowane jako obiekty typu `Person`. W rzeczywistości kompilator tworzy unię typów znajdujących się w tablicy, `Person | Employee`, i ustala, że to jest odpowiednik typu `Person`, ponieważ unia przedstawia jedynie funkcjonalność istniejącą we wszystkich typach. Trzeba koniecznie pamiętać, że kompilator zwraca uwagę na kształt obiektu, nawet jeśli programista przykładą wagę do klas. Może się to wydawać mało istotną różnicą, ale ma konsekwencje podczas używania obiektów współdzielących tę samą klasę nadrzędną, jak pokazałem na listingu 11.14.

Listing 11.14. Używanie obiektów ze wspólną klasą nadrzędną w kodzie pliku *index.ts* w katalogu *src*

```
class Person {
    constructor(public id: string, public name: string,
        public city: string) { }
}

class Employee extends Person {
    constructor(public readonly id: string, public name: string,
        private dept: string, public city: string) {
        super(id, name, city);
    }

    writeDept() {
        console.log(`${this.name} pracuje w dziale ${this.dept}`);
    }
}

class Customer extends Person {
    constructor(public readonly id: string, public name: string,
        public city: string, public creditLimit: number) {
        super(id, name, city);
    }
}

class Supplier extends Person {
    constructor(public readonly id: string, public name: string,
        public city: string, public companyName: string) {
        super(id, name, city);
    }
}

let data = [new Employee("fvega", "Fidel Vega", "sprzedaży", "Paryż"),
    new Customer("ajanowska", "Alicja Janowska", "Londyn", 500)];

data.push(new Supplier("dpetecka", "Dorota Petecka", "Nowy Jork", "Acme"));

data.forEach(item => {
    console.log(`Person: ${item.name}, ${item.city}`);
    if (item instanceof Employee) {
        item.writeDept();
    } else if (item instanceof Customer) {
        console.log(`Klient ${item.name} ma limit ${item.creditLimit}`);
    } else if (item instanceof Supplier) {
        console.log(`Dostawca ${item.name} pracuje dla firmy ${item.companyName}`);
    }
});
```

Ten przykładowy fragment kodu nie zostanie skompilowany, ponieważ kompilator TypeScriptu określił typ tablicy *data* na podstawie znajdujących się w niej obiektów i nie odzwierciedlił współdzielonej klasy nadrzędnej. Oto polecenie pochodzące z pliku *index.d.ts* w katalogu *dist* i pokazujące typ ustalony przez kompilator:

```
...
declare let data: (Employee | Customer)[];
...
```

Tablica może zawierać jedynie obiekty `Employee` i `Customer`, a błędy zostały zgłoszone z powodu dodania obiektu typu `Supplier`. Aby rozwiązać ten problem, można skorzystać z adnotacji typu i wskazać kompilatorowi, że tablica może zawierać obiekty typu `Product`, jak pokazałem na listingu 11.15.

Listing 11.15. *Używanie adnotacji typu w kodzie pliku `index.ts` w katalogu `src`*

```
...
let data: Person[] = [new Employee("fvega", "Fidel Vega", "sprzedaży", "Paryż"),
    new Customer("ajanowska", "Alicja Janowska", "Londyn", 500)];

data.push(new Supplier("dpetcka", "Dorota Petecka", "Nowy Jork", "Acme"));
...
```

Teraz kompilator pozwoli, aby tablica przechowywała obiekty typu `Product` i inne utworzone na podstawie jej podklasy. Po wprowadzeniu w pliku `index.ts` zmian przedstawionych na listingu 11.15, a następnie po skompilowaniu i uruchomieniu aplikacji zostaną wygenerowane następujące dane wyjściowe:

```
Person: Fidel Vega, Paryż
Fidel Vega pracuje w dziale sprzedaży
Person: Alicja Janowska, Londyn
Klient Alicja Janowska ma limit 500
Person: Dorota Petecka, Nowy Jork
Dostawca Dorota Petecka pracuje dla firmy Acme
```

Używanie klasy abstrakcyjnej

Nie można bezpośrednio utworzyć egzemplarza klasy abstrakcyjnej, ponieważ jest ona używana do opisanie wspólnej funkcjonalności koniecznej do zaimplementowania przez podklasy. W ten sposób podklasy są zmuszane do zastosowania określonego kształtu, choć jednocześnie mogą mieć własne implementacje metod charakterystyczne dla danej klasy. Spójrz na listing 11.16.

Listing 11.16. *Definiowanie klasy abstrakcyjnej w kodzie pliku `index.ts` w katalogu `src`*

```
abstract class Person {

    constructor(public id: string, public name: string,
        public city: string) { }

    getDetails(): string {
        return `${this.name}, ${this.getSpecificDetails()}`;
    }
    abstract getSpecificDetails(): string;
}
```

```
class Employee extends Person {
    constructor(public readonly id: string, public name: string,
        private dept: string, public city: string) {
        super(id, name, city);
    }
    getSpecificDetails() {
        return `pracuje w dziale ${this.dept}`;
    }
}

class Customer extends Person {
    constructor(public readonly id: string, public name: string,
        public city: string, public creditLimit: number) {
        super(id, name, city);
    }

    getSpecificDetails() {
        return `ma limit ${this.creditLimit}`;
    }
}

class Supplier extends Person {
    constructor(public readonly id: string, public name: string,
        public city: string, public companyName: string) {
        super(id, name, city);
    }

    getSpecificDetails() {
        return `pracuje dla firmy ${this.companyName}`;
    }
}

let data: Person[] = [new Employee("fvega", "Fidel Vega", "sprzedaży", "Paryż"),
    new Customer("ajanowska", "Alicja Janowska", "Londyn", 500)];
data.push(new Supplier("dpetecka", "Dorota Petecka", "Nowy Jork", "Acme"));

data.forEach(item => console.log(item.getDetails()));
```

Klasa abstrakcyjna jest tworzona za pomocą słowa kluczowego `abstract` umieszczonego przed słowem kluczowym `class`, jak pokazałem na rysunku 11.5.

Słowo kluczowe
↓
`abstract` class Person {

Rysunek 11.5. Definiowanie klasy abstrakcyjnej

Słowo kluczowe `abstract` jest również stosowane dla poszczególnych metod, które są deklarowane bez definicji, jak pokazałem na rysunku 11.6.

Słowo kluczowe
↓
`abstract` `getSpecificDetails(): string;`

Rysunek 11.6. Definiowanie metody abstrakcyjnej

Gdy klasa rozszerza klasę abstrakcyjną, musi implementować wszystkie metody abstrakcyjne. W omawianym przykładzie klasa abstrakcyjna `Person` definiuje metodę abstrakcyjną o nazwie `getSpecificDetails()`, która musi być implementowana przez klasy `Employee`, `Customer` i `Supplier`. Klasa `Person` definiuje również zwykłą metodę o nazwie `getDetails()` wywołującą metodę abstrakcyjną i używającą wyniku jej wykonania.

Obiekt utworzony na podstawie klasy wywodzącej się z klasy abstrakcyjnej może być używany poprzez typ klasy abstrakcyjnej, co oznacza, że obiekty `Employee`, `Customer` i `Supplier` mogą być przechowywane w tablicy `Person`, choć tylko właściwości i metody definiowane przez klasę `Person` będą możliwe do użycia, o ile obiekt nie zostanie zawężony do bardziej dokładnie określonego typu. Kod przedstawiony na listingu 11.16 powoduje wygenerowanie następujących danych wyjściowych:

```
Fidel Vega pracuje w dziale sprzedaży
Alicja Janowska ma limit 500
Dorota Petecka pracuje dla firmy Acme
```

Wartownik typu klasy abstrakcyjnej

Klasa abstrakcyjna jest w kodzie JavaScriptu wygenerowanym przez kompilator TypeScriptu implementowana jako zwykła klasa. Wadą takiego podejścia jest to, że kompilator TypeScriptu uniemożliwia tworzenie egzemplarzy klas abstrakcyjnych, choć takie ograniczenie nie jest egzekwowane za pomocą kodu JavaScriptu, więc potencjalnie może dojść do utworzenia obiektu na podstawie klasy abstrakcyjnej. Jednak takie podejście oznacza również możliwość użycia słowa kluczowego `instanceof` do zawężenia typów, jak pokazałem na listingu 11.17.

Listing 11.17. Stosowanie wartownika typu w klasie abstrakcyjnej w kodzie pliku `index.ts` w katalogu `src`

```
abstract class Person {
    constructor(public id: string, public name: string,
                public city: string) { }

    getDetails(): string {
        return `${this.name}, ${this.getSpecificDetails()}`;
    }

    abstract getSpecificDetails(): string;
}

class Employee extends Person {
    constructor(public readonly id: string, public name: string,
                private dept: string, public city: string) {
```

```

        super(id, name, city);
    }

    getSpecificDetails() {
        return `pracuje w dziale ${this.dept}`;
    }
}

class Customer {

    constructor(public readonly id: string, public name: string,
                public city: string, public creditLimit: number) {
    }

}

let data: (Person | Customer)[] = [
    new Employee("fvega", "Fidel Vega", "sprzedaży", "Paryż"),
    new Customer("ajanowska", "Alicja Janowska", "Londyn", 500)];

data.forEach(item => {
    if (item instanceof Person) {
        console.log(item.getDetails());
    } else {
        console.log(`Typ Customer: ${item.name}`);
    }
});
```

Na tym listingu klasa `Employee` rozszerza klasę abstrakcyjną `Person`, natomiast klasa `Customer` już nie. Operator `instanceof` może być używany do identyfikowania wszelkich obiektów utworzonych na podstawie klasy rozszerzającej klasę abstrakcyjną, co pozwala na zawężenie unii `Person | Customer` używanej jako typ tablicy. Kod przedstawiony na listingu 11.17 powoduje wygenerowanie następujących danych wyjściowych:

```

Fidel Vega pracuje w dziale sprzedaży
Typ Customer: Alicja Janowska
```

Używanie interfejsu

Interfejs jest używany do opisanego kształtu obiektu, z którym musi być zgodna klasa implementująca dany interfejs. Przykład wykorzystania interfejsu przedstawiłem na listingu 11.18.

-
- **Uwaga** Przeznaczenie interfejsu jest podobne do omówionego w rozdziale 10. kształtu typu. W kolejnych wersjach TypeScriptu granice między tymi dwiema funkcjonalnościami powoli się zacierają — w pewnym momencie można z nich korzystać zamiennie w celu osiągnięcia tego samego efektu, zwłaszcza podczas pracy z typami prostymi. Jednak interfejs oferuje pewne użyteczne funkcje i zapewnia doskonałe wrażenia podczas programowania, spójne z tymi odnoszonymi przy programowaniu w innych językach, takich jak C#.
-

Listing 11.18. Używanie interfejsu w kodzie pliku *index.ts* w katalogu *src*

```

interface Person {
    name: string;
    getDetails(): string;
}

class Employee implements Person {

    constructor(public readonly id: string, public name: string,
        private dept: string, public city: string) {
        // Polecenia nie są wymagane.
    }

    getDetails() {
        return `${this.name} pracuje w dziale ${this.dept}`;
    }
}

class Customer implements Person {

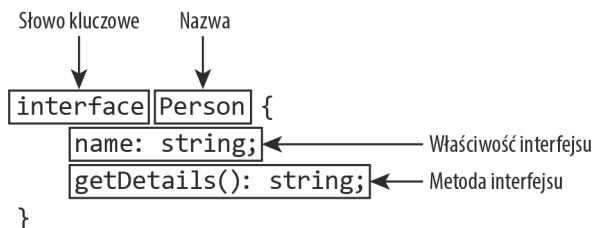
    constructor(public readonly id: string, public name: string,
        public city: string, public creditLimit: number) {
        // Polecenia nie są wymagane.
    }

    getDetails() {
        return `${this.name} ma limit ${this.creditLimit}`;
    }
}

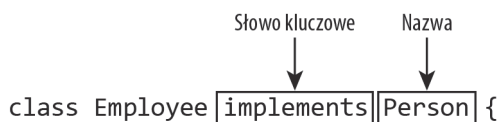
let data: Person[] = [
    new Employee("fvega", "Fidel Vega", "sprzedaży", "Paryż"),
    new Customer("ajanowska", "Alicja Janowska", "Londyn", 500)];
data.forEach(item => console.log(item.getDetails()));

```

Interfejs jest definiowany za pomocą słowa kluczowego `interface` oraz zawiera zbiór właściwości i metod, które klasa musi dostarczać, aby była zgodna z danym interfejsem, jak pokazałem na rysunku 11.7.

**Rysunek 11.7.** Definiowanie interfejsu

W przeciwieństwie do klasy abstrakcyjnej, interfejs nie implementuje metod i nie definiuje konstruktora, a jedynie wskazuje kształt. Interfejs jest implementowany przez klasę poprzez słowo kluczowe `implements`, jak pokazałem na rysunku 11.8.



Rysunek 11.8. Implementowanie interfejsu

Interfejs `Person` definiuje właściwość `name` i metodę `getDetails()`, więc klasy `Employee` i `Customer` muszą definiować tę samą właściwość i metodę. Wprawdzie wymienione klasy mogą definiować dodatkowe właściwości i metody, ale zgodność z interfejsem zapewnia tylko zdefiniowanie właściwości `name` i metody `getDetails()`. Interfejs może być używany w adnotacjach typu, np. w tablicy, jak w omawianym przykładzie.

```
...
let data: Person[] = [
    new Employee("fvega", "Fidel Vega", "sprzedaży", "Paryż"),
    new Customer("ajanowska", "Alicja Janowska", "Londyn", 500)];
...
```

Tablica `data` zawiera każdy obiekt utworzony na podstawie klasy implementującej tablicę `Product`, choć funkcja przekazana metodzie `forEach()` ma możliwość uzyskania dostępu jedynie do funkcjonalności definiowanej przez interfejs, o ile obiekty nie zostaną zawężone do znacznie dokładniej określonego typu. Kod przedstawiony na listingu 11.18 powoduje wygenerowanie następujących danych wyjściowych:

```
Fidel Vega pracuje w dziale sprzedaży
Alicja Janowska ma limit 500
```

Łączenie deklaracji interfejsów

Interfejs może być zdefiniowany w wielu deklaracjach `interface`, które są następnie łączone przez kompilator i tworzą pojedynczy interfejs. To dziwna funkcja, dla której jeszcze nie znalazłem użytecznych zastosowań w moich projektach. Deklaracje muszą znajdować się w tym samym pliku kodu, a także muszą być wyeksportowane (czyli zdefiniowane za pomocą słowa kluczowego `export`) lub zdefiniowane lokalnie (czyli bez wymienionego słowa kluczowego).

Implementowanie wielu interfejsów

Klasa może implementować więcej niż tylko jeden interfejs, co oznacza konieczność zdefiniowania metod i właściwości znajdujących się we wszystkich interfejsach, jak pokazałem na listingu 11.19.

Listing 11.19. Implementowanie wielu interfejsów w kodzie pliku *index.ts* w katalogu *src*

```

interface Person {
    name: string;
    getDetails(): string;
}

interface DogOwner {
    dogName: string;
    getDogDetails(): string;
}

class Employee implements Person {

    constructor(public readonly id: string, public name: string,
        private dept: string, public city: string) {
        // Polecenia nie są wymagane.
    }

    getDetails() {
        return `${this.name} pracuje w dziale ${this.dept}`;
    }
}

class Customer implements Person, DogOwner {

    constructor(public readonly id: string, public name: string,
        public city: string, public creditLimit: number,
        public dogName ) {
        // Polecenia nie są wymagane.
    }

    getDetails() {
        return `${this.name} ma limit ${this.creditLimit}`;
    }

    getDogDetails() {
        return `${this.name} ma psa o imieniu ${this.dogName}`;
    }
}

let alice = new Customer("ajanowska", "Alicja Janowska", "Londyn", 500, "Fido");

let dogOwners: DogOwner[] = [alice];
dogOwners.forEach(item => console.log(item.getDogDetails()));

let data: Person[] = [new Employee("fvega", "Fidel Vega", "sprzedaży", "Paryż"), alice];
data.forEach(item => console.log(item.getDetails()));

```

Rozdzielone przecinkami nazwy interfejsów są wymienione po słowie kluczowym `implements`. W kodzie przedstawionym na omawianym listingu klasa `Customer` implementuje interfejsy `Person` i `DogOwner`, co oznacza przypisanie obiektu `Person` zmiennej `alice`, która następnie będzie mogła zostać dodana do tablicy przechowującej obiekty typów `Person` i `DogOwner`. Kod przedstawiony na listingu 11.19 powoduje wygenerowanie następujących danych wyjściowych:

```
Alicja Janowska ma psa o imieniu Fido
Fidel Vega pracuje w dziale sprzedaży
Alicja Janowska ma limit 500
```

-
- **Uwaga** Klasa może implementować wiele interfejsów tylko wtedy, gdy nie istnieją nakładające się właściwości o niezgodnych ze sobą typach. Na przykład jeśli interfejs `Person` zawiera właściwość `id` typu `string` i jeśli interfejs `DogOwner` zawiera właściwość o takiej samej nazwie, ale o typie `number`, wówczas klasa `Customer` nie będzie w stanie implementować obu interfejsów, ponieważ nie istnieje wartość możliwa do przypisania właściwości `id` i przedstawiająca oba wymienione wcześniej typy.
-

Rozszerzanie interfejsu

Podobnie jak klasę, także interfejs można rozszerzyć. Stosowane jest takie samo ogólne podejście, a wynikiem jest interfejs zawierający właściwości i metody dziedziczone po jego interfejsach nadrzędnych, a także nowa funkcjonalność zdefiniowana w interfejsie. Spójrz na przykład przedstawiony na listingu 11.20.

Listing 11.20. Rozszerzanie interfejsu w kodzie pliku `index.ts` w katalogu `src`

```
interface Person {
    name: string;
    getDetails(): string;
}

interface DogOwner extends Person {
    dogName: string;
    getDogDetails(): string;
}

class Employee implements Person {
    constructor(public readonly id: string, public name: string,
        private dept: string, public city: string) {
        // Polecenia nie są wymagane.
    }

    getDetails() {
        return `${this.name} pracuje w dziale ${this.dept}`;
    }
}

class Customer implements DogOwner {
    constructor(public readonly id: string, public name: string,
        public city: string, public creditLimit: number,
        public dogName ) {
        // Polecenia nie są wymagane.
    }
}
```

```

    getDetails() {
        return `${this.name} ma limit ${this.creditLimit}`;
    }

    getDogDetails() {
        return `${this.name} ma psa o imieniu ${this.dogName}`;
    }
}

let alice = new Customer("ajanowska", "Alicja Janowska", "Londyn", 500, "Fido");

let dogOwners: DogOwner[] = [alice];
dogOwners.forEach(item => console.log(item.getDogDetails()));

let data: Person[] = [new Employee("fvega", "Fidel Vega", "sprzedaży", "Paryż"), alice];
data.forEach(item => console.log(item.getDetails()));

```

Słowo kluczowe `extend` zostało użyte w celu rozszerzenia interfejsu. W omawianym przykładzie interfejs `DogOwner` rozszerza interfejs `Person`, więc klasa implementująca `DogOwner` musi definiować właściwości i metody obu interfejsów. Obiekt utworzony na podstawie klasy `Customer` może być traktowany albo jako obiekt typu `Customer`, albo `DogOwner`, ponieważ wymienione typy zawsze definiują kształt wymagany przez poszczególne interfejsy. Kod przedstawiony na listingu 11.20 powoduje wygenerowanie następujących danych wyjściowych:

```

Alicja Janowska ma psa o imieniu Fido
Fidel Vega pracuje w dziale sprzedaży
Alicja Janowska ma limit 500

```

Interfejs i kształt typu

Jak wcześniej wspomniałem, kształty typu i interfejsy bardzo często mogą być stosowane zamiennie. Na przykład klasa może za pomocą słowa kluczowego `implements` i kształtu typu wskazywać właściwości implementowane w kształcie:

```

...
type Person = {
    name: string;
    getDetails(): string;
};

class Employee implements Person {
    constructor(public readonly id: string, public name: string,
        private dept: string, public city: string) {
        // Polecenia nie są wymagane.
    }

    getDetails() {
        return `${this.name} pracuje w dziale ${this.dept}`;
    }
}
...

```

Ten fragment kodu został oparty na listingu 11.20 i zastępuje interfejs `Person` kształtem typu o takich samych właściwościach. Klasa `Employee` używa słowa kluczowego `implements` do zadeklarowania zgodności z kształtem `Person`.

Interfejs może również zapewnić zgodność z kształtem typu, używając słowa kluczowego `extends`, jak pokazałem w kolejnym fragmencie kodu:

```
...
type NamedObject = {
  name: string;
};

interface Person extends NamedObject {
  getDetails(): string;
};
...
```

Tutaj interfejs `Person` dziedziczy właściwość `name` po kształcie typu `NamedObject`. Klasa implementująca interfejs `Person` musi definiować bezpośrednio wskazane przez interfejs właściwość `name` i metodę `getDetails()`.

Definiowanie opcjonalnych właściwości i metod interfejsu

Dodanie właściwości opcjonalnej do interfejsu pozwala implementującej go klasie na dostarczenie właściwości, która nie będzie wymagana. Przykład takiego rozwiązania przedstawiłem na listingu 11.21.

Listing 11.21. Dodawanie właściwości opcjonalnej w kodzie pliku `index.ts` w katalogu `src`

```
interface Person {
  name: string;
  getDetails(): string;

  dogName?: string;
  getDogDetails?(): string;
}

class Employee implements Person {
  constructor(public readonly id: string, public name: string,
    private dept: string, public city: string) {
    // Polecenia nie są wymagane.
  }

  getDetails() {
    return `${this.name} pracuje w dziale ${this.dept}`;
  }
}

class Customer implements Person {
  constructor(public readonly id: string, public name: string,
    public city: string, public creditLimit: number,
    public dogName) {
```

```

    // Polecenia nie są wymagane.
}

getDetails() {
    return `${this.name} ma limit ${this.creditLimit}`;
}

getDogDetails() {
    return `${this.name} ma psa o imieniu ${this.dogName}`;
}
}

let alice = new Customer("ajanowska", "Alicja Janowska", "Londyn", 500, "Fido");
let data: Person[] = [new Employee("fvega", "Fidel Vega", "sprzedaży", "Paryż"), alice];
data.forEach(item => {
    console.log(item.getDetails());
    if (item.getDogDetails) {
        console.log(item.getDogDetails());
    }
});

```

Zadeklarowanie opcjonalnej właściwości interfejsu odbywa się za pomocą znaku zapytania umieszczonego po nazwie, jak pokazałem na rysunku 11.9.

```

interface Person {
    name: string;
    getDetails(): string;

    dogName?: string;
    getDogDetails?:(): string;
}

```

Rysunek 11.9. Definiowanie opcjonalnych elementów składowych interfejsu

Opcjonalna funkcjonalność interfejsu może być zdefiniowana za pomocą typu interfejsu i nie spowoduje błędów kompilatora. Musisz się jednak upewnić, że nie otrzymasz wartości `undefined`, ponieważ obiekt mógł zostać utworzony na podstawie klasy, która nie implementuje danej funkcjonalności.

```

...
data.forEach(item => {
    console.log(item.getDetails());
    if (item.getDogDetails) {
        console.log(item.getDogDetails());
    }
});
...

```

Tylko jeden typ w kodzie na listingu 11.21 implementujący interfejs `Person` definiuje metodę `getDogDetails()`. Dostęp do tej metody można uzyskać poprzez typ `Person`, bez zawężania do konkretnej klasy. Jednak ta metoda mogła zostać niezdefiniowana, stąd zastosowanie koercji

typu w wyrażeniu warunkowym, aby wywołanie metody odbywało się tylko w obiekcie, który ją zdefiniował. Kod przedstawiony na listingu 11.21 powoduje wygenerowanie następujących danych wyjściowych:

```
Fidel Vega pracuje w dziale sprzedaży
Alicja Janowska ma limit 500
Alicja Janowska ma psa o imieniu Fido
```

Definiowanie implementacji interfejsu abstrakcyjnego

Klasa abstrakcyjna może być używana do implementacji wybranej lub całej funkcjonalności opisywanej przez interfejs, jak pokazałem na listingu 11.22. Może to zmniejszyć ilość powielanego kodu, gdy klasa implementująca interfejs będzie implementować funkcjonalność w ten sam sposób i za pomocą tego samego kodu.

Listing 11.22. Tworzenie implementacji abstrakcyjnej w kodzie pliku *index.ts* w katalogu *src*

```
interface Person {
    name: string;
    getDetails(): string;

    dogName?: string;
    getDogDetails?(): string;
}

abstract class AbstractDogOwner implements Person {

    abstract name: string;
    abstract dogName?: string;

    abstract getDetails();

    getDogDetails() {
        if (this.dogName) {
            return `${this.name} ma psa o imieniu ${this.dogName}`;
        }
    }
}

class DogOwningCustomer extends AbstractDogOwner {

    constructor(public readonly id: string, public name: string,
                public city: string, public creditLimit: number,
                public dogName) {
        super();
    }

    getDetails() {
        return `${this.name} ma limit ${this.creditLimit}`;
    }
}
```

```
let alice = new DogOwningCustomer("ajanowska", "Alicja Janowska", "Londyn", 500, "Fido");
if (alice.getDogDetails()) {
  console.log(alice.getDogDetails());
}
```

`AbstractDogOwner` dostarcza częściową implementację interfejsu `Person` i deklaruje funkcjonalność interfejsu nieimplementowaną jako abstrakcyjna, co wymusza na podklasach implementowanie tych funkcjonalności. Istnieje tylko jedna podklasa rozszerzająca `AbstractDogOwner` i dziedzicząca metodę `getDogDetails()` po klasie abstrakcyjnej. Kod przedstawiony na listingu 11.22 powoduje wygenerowanie następujących danych wyjściowych:

```
Alicja Janowska ma psa o imieniu Fido
```

Wartownik typu interfejsu

Nie istnieje odpowiednik JavaScriptu dla interfejsu, więc żadne szczegóły z interfejsem nie są umieszczane w kodzie JavaScriptu wygenerowanym przez kompilator TypeScriptu. Oznacza to brak możliwości użycia słowa kluczowego `instanceof` do zawężania typów interfejsu, a wartownik typu może jedynie sprawdzać pod kątem jednej lub więcej właściwości definiowanych przez interfejs, jak pokazałem w przykładzie na listingu 11.23.

Listing 11.23. Używanie wartownika typu w kodzie pliku `index.ts` w katalogu `src`

```
interface Person {
  name: string;
  getDetails(): string;
}

interface Product {
  name: string;
  price: number;
}

class Employee implements Person {
  constructor(public name: string, public company: string) {
    // Polecenia nie są wymagane.
  }

  getDetails() {
    return `${this.name} pracuje dla firmy ${this.company}`;
  }
}

class SportsProduct implements Product {
  constructor(public name: string, public category: string,
    public price: number) {
    // Polecenia nie są wymagane.
  }
}

let data: (Person | Product)[] = [new Employee("Bartek Nowak", "Acme"),
```

```

    new SportsProduct("Buty do biegania", "bieganie", 90.50),
    new Employee("Dorota Petecka", "BigCo")]);

data.forEach(item => {
  if ("getDetails" in item) {
    console.log(`Typ Person: ${item.getDetails()}`);
  } else {
    console.log(`Typ Product: ${item.name}, ${item.price}`);
  }
});

```

Kod na tym listingu wykorzystuje obecność właściwości `getDetails` do zidentyfikowania obiektów implementujących interfejs `Person`, a to pozwala na zawężenie zawartości tablicy `data` do typu `Person` lub `Product`. Kod przedstawiony na listingu 11.23 powoduje wygenerowanie następujących danych wyjściowych:

```

Typ Person: Bartek Nowak pracuje dla firmy Acme
Typ Product: Buty do biegania, 90.5
Typ Person: Dorota Petecka pracuje dla firmy BigCo

```

Dynamiczne tworzenie właściwości

Kompilator TypeScriptu pozwala na przypisywanie wartości tylko właściwościom będącym częścią typu obiektu. Dlatego też to interfejsy i klasy muszą definiować wszystkie właściwości wymagane przez aplikację.

Z kolei JavaScript pozwala na tworzenie nowych właściwości w obiekcie przez przypisanie wartości nieużywanym wcześniej nazwom. *Sygnatura indeksu* w TypeScriptie zapewnia rodzaj pomostu między tymi dwoma modelami oraz pozwala na dynamiczne tworzenie właściwości i zachowanie bezpieczeństwa typu, jak pokazałem na listingu 11.24.

Listing 11.24. Definiowanie sygnatury indeksu w kodzie pliku `index.ts` w katalogu `src`

```

interface Product {
  name: string;
  price: number;
}

class SportsProduct implements Product {
  constructor(public name: string, public category: string,
    public price: number) {
    // Polecenia nie są wymagane.
  }
}

class ProductGroup {
  constructor(...initialProducts: [string, Product][]) {
    initialProducts.forEach(p => this[p[0]] = p[1]);
  }

  [propertyName: string]: Product;
}

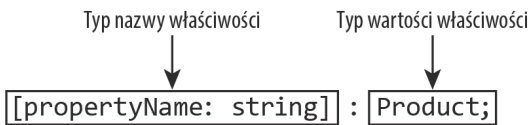
```



```
let group
  = new ProductGroup(["shoes", new SportsProduct("Buty do biegania", "bieganie", 90.50)]);
group.hat = new SportsProduct("Czapka", "narcciarstwo", 20);
Object.keys(group).forEach(k => console.log(`Nazwa właściwości: ${k}`));
```

Klasa `ProductGroup` otrzymuje tablicę krotek `[string, Product]` poprzez jej konstruktor, a każda krotka jest używana do tworzenia właściwości z nazwą typu `string` i wartością typu `Product`. Kompilator pozwala konstruktorowi na utworzenie właściwości i nadaje jej typ `any`, o ile nie zostały włączone opcje kompilatora `noImplicitAny` lub `strict`, ponieważ wtedy zostaje wygenerowany komunikat błędu.

Klasa może zdefiniować sygnaturę indeksu i pozwalać na dynamiczne tworzenie właściwości poza konstruktorem (unikając błędów kompilatora związanych z opcją `noImplicitAny`). Sygnatura indeksu używa nawiasu kwadratowego do określenia typu klucza właściwości, a następnie znajduje się adnotacja typu ograniczająca typy możliwe do wykorzystania podczas tworzenia właściwości dynamicznej, jak pokazałem na rysunku 11.10.



Rysunek 11.10. Sygnatura indeksu

Typem nazwy właściwości może być tylko `string` lub `number`, natomiast typ wartości właściwości może być dowolny. Sygnatura indeksu pokazana na rysunku wskazuje kompilatorowi, że może tworzyć właściwości dynamiczne z nazwami typu `string` i wartościami typu `Product`, jak pokazałem w kolejnym fragmencie kodu:

```
...
group.hat = new SportsProduct("Czapka", "narcciarstwo", 20);
...
```

To polecenie tworzy właściwość o nazwie `hat`. Kod przedstawiony na listingu 11.24 powoduje wygenerowanie następujących danych wyjściowych:

```
Nazwa właściwości: shoes
Nazwa właściwości: hat
```

Sprawdzanie wartości indeksu

Potencjalną wadą pracy z sygnaturą indeksu jest przyjęcie w kompilatorze TypeScriptu założenia, że chcesz uzyskać dostęp jedynie do istniejącej właściwości. Jest to sprzeczne z szerszym podejściem stosowanym w TypeScriptie i polegającym na wymuszaniu otwartych założeń, które mogą być wyraźnie zweryfikowane. Na listingu 11.25 pokazałem przykład uzyskania za pomocą sygnatury indeksu dostępu do nieistniejącej właściwości.

Listing 11.25. Przykład uzyskania dostępu do nieistniejącej właściwości w kodzie pliku *index.ts* w katalogu *src*

```
interface Product {
  name: string;
  price: number;
}

class SportsProduct implements Product {
  constructor(public name: string, public category: string,
    public price: number) {
    // Polecenia nie są wymagane.
  }
}

class ProductGroup {
  constructor(...initialProducts: [string, Product][]) {
    initialProducts.forEach(p => this[p[0]] = p[1]);
  }

  [propertyName: string]: Product;
}

let group
  = new ProductGroup(["shoes", new SportsProduct("Buty do biegania", "bieganie", 90.50)]);
group.hat = new SportsProduct("Czapka", "narcciarstwo", 20);

let total = group.hat.price + group.boots.price;
console.log(`Wartość całkowita: ${total}`);
```

Polecenie przypisujące wartość `total` używa sygnatury indeksu w celu uzyskania dostępu do właściwości `hat` i `boots`. Wprawdzie właściwość `boots` nie została utworzona, ale mimo to kod nadal będzie skompilowany, choć próba jego uruchomienia zakończy się wygenerowaniem następującego komunikatu błędu.

```
let total = group.hat.price + group.boots.price;
                                ^
TypeError: Cannot read property 'price' of undefined
    at Object.<anonymous> (C:\types\dist\index.js:16:43)
```

Kompilator można skonfigurować w celu sprawdzania dostępu za pomocą sygnatury indeksu — wystarczy przypisać wartość `true` opcji konfiguracyjnej `strictNullChecks`, jak pokazałem na listingu 11.26.

Listing 11.26. Konfiguracja kompilatora w pliku *tsconfig.json* w katalogu *types*

```
{
  "compilerOptions": {
    "target": "esNext",
    "outDir": "./dist",
    "rootDir": "./src",
    "declaration": true,
    "strictNullChecks": true,
    "noUncheckedIndexedAccess": true
  }
}
```

Po zapisaniu zmian w pliku konfiguracyjnym kompilatora kod zostanie ponownie skompilowany. Tym razem kompilator TypeScriptu wygeneruje komunikat błędu:

```
src/index.ts(25,31): error TS2532: Object is possibly 'undefined'.
```

Aby uniknąć tego błędu, przed próbą uzyskania dostępu do właściwości trzeba upewnić się o jej istnieniu, jak pokazałem na listingu 11.27. W ten sposób chronimy się przed wartością `undefined`.

Listing 11.27. Sprawdzenie istnienia właściwości w kodzie pliku *index.ts* w katalogu *src*

```
interface Product {
  name: string;
  price: number;
}

class SportsProduct implements Product {
  constructor(public name: string, public category: string,
    public price: number) {
    // Polecenia nie są wymagane.
  }
}

class ProductGroup {
  constructor(...initialProducts: [string, Product][]) {
    initialProducts.forEach(p => this[p[0]] = p[1]);
  }

  [propertyName: string]: Product;
}

let group
  = new ProductGroup(["shoes", new SportsProduct("Buty do biegania", "bieganie", 90.50)]);
group.hat = new SportsProduct("Czapka", "narciarstwo", 20);

if (group.hat && group.boots) {
  let total = group.hat.price + group.boots.price;
  console.log(`Wartość całkowita: ${total}`);
}
```

Wyrażenie `if` gwarantuje, że właściwość `boots` nie zostanie użyta, jeśli jej wartością jest `undefined`. Podejście alternatywne polega na zastosowaniu łączenia opcjonalnego i operatora koalescencyjnego `nullish` w celu dostarczenia wartości awaryjnej, jak pokazałem na listingu 11.28.

Listing 11.28. Przykład użycia wartości awaryjnej w kodzie pliku *index.ts* w katalogu *src*

```
interface Product {
  name: string;
  price: number;
}

class SportsProduct implements Product {
  constructor(public name: string, public category: string,
    public price: number) {
```

```

        // Polecenia nie są wymagane.
    }
}

class ProductGroup {
    constructor(...initialProducts: [string, Product][] ) {
        initialProducts.forEach(p => this[p[0]] = p[1]);
    }

    [propertyName: string]: Product;
}

let group
    = new ProductGroup(["shoes", new SportsProduct("Buty do biegania", "bieganie",
90.50)]);
group.hat = new SportsProduct("Czapka", "narciarstwo", 20);

let total = group.hat.price + (group.boots?.price ?? 0);
console.log(`Wartość całkowita: ${total}`);

```

Kod przedstawiony na listingu 11.28 powoduje wygenerowanie następujących danych wyjściowych:

```
Wartość całkowita: 20
```

Podsumowanie

W tym rozdziale wyjaśniłem sposoby, w jakie TypeScript usprawnia funkcjonalność klas JavaScriptu, zapewniając obsługę dla zwięzłych konstruktorów, klas abstrakcyjnych i słów kluczowych kontroli dostępu. Omówiłem również funkcjonalność w postaci interfejsów, która jest implementowana przez kompilator i oferuje alternatywny sposób na opisywanie kształtu obiektów, aby klasy mogły bez trudu być z tym kształtem zgodne. W następnym rozdziale przedstawię obsługę typów generycznych w TypeScriptie.



Używanie typów generycznych

Typy generyczne to miejsce zarezerwowane dla typów, które zostaną określone podczas wykonywania klasy lub funkcji. Pozwala to zachować bezpieczeństwo typów w kodzie utworzonym w taki sposób, aby radził sobie z obsługą wielu różnych typów, np. klas kolekcji. To koncepcja, którą znacznie łatwiej zaprezentować praktycznie, niż wyjaśnić. Dlatego też na początku rozdziału przedstawię problemy rozwiązywane przez typy generyczne, a dopiero potem przejdę do omówienia podstawowych sposobów, w jakie są używane typy generyczne. Natomiast w rozdziale 13. przedstawię zaawansowaną funkcjonalność typów generycznych dostarczaną przez TypeScript. Streszczenie materiału przedstawionego w rozdziale znajdziesz w tabeli 12.1.

Tabela 12.1. Streszczenie materiału przedstawionego w rozdziale

Problem	Rozwiązanie	Listing
Definiowanie klasy lub funkcji, która może bezpiecznie obsługiwać różne typy	Zdefiniuj parametr typu generycznego	Od 5 do 7, 19 i 20
Określenie typu dla parametru typu generycznego	Użyj argumentu typu ogólnego podczas tworzenia egzemplarza klasy lub wywołaj funkcję	Od 8 do 13
Rozszerzanie klasy generycznej	Utwórz klasę, która przekazuje, ogranicza lub na stałe definiuje parametr typu generycznego dziedziczony po klasie nadrzędnej	Od 14 do 16
Stosowanie wartownika typu dla typu generycznego	Użyj funkcji predykatu typu	17 i 18
Opisywanie typu generycznego bez dostarczania implementacji	Zdefiniuj interfejs z parametrem typu generycznego	Od 21 do 25

W tabeli 12.2 wymieniałem opcje kompilatora TypeScriptu użyte w rozdziale.

Tabela 12.2. Opcje kompilatora TypeScriptu użyte w rozdziale

Opcja	Opis
declaration	Ta opcja generuje pliki deklaracji typu, które pomagają zrozumieć, w jaki sposób zostały ustalone typy dla kodu JavaScriptu. Wspomniane pliki zostaną szczegółowo omówione w rozdziale 14.
module	Ta opcja określa format używany dla modułów zgodnie z opisem przedstawionym w rozdziale 5.
outDir	Ta opcja określa katalog, w którym zostaną umieszczone wygenerowane pliki JavaScriptu
rootDir	Ta opcja określa katalog główny używany przez kompilator do wyszukiwania plików TypeScriptu
target	Ta opcja określa docelową wersję języka JavaScript, dla której kompilator wygeneruje kod

Przygotowanie projektu

W rozdziale będę kontynuował używanie projektu *types* utworzonego w rozdziale 7. i uaktualnionego w kolejnych. Aby przygotować się do tego rozdziału, należy w podkatalogu *src* projektu utworzyć plik o nazwie *dataTypes.ts* z zawartością przedstawioną na listingu 12.1.

■ **Wskazówka** Przykładowy projekt dla tego rozdziału, a także projekty dla pozostałych rozdziałów książki zostały umieszczone w archiwum dostępnym pod adresem <https://ftp.helion.pl/przyklady/tys4o2.zip>.

Listing 12.1. Zawartość pliku *dataTypes.ts* w katalogu *src*

```
export class Person {
    constructor(public name: string, public city: string) {}
}

export class Product {
    constructor(public name: string, public price: number) {}
}

export class City {
    constructor(public name: string, public population: number) {}
}

export class Employee {
    constructor(public name: string, public role: string) {}
}
```

Ponadto zawartość pliku *index.ts* w katalogu *src* trzeba zastąpić kodem przedstawionym na listingu 12.2.

Listing 12.2. Nowa zawartość pliku *index.ts* w katalogu *src*

```
import { Person, Product } from "../dataTypes";

let people = [new Person("Bartek Nowak", "Londyn"),
  new Person("Dorota Petecka", "Nowy Jork")];
let products = [new Product("Buty do biegania", 100), new Product("Czapka", 25)];

[...people, ...products].forEach(item => console.log(`Element: ${item.name}`));
```

Na listingu 12.2 polecenie `import` zostało użyte do zadeklarowania zależności od klas `Person` i `Product` zdefiniowanych w module `dataTypes`. Aby umożliwić właściwe określanie modułu, jak to omówiłem w rozdziale 5., w pliku *tsconfig.json* w katalogu *types* należy wprowadzić zmiany przedstawione na listingu 12.3.

Listing 12.3. Konfigurowanie kompilatora w pliku *tsconfig.json* w katalogu *types*

```
{
  "compilerOptions": {
    "target": "es2020",
    "outDir": "../dist",
    "rootDir": "../src",
    "declaration": true,
    // "strictNullChecks": true,
    // "noUncheckedIndexedAccess": true,
    "module": "commonjs"
  }
}
```

Otwórz nowe okno powłoki, przejdź do katalogu *types*, a następnie z jego poziomu wydaj polecenie przedstawione na listingu 12.4, uruchamiające kompilator TypeScriptu w trybie automatycznego kompilowania kodu po wykryciu zmiany w dowolnym pliku i wykonywania kodu po jego skompilowaniu.

Listing 12.4. Uruchamianie kompilatora TypeScriptu

```
$ npm start
```

Kompilator przeprowadzi kompilację kodu w pliku *index.ts*, wykona wygenerowany kod JavaScriptu, a następnie przejdzie do trybu monitorowania i wyświetli następujące dane wyjściowe:

```
7:22:32 AM - Starting compilation in watch mode...
7:22:34 AM - Found 0 errors. Watching for file changes.
Element: Bartek Nowak
Element: Dorota Petecka
Element: Buty do biegania
Element: Czapka
```

Zrozumienie problemu

Najlepszym sposobem na zrozumienie sposobu działania typów generycznych — i dostrzeżenia ich użyteczności — jest praca z często spotykanym przypadkiem, w którym zwykłe typy okazują się trudne w zarządzaniu. Na listingu 12.5 została zdefiniowana klasa przeznaczona do zarządzania kolekcją obiektów `Person`.

Listing 12.5. Definiowanie klasy w kodzie pliku `index.ts` w katalogu `src`

```
import { Person, Product } from "../dataTypes";

let people = [new Person("Bartek Nowak", "Londyn"),
  new Person("Dorota Petecka", "Nowy Jork")];
let products = [new Product("Buty do biegania", 100), new Product("Czapka", 25)];

class PeopleCollection {
  private items: Person[] = [];

  constructor(initialItems: Person[]) {
    this.items.push(...initialItems);
  }

  add(newItem: Person) {
    this.items.push(newItem);
  }

  getNames(): string[] {
    return this.items.map(item => item.name);
  }

  getItem(index: number): Person {
    return this.items[index];
  }
}

let peopleData = new PeopleCollection(people);

console.log(`Osoby: ${peopleData.getNames().join(", ")}`);
let firstPerson = peopleData.getItem(0);
console.log(`Pierwsza osoba: ${firstPerson.name}, ${firstPerson.city}`);
```

Klasa `PeopleCollection` działa na obiektach typu `Person` dostarczanych za pomocą konstruktora lub metody `add()`. Wartością zwrótną metody `getNames()` jest tablica zawierająca wartość `name` każdego obiektu `Person`, natomiast metoda `getItem()` pozwala na pobieranie obiektu `Person` za pomocą indeksu. Następuje utworzenie nowego egzemplarza klasy `PeopleCollection`, a wywołania jego metod prowadzą do wygenerowania następujących danych wyjściowych:

```
Osoby: Bartek Nowak, Dorota Petecka
Pierwsza osoba: Bartek Nowak, Londyn
```

Dodawanie obsługi innego typu

Problem z klasą `PeopleCollection` polega na tym, że działa ona tylko z obiektami typu `Person`. Jeżeli ten sam zbiór operacji chcesz przeprowadzić na obiektach typu `Product`, wówczas oczywisty wybór wiąże się z kompromisami. Mógłbyś utworzyć nową klasę powielającą funkcjonalność klasy już istniejącej. Można to łatwo zrobić, ale w przyszłości może pojawić się kolejny typ wymagający obsługi i powielające się klasy szybko staną się trudne w zarządzaniu. Innym rozwiązaniem jest wykorzystanie zalet funkcjonalności TypeScriptu oraz modyfikacja istniejącej klasy w taki sposób, aby zapewnić obsługę wielu typów, jak pokazałem w kodzie na listingu 12.6.

Listing 12.6. Dodawanie obsługi typów w kodzie pliku `index.ts` w katalogu `src`

```
import { Person, Product } from "./dataTypes";

let people = [new Person("Bartek Nowak", "Londyn"),
  new Person("Dorota Petecka", "Nowy Jork")];
let products = [new Product("Buty do biegania", 100), new Product("Czapka", 25)];

type dataType = Person | Product;

class DataCollection {
  private items: dataType[] = [];

  constructor(initialItems: dataType[]) {
    this.items.push(...initialItems);
  }

  add(newItem: dataType) {
    this.items.push(newItem);
  }

  getNames(): string[] {
    return this.items.map(item => item.name);
  }

  getItem(index: number): dataType {
    return this.items[index];
  }
}

let peopleData = new DataCollection(people);

console.log(`Osoby: ${peopleData.getNames().join(", ")}`);
let firstPerson = peopleData.getItem(0);
if (firstPerson instanceof Person) {
  console.log(`Pierwsza osoba: ${firstPerson.name}, ${firstPerson.city}`);
}
```

Ten listing używa unii typu zapewniającej obsługę również klasy `Product`. Można też zdefiniować interfejs, klasę abstrakcyjną lub funkcję nadpisującą typ, ale mimo to obsługa szerokiej gamy typów będzie wymagała pewnej formy zawężania typu, aby powrócić do konkretnego typu. Kolejny problem polega na tym, że klasa `DataCollection` będzie akceptowała obiekty typu

zarówno Person, jak i Product. Moim celem było zapewnienie obsługi obiektów Person lub Product, a nie obu jednocześnie. Kod przedstawiony na listingu 12.6 powoduje wygenerowanie następujących danych wyjściowych:

Osoby: Bartek Nowak, Dorota Petecka
 Pierwsza osoba: Bartek Nowak, Londyn

Tworzenie klasy generycznej

Klasa generyczna to taka, która ma parametr typu generycznego. Taki parametr działa w charakterze miejsca zarezerwowanego dla typu, który zostanie ustalony w chwili użycia klasy do utworzenia nowego obiektu. Parametr typu generycznego pozwala na utworzenie klasy w sposób umożliwiający jej działanie z określonym typem, który pozostaje nieznany w chwili definiowania klasy, jak pokazałem na listingu 12.7.

Listing 12.7. Używanie typu generycznego w kodzie pliku *index.ts* w katalogu *src*

```
import { Person, Product } from "../dataTypes";

let people = [new Person("Bartek Nowak", "Londyn"),
  new Person("Dorota Petecka", "Nowy Jork")];
let products = [new Product("Buty do biegania", 100), new Product("Czapka", 25)];

//type dataType = Person | Product;
class DataCollection<T> {

  private items: T[] = [];

  constructor(initialItems: T[]) {
    this.items.push(...initialItems);
  }

  add(newItem: T) {
    this.items.push(newItem);
  }

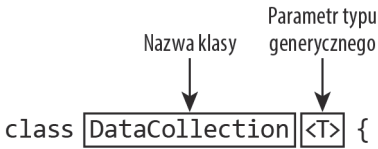
  //getNames(): string[] {
  //  return this.items.map(item => item.name);
  //}

  getItem(index: number): T {
    return this.items[index];
  }
}

let peopleData = new DataCollection<Person>(people);

//console.log(`Osoby: ${peopleData.getNames().join(", ")}`);
let firstPerson = peopleData.getItem(0);
//if (firstPerson instanceof Person) {
console.log(`Pierwsza osoba: ${firstPerson.name}, ${firstPerson.city}`);
//}
```

Klasa `DataCollection` została zdefiniowana z parametrem typu generycznego, który jest częścią deklaracji klasy, jak pokazałem na rysunku 12.1.



```
class DataCollection <T> {
```

Rysunek 12.1. Parametr typu generycznego

Parametr typu generycznego został zdefiniowany w nawiasie ostrym i podana jest tylko jego nazwa. Zgodnie z konwencją nazwa typu takiego parametru rozpoczyna się od litery `T`, choć możesz wybrać dowolny schemat nazw, który ma sens w danym projekcie.

Wynikiem jest *klasa generyczna*, czyli taka, która ma przynajmniej jeden parametr typu generycznego. W omawianym przykładzie parametr typu generycznego ma nazwę `T` i może być używany w miejscu konkretnego typu. Na przykład można zdefiniować konstruktor akceptujący tablicę wartości `T`, jak pokazałem w kolejnym fragmencie kodu:

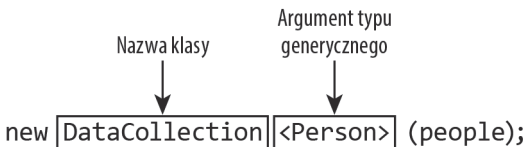
```
...
constructor(initialItems: T[]) {
    this.items.push(...initialItems);
}
...
```

Jak widać, typ generyczny można stosować w adnotacjach typu, choć jeszcze nie są znane szczegóły typu, który będzie w tym miejscu użyty. Klasa przedstawiona na listingu 12.7 definiuje jeden parametr typu o nazwie `T` i jest określana jako `DataCollection<T>`, co wyraźnie wskazuje, że mamy do czynienia z klasą generyczną. Kod przedstawiony na listingu 12.7 powoduje wygenerowanie następujących danych wyjściowych:

Pierwsza osoba: Bartek Nowak, Londyn

Argumenty typu generycznego

Parametr typu generycznego ma przypisywany konkretny typ po użyciu argumentu typu generycznego podczas tworzenia egzemplarza klasy `DataCollection<T>` za pomocą słowa kluczowego `new`, jak pokazałem na rysunku 12.2.



```
new DataCollection <Person> (people);
```

Rysunek 12.2. Tworzenie obiektu z argumentem typu generycznego

Typ argumentu używa nawiasu ostrego — w omawianym przykładzie ten argument wskazuje klasę `Person`.

```
...
let peopleData = new DataCollection<Person>(people);
...
```

To polecenie tworzy obiekt klasy `DataCollection<T>`, a parametrem typu `T` będzie `Person`. Gdy obiekt jest tworzony na podstawie klasy generycznej, jego typ zostaje włączony do argumentu np. `DataCollection<Person>`. Kompilator wymusza stosowanie reguł typu TypeScriptu, używając `Person` po napotkaniu `T`, co oznacza, że tylko obiekty typu `Person` mogą być przekazywane do konstruktora i metody `add()`, a wywołanie metody `getItem()` zwróci obiekt typu `Person`. TypeScript monitoruje argumenty typu używane do utworzenia obiektu `DataCollection<Person>` i nie ma konieczności stosowania asercji typu lub zawężania typu.

Używanie argumentów innego typu

Wartość parametru typu generycznego wpływa tylko na jeden obiekt, a różne typy mogą być używane dla argumentu typu generycznego dla każdego słowa kluczowego `new`, co prowadzi do utworzenia obiektu `DataCollection<T>` działającego z innym typem, jak pokazałem na listingu 12.8.

Listing 12.8. *Używanie argumentu innego typu w kodzie pliku `index.ts` w katalogu `src`*

```
import { Person, Product } from "../dataTypes";

let people = [new Person("Bartek Nowak", "Londyn"),
  new Person("Dorota Petecka", "Nowy Jork")];
let products = [new Product("Buty do biegania", 100), new Product("Czapka", 25)];

class DataCollection<T> {
  private items: T[] = [];

  constructor(initialItems: T[]) {
    this.items.push(...initialItems);
  }

  add(newItem: T) {
    this.items.push(newItem);
  }

  // getNames(): string[] {
  //   return this.items.map(item => item.name);
  // }

  getItem(index: number): T {
    return this.items[index];
  }
}

let peopleData = new DataCollection<Person>(people);
let firstPerson = peopleData.getItem(0);
console.log(`Pierwsza osoba: ${firstPerson.name}, ${firstPerson.city}`);
```

```
let productData = new DataCollection<Product>(products);
let firstProduct = productData.getItem(0);
console.log(`Pierwszy produkt: ${firstProduct.name}, ${firstProduct.price}`);
```

Pierwsze polecenia tworzą obiekt `DataCollection<Product>`, używając `Product` jako typu argumentu generycznego. TypeScript monitoruje typy podawane dla poszczególnych obiektów i gwarantuje, że tylko dany typ będzie mógł być używany. Kod przedstawiony na listingu 12.8 powoduje wygenerowanie następujących danych wyjściowych:

```
Pierwsza osoba: Bartek Nowak, Londyn
Pierwszy produkt: Buty do biegania, 100
```

Ograniczanie wartości typu generycznego

W kodzie przedstawionym na listingach 12.7 i 12.8 metodę `getNames()` umieściłem w komentarzu. Domyślnie dowolny typ może być stosowany z argumentem typu generycznego, więc kompilator traktuje typ generyczny jako `any` i dlatego nie pozwoli na uzyskanie dostępu do właściwości `name` używanej przez metodę `getName()`, o ile nie będzie wykorzystane pewne rozwiązanie w zakresie zawężenia typu.

Wprawdzie można zastosować zawężenie typu dla metody `getNames()`, ale znacznie bardziej eleganckim podejściem jest ograniczenie zakresu typów, które mogą być używane jako wartości parametru typu generycznego. W takim przypadku egzemplarze klasy będą mogły być tworzone jedynie z typami definiującymi funkcjonalność, na której opiera się klasa generyczna, jak pokazałem na listingu 12.9.

Listing 12.9. Ograniczenie typów generycznych w kodzie pliku `index.ts` w katalogu `src`

```
import { Person, Product } from "./dataTypes";

let people = [new Person("Bartek Nowak", "Londyn"),
  new Person("Dorota Petecka", "Nowy Jork")];
let products = [new Product("Buty do biegania", 100), new Product("Czapka", 25)];

class DataCollection<T extends (Person | Product)> {
  private items: T[] = [];

  constructor(initialItems: T[]) {
    this.items.push(...initialItems);
  }

  add(newItem: T) {
    this.items.push(newItem);
  }

  getNames(): string[] {
    return this.items.map(item => item.name);
  }

  getItem(index: number): T {
    return this.items[index];
  }
}
```

```

}

let peopleData = new DataCollection<Person>(people);
let firstPerson = peopleData.getItem(0);
console.log(`Pierwsza osoba: ${firstPerson.name}, ${firstPerson.city}`);
console.log(`Osoby: ${peopleData.getNames().join(", ")}`);

let productData = new DataCollection<Product>(products);
let firstProduct = productData.getItem(0);
console.log(`Pierwszy produkt: ${firstProduct.name}, ${firstProduct.price}`);
console.log(`Produkty: ${productData.getNames().join(", ")}`);

```

Słowo kluczowe `extends` zostało użyte po nazwie typu parametru i pozwala na zdefiniowanie ograniczenia, jak pokazałem na rysunku 12.3.



```

class DataCollection<T extends (Person | Product) > {

```

Rysunek 12.3. Ograniczenie parametru typu generycznego

Zmiana przedstawiona na listingu 12.9 może być potraktowana jako zdefiniowanie dwóch poziomów ograniczenia dla klasy `DataCollection<T>`: jedno stosowane podczas tworzenia nowego obiektu i jedno podczas używania tego obiektu.

Pierwsze ograniczenie zawęży typy, które mogą być używane jako argument typu generycznego do utworzenia nowego obiektu `DataCollection<Product | Person>`, aby tylko typy możliwe do przypisania `Product | Person` mogły być stosowane jako wartość typu parametru. Istnieją trzy typy spełniające ten warunek: `Person`, `Product` i `Person | Product`. Są to jedyne typy możliwe do przypisania parametrowi typu generycznego `T`.

Drugie ograniczenie ma zastosowanie dla wartości parametru typu generycznego podczas używania obiektu. Gdy nowy obiekt jest tworzony z parametrem typu np. `Product`, wówczas `Product` będzie wartością `T`: konstruktor i metody będą akceptować jedynie obiekty typu `Product`, a wartością zwrótną metody `getItem()` będzie jedynie obiekt typu `Product`. Gdy jako typ parametru zostanie użyty `Person`, wówczas `Person` będzie wartością `T` stosowaną przez konstruktor i metody.

Ujmując rzecz inaczej, słowo kluczowe `extends` ogranicza typy, które mogą być przypisywane parametrowi typu, który z kolei ogranicza typy możliwe do użycia przez określone egzemplarze klasy. Skoro kompilator zna wszystkie typy, które mogą być używane przez parametr typu generycznego do zdefiniowania właściwości `name`, pozwala to na usunięcie znaków komentarza na początku wierszy definiujących metodę `getItem()`, a także na odczytywanie wartości właściwości `name` bez powodowania błędów. Kod przedstawiony na listingu 12.9 powoduje wygenerowanie następujących danych wyjściowych:

```

Pierwsza osoba: Bartek Nowak, Londyn
Osoby: Bartek Nowak, Dorota Petecka
Pierwszy produkt: Buty do biegania, 100
Produkty: Buty do biegania, Czapka

```

Ograniczanie typu generycznego za pomocą kształtu

Używanie unii typów do ograniczania parametrów typu generycznego jest użyteczne, ale unia musi być rozszerzana dla każdego nowego typu, którego obsługę trzeba zapewnić. Alternatywne podejście polega na wykorzystaniu kształtu podczas ograniczania typu parametru, co pozwoli na opisywanie jedynie tych właściwości, które są używane przez klasę generyczną, jak pokazałem na listingu 12.10.

Listing 12.10. *Używanie kształtu typu w kodzie pliku index.ts w katalogu src*

```
import { City, Person, Product } from "../dataTypes";

let people = [new Person("Bartek Nowak", "Londyn"),
  new Person("Dorota Petecka", "Nowy Jork")];
let products = [new Product("Buty do biegania", 100), new Product("Czapka", 25)];
let cities = [new City("Londyn", 8136000), new City("Paryż", 2141000)];

class DataCollection<T extends { name: string }> {
  private items: T[] = [];

  constructor(initialItems: T[]) {
    this.items.push(...initialItems);
  }

  add(newItem: T) {
    this.items.push(newItem);
  }

  getNames(): string[] {
    return this.items.map(item => item.name);
  }

  getItem(index: number): T {
    return this.items[index];
  }
}

let peopleData = new DataCollection<Person>(people);
let firstPerson = peopleData.getItem(0);
console.log(`Pierwsza osoba: ${firstPerson.name}, ${firstPerson.city}`);
console.log(`Osoby: ${peopleData.getNames().join(", ")} `);

let productData = new DataCollection<Product>(products);
let firstProduct = productData.getItem(0);
console.log(`Pierwszy produkt: ${firstProduct.name}, ${firstProduct.price}`);
console.log(`Produkty: ${productData.getNames().join(", ")} `);

let cityData = new DataCollection<City>(cities);
console.log(`Miasta: ${cityData.getNames().join(", ")} `);
```

Kształt zdefiniowany na listingu 12.10 wskazuje kompilatorowi, że egzemplarz klasy `DataCollection<T>` może być utworzony z wykorzystaniem dowolnego typu zawierającego właściwość `name` zwracającą wartość typu `string`. Pozwala to na tworzenie obiektów klasy `DataCollection` zapewniających obsługę typów `Person`, `Product` i `City` bez konieczności podawania poszczególnych typów.

- **Wskazówka** Parametry typu generycznego mogą być ograniczane również za pomocą aliasów typu i interfejsów. Istnieje również możliwość ograniczania typów generycznych do tych, które definiują konstruktory określonego kształtu — odbywa się to za pomocą słów kluczowych `extends` `new`, jak pokażę w rozdziale 13.

Kod przedstawiony na listingu 12.10 powoduje wygenerowanie następujących danych wyjściowych:

```
Pierwsza osoba: Bartek Nowak, Londyn
Osoby: Bartek Nowak, Dorota Petecka
Pierwszy produkt: Buty do biegania, 100
Produkty: Buty do biegania, Czapka
Miasta: Londyn, Paryż
```

Definiowanie parametrów wielu typów

Klasa może definiować parametry wielu typów. Przykład przedstawiony na listingu 12.11 dodaje drugi parametr typu do klasy `DataCollection<T>` i używa go do korelacji wartości danych. (Na tym listingu z klasy zostały usunięte metody, które nie będą już potrzebne w przykładach).

Listing 12.11. Definiowanie innego parametru typu generycznego w kodzie pliku `index.ts` w katalogu `src`

```
import { City, Person, Product } from "../dataTypes";

let people = [new Person("Bartek Nowak", "Londyn"),
  new Person("Dorota Petecka", "Nowy Jork")];
let products = [new Product("Buty do biegania", 100), new Product("Czapka", 25)];
let cities = [new City("Londyn", 8136000), new City("Paryż", 2141000)];

class DataCollection<T extends { name: string }, U> {
  private items: T[] = [];

  constructor(initialItems: T[]) {
    this.items.push(...initialItems);
  }

  collate(targetData: U[], itemProp: string, targetProp: string): (T & U)[] {
    let results = [];
    this.items.forEach(item => {
      let match = targetData.find(d => d[targetProp] === item[itemProp]);
      if (match !== undefined) {
        results.push({ ...match, ...item });
      }
    });
    return results;
  }
}

let peopleData = new DataCollection<Person, City>(people);
let collatedData = peopleData.collate(cities, "city", "name");
collatedData.forEach(c => console.log(`${c.name}, ${c.city}, ${c.population}`));
```


Dodatkowe parametry typu są rozdzielone przecinkami, podobnie jak w przypadku parametrów zwykłych funkcji lub metod. Klasa `DataCollection<T, U>` definiuje dwa parametry typu generycznego. Nowy parametr o nazwie `U` został użyty do zdefiniowania typu argumentu przekazywanego metodzie `collate()` porównującej właściwości tablicy obiektów i złączenia obiektów `T` i `U`, które mają te same wartości właściwości.

Podczas tworzenia egzemplarza klasy generycznej konieczne jest podanie rozdzielonych przecinkami argumentów dla każdego parametru typu generycznego, jak pokazałem w kolejnym fragmencie kodu:

```
...
let peopleData = new DataCollection<Person, City>(people);
...
```

To polecenie tworzy obiekt `DataCollection<Person, City>` przechowujący obiekty `Person` i porównujący je z obiektami `City`. Tablica obiektów `City` jest przekazywana metodzie `collate()`, porównywane są wartości właściwości `city` obiektu `Person` i wartości właściwości `name` obiektu `City`.

Właściwości obiektów o dopasowanych wartościach są łączone za pomocą operatora rozwinięcia w celu utworzenia złączenia.

```
...
results.push({ ...match, ...item });
...
```

Istnieje jedna para obiektów o dopasowanych wartościach, a kod przedstawiony na listingu 12.11 powoduje wygenerowanie następujących danych wyjściowych:

```
Bartek Nowak, Londyn, 8136000
```

Stosowanie parametru typu w metodzie

Drugi parametr typu w kodzie przedstawionym na listingu 12.11 nie jest tak elastyczny, jak mógłby być, ponieważ wymaga, aby typ danych używany przez metodę `collate()` został podany podczas tworzenia obiektu `DataCollection` — to będzie jedyny typ danych, który może być używany przez tę metodę.

Gdy typ jest używany tylko przez jedną metodę, parametr typu można przenieść z deklaracji klasy i zastosować bezpośrednio w metodzie, co pozwala na podawanie różnych typów w trakcie każdego wywołania metody, jak pokazałem na listingu 12.12.

Listing 12.12. *Zastosowanie parametru typu w metodzie w kodzie pliku `index.ts` w katalogu `src`*

```
import { City, Person, Product, Employee } from "../dataTypes";

let people = [new Person("Bartek Nowak", "Londyn"),
  new Person("Dorota Petecka", "Nowy Jork")];
let products = [new Product("Buty do biegania", 100), new Product("Czapka", 25)];
let cities = [new City("Londyn", 8136000), new City("Paryż", 2141000)];
let employees = [new Employee("Bartek Nowak", "handlowiec"),
  new Employee("Alicja Janowska", "handlowiec)];

class DataCollection<T extends { name: string }> {
  private items: T[] = [];
```

```

    constructor(initialItems: T[]) {
        this.items.push(...initialItems);
    }

    collate<U>(targetData: U[], itemProp: string, targetProp: string): (T & U)[] {
        let results = [];
        this.items.forEach(item => {
            let match = targetData.find(d => d[targetProp] === item[itemProp]);
            if (match !== undefined) {
                results.push({ ...match, ...item });
            }
        });
        return results;
    }
}

let peopleData = new DataCollection<Person>(people);
let collatedData = peopleData.collate<City>(cities, "city", "name");
collatedData.forEach(c => console.log(`${c.name}, ${c.city}, ${c.population}`));
let empData = peopleData.collate<Employee>(employees, "name", "name");
empData.forEach(c => console.log(`${c.name}, ${c.city}, ${c.role}`));

```

Parametr typu `U` jest stosowany bezpośrednio w metodzie `collate()`, co pozwala na podanie typu podczas wywoływania metody, jak pokazałem w kolejnym fragmencie kodu:

```

...
let collatedData = peopleData.collate<City>(cities, "city", "name");
...

```

Parametr typu metody umożliwia metodzie `collate()` wywołanie z obiektami `City`, a następnie ponownie z obiektami `Employee`. Kod przedstawiony na listingu 12.12 powoduje wygenerowanie następujących danych wyjściowych:

```

Bartek Nowak, Londyn, 8136000
Bartek Nowak, Londyn, handlowiec

```

Pozostawienie kompilatorowi zadania ustalenia typu argumentu

Kompilator TypeScriptu ma możliwość ustalenia typu argumentu na podstawie sposobu tworzenia obiektu lub wywoływania metody. Może to być użyteczny sposób na tworzenie zwięzłego kodu, choć jednocześnie wymaga ostrożności, ponieważ trzeba zagwarantować inicjalizację obiektów z wyraźnie wymienionymi typami. Kod przedstawiony na listingu 12.13 tworzy egzemplarz klasy `DataCollection<T>` i wywołuje metodę `collate()` bez typu argumentów, pozostawiając zadanie ich ustalenia kompilatorowi.

Listing 12.13. Ustalanie typu generycznego w kodzie pliku `index.ts` w katalogu `src`

```

import { City, Person, Product, Employee } from "../dataTypes";

let people = [new Person("Bartek Nowak", "Londyn"),
    new Person("Dorota Petecka", "Nowy Jork")];
let products = [new Product("Buty do biegania", 100), new Product("Czapka", 25)];

```

```

let cities = [new City("Londyn", 8136000), new City("Paryż", 2141000)];
let employees = [new Employee("Bartek Nowak", "handlowiec"),
  new Employee("Alicja Janowska", "handlowiec)];

class DataCollection<T extends { name: string }> {
  private items: T[] = [];

  constructor(initialItems: T[]) {
    this.items.push(...initialItems);
  }

  collate<U>(targetData: U[], itemProp: string, targetProp: string): (T & U)[] {
    let results = [];
    this.items.forEach(item => {
      let match = targetData.find(d => d[targetProp] === item[itemProp]);
      if (match !== undefined) {
        results.push({ ...match, ...item });
      }
    });
    return results;
  }
}

```

```

export let peopleData = new DataCollection(people);
export let collatedData = peopleData.collate(cities, "city", "name");
collatedData.forEach(c => console.log(`${c.name}, ${c.city}, ${c.population}`));
export let empData = peopleData.collate(employees, "name", "name");
empData.forEach(c => console.log(`${c.name}, ${c.city}, ${c.role}`));

```

Kompilator ma możliwość ustalenia typu argumentów na podstawie argumentu przekazanego konstruktorowi `DataCollection<T>` i pierwszego argumentu przekazanego metodzie `collate()`. Jeżeli chcesz sprawdzić typ określony przez kompilator, przeanalizuj zawartość pliku *index.d.ts* w katalogu *dist*, który zostanie utworzony w przypadku wybrania opcji *declaration*.

■ **Wskazówka** W projekcie używającym modułów pliki tworzone za pomocą opcji *declaration* zawierają jedynie te typy, które są eksportowane w module. Dlatego też w kodzie przedstawionym na listingu 12.13 znalazło się słowo kluczowe `export`.

Spójrz na typy określone przez kompilator w omawianym przykładzie:

```

...
export declare let peopleData: DataCollection<Person>;
export declare let collatedData: (Person & City)[];
export declare let empData: (Person & Employee)[];
...

```

Przykład zaprezentowany na listingu 12.13 powoduje wygenerowanie następujących danych wyjściowych:

```

Bartek Nowak, Londyn, 8136000
Bartek Nowak, Londyn, handlowiec

```

Rozszerzanie klasy generycznej

Klasa generyczna może być rozszerzona, a podklasy mogą działać z parametrami typu generycznego na wiele sposobów, jak to przedstawiłem w kolejnych punktach.

Dodawanie funkcjonalności do istniejących parametrów typu

Pierwsze podejście polega na dodawaniu nowej funkcjonalności do już zdefiniowanej przez klasę nadrzędną, używając tych samych typów generycznych, jak pokazałem na listingu 12.14.

Listing 12.14. Tworzenie podklasy klasy generycznej w kodzie pliku index.ts w katalogu src

```
import { City, Person, Product, Employee } from "./dataTypes";

let people = [new Person("Bartek Nowak", "Londyn"),
  new Person("Dorota Petecka", "Nowy Jork")];
let products = [new Product("Buty do biegania", 100), new Product("Czapka", 25)];
let cities = [new City("Londyn", 8136000), new City("Paryż", 2141000)];
let employees = [new Employee("Bartek Nowak", "handlowiec"),
  new Employee("Alicja Janowska", "handlowiec")];

class DataCollection<T extends { name: string }> {
  protected items: T[] = [];

  constructor(initialItems: T[]) {
    this.items.push(...initialItems);
  }

  collate<U>(targetData: U[], itemProp: string, targetProp: string): (T & U)[] {
    let results = [];
    this.items.forEach(item => {
      let match = targetData.find(d => d[targetProp] === item[itemProp]);
      if (match !== undefined) {
        results.push({ ...match, ...item });
      }
    });
    return results;
  }
}

class SearchableCollection<T extends { name: string }> extends DataCollection<T> {
  constructor(initialItems: T[]) {
    super(initialItems);
  }

  find(name: string): T | undefined {
    return this.items.find(item => item.name === name);
  }
}

let peopleData = new SearchableCollection<Person>(people);
let foundPerson = peopleData.find("Bartek Nowak");
```

```
if (foundPerson !== undefined) {
    console.log(`Osoba ${ foundPerson.name }, ${ foundPerson.city}`);
}
```

Klasa `SearchableCollection<T>` jest klasą pochodną `DataCollection<T>` i definiuje metodę `find()` wyszukującą obiekt na podstawie jego właściwości `name`. Deklaracja klasy `SearchableCollection<T>` używa słowa kluczowego `extends` i zawiera typy parametrów, jak pokazałem w kolejnym fragmencie kodu:

```
...
class SearchableCollection<T extends { name: string }> extends DataCollection<T> {
...

```

Typ klasy generycznej zawiera typy jej parametrów, więc klasą nadrzędną jest `DataCollection<T>`. Typ parametru definiowanego przez klasę `SearchableCollection<T>` musi być zgodny z typem parametru klasy nadrzędnej. Dlatego też użyłem tego samego kształtu typu do określenia typów definiujących właściwość `name`.

■ **Wskazówka** Zwróć uwagę na zmianę słowa kluczowego kontroli dostępu (na `protected`) we właściwości `items` w kodzie na listingu 12.14. Pozwala to na uzyskanie dostępu przez klasę pochodną. W rozdziale 11. przedstawiłem więcej informacji na temat dostarczanych przez TypeScript słów kluczowych kontroli dostępu.

Egzemplarz klasy `SearchableCollection<T>` jest tworzony podobnie jak każdy inny, za pomocą argumentu typu (lub po ustaleniu typu przez kompilator). Kod przedstawiony na listingu 12.14 powoduje wygenerowanie następujących danych wyjściowych:

```
Osoba Bartek Nowak, Londyn
```

Ustawienie na stałe parametru typu generycznego

Część klas musi definiować funkcjonalność dostępną jedynie z użyciem podzbioru typów obsługiwanych przez klasę nadrzędną. W większości sytuacji podklasa może użyć na stałe zdefiniowanego typu dla typu parametru klasy nadrzędnej w taki sposób, że podklasa nie jest klasą generyczną, jak pokazałem na listingu 12.15.

Listing 12.15. Ustawienie na stałe parametru typu generycznego w kodzie pliku `index.ts` w katalogu `src`

```
import { City, Person, Product, Employee } from "../dataTypes";

let people = [new Person("Bartek Nowak", "Londyn"),
    new Person("Dorota Petecka", "Nowy Jork")];
let products = [new Product("Buty do biegania", 100), new Product("Czapka", 25)];
let cities = [new City("Londyn", 8136000), new City("Paryż", 2141000)];
let employees = [new Employee("Bartek Nowak", "handlowiec"),
    new Employee("Alicja Janowska", "handlowiec")];

class DataCollection<T extends { name: string }> {
    protected items: T[] = [];
```

```

    constructor(initialItems: T[]) {
        this.items.push(...initialItems);
    }

    collate<U>(targetData: U[], itemProp: string, targetProp: string): (T & U)[] {
        let results = [];
        this.items.forEach(item => {
            let match = targetData.find(d => d[targetProp] === item[itemProp]);
            if (match !== undefined) {
                results.push({ ...match, ...item });
            }
        });
        return results;
    }
}

class SearchableCollection extends DataCollection<Employee> {

    constructor(initialItems: Employee[]) {
        super(initialItems);
    }

    find(searchTerm: string): Employee[] {
        return this.items.filter(item =>
            item.name === searchTerm || item.role === searchTerm);
    }
}

let employeeData = new SearchableCollection(employees);
employeeData.find("handlowiec").forEach(e =>
    console.log(`Pracownik ${ e.name }, ${ e.role}`));

```

Klasa `SearchableCollection` rozszerza klasę `DataCollection<Employee>`, która na stałe definiuje parametr typu generycznego, aby klasa `SearchableCollection` mogła współdziałać jedynie z obiektami klasy `Employee`. Żaden typ parametru nie może zostać użyty podczas tworzenia obiektu klasy `SearchableCollection`, a kod metody `find()` może bezpiecznie uzyskiwać dostęp do właściwości definiowanych przez klasę `Employee`. Kod przedstawiony na listingu 12.15 powoduje wygenerowanie następujących danych wyjściowych:

```

Pracownik Bartek Nowak, handlowiec
Pracownik Alicja Janowska, handlowiec

```

Ograniczanie parametru typu generycznego

Trzecie rozwiązanie pozwala na zachowanie równowagi między dwoma wcześniejszymi przykładami: wykorzystywana jest zmienna typu generycznego i jednocześnie następuje ograniczenie do określonych typów, jak pokazałem na listingu 12.16. Pozwala to na otrzymanie funkcjonalności zależnej od funkcji poszczególnych klas bez ustalania na stałe parametru typu.

Listing 12.16. Ograniczanie parametru typu w pliku *index.ts* w katalogu *src*

```

import { City, Person, Product, Employee } from "./dataTypes";

let people = [new Person("Bartek Nowak", "Londyn"),
  new Person("Dorota Petecka", "Nowy Jork")];
let products = [new Product("Buty do biegania", 100), new Product("Czapka", 25)];
let cities = [new City("Londyn", 8136000), new City("Paryż", 2141000)];
let employees = [new Employee("Bartek Nowak", "handlowiec"),
  new Employee("Alicja Janowska", "handlowiec")];

class DataCollection<T extends { name: string }> {
  protected items: T[] = [];

  constructor(initialItems: T[]) {
    this.items.push(...initialItems);
  }

  collate<U>(targetData: U[], itemProp: string, targetProp: string): (T & U)[] {
    let results = [];
    this.items.forEach(item => {
      let match = targetData.find(d => d[targetProp] === item[itemProp]);
      if (match !== undefined) {
        results.push({ ...match, ...item });
      }
    });
    return results;
  }
}

class SearchableCollection<T extends Employee | Person> extends DataCollection<T> {
  constructor(initialItems: T[]) {
    super(initialItems);
  }

  find(searchTerm: string): T[] {
    return this.items.filter(item => {
      if (item instanceof Employee) {
        return item.name === searchTerm || item.role === searchTerm;
      } else if (item instanceof Person) {
        return item.name === searchTerm || item.city === searchTerm;
      }
    });
  }
}

let employeeData = new SearchableCollection<Employee>(employees);
employeeData.find("handlowiec").forEach(e =>
  console.log(`Pracownik ${ e.name }, ${ e.role }`));

```

Parametr typu określony przez podklasę musi być możliwy do przypisania dziedzicznemu parametrowi typu, co oznacza, że dozwolone jest używanie tylko bardziej restrykcyjnego typu.

W omawianym przykładzie unia `Employee | Person` może być przypisana kształtowi używanemu do ograniczenia parametru typu `DataCollection<T>`.

■ **Ostrzeżenie** Podczas używania unii do ograniczania parametru typu generycznego trzeba pamiętać, że unia sama w sobie jest akceptowalnym argumentem dla tego parametru. Oznacza to, że egzemplarz przedstawionej na listingu 12.16 klasy `SearchableCollection` może być utworzony z parametrem typu `Employee`, `Product` i `Employee | Product`. Więcej informacji na temat ograniczania typu argumentów znajdziesz w rozdziale 13.

Metoda `find()` używa słowa kluczowego `instanceof` do zawężenia obiektów do określonych typów w celu przeprowadzania porównań wartości. Kod przedstawiony na listingu 12.16 powoduje wygenerowanie następujących danych wyjściowych:

```
Pracownik Bartek Nowak, handlowiec
Pracownik Alicja Janowska, handlowiec
```

Wartownik typu generycznego

Klasa `SearchableCollection` przedstawiona w kodzie na listingu 12.16 używała słowa kluczowego `instanceof` do identyfikacji obiektów `Employee` i `Person`. Jest to rozwiązanie możliwe do zarządzania, ponieważ ograniczenie zastosowane do parametru typu oznacza konieczność pracy z niewielką liczbą typów. W przypadku klas o nieograniczonych parametrach typu zawężenie do określonego typu może być trudne, jak pokazałem na listingu 12.17.

Listing 12.17. Zawężanie typu generycznego w kodzie pliku `index.ts` w katalogu `src`

```
import { City, Person, Product, Employee } from "../dataTypes";

let people = [new Person("Bartek Nowak", "Londyn"),
  new Person("Dorota Petecka", "Nowy Jork")];
let products = [new Product("Buty do biegania", 100), new Product("Czapka", 25)];
let cities = [new City("Londyn", 8136000), new City("Paryż", 2141000)];
let employees = [new Employee("Bartek Nowak", "handlowiec"),
  new Employee("Alicja Janowska", "handlowiec")];

class DataCollection<T> {
  protected items: T[] = [];

  constructor(initialItems: T[]) {
    this.items.push(...initialItems);
  }

  filter<V extends T>(): V[] {
    return this.items.filter(item => item instanceof V) as V[];
  }
}

let mixedData = new DataCollection<Person | Product>([...people, ...products]);
let filteredProducts = mixedData.filter<Product>();
filteredProducts.forEach(p => console.log(`Produkt: ${p.name}, ${p.price}`));
```


Kod przedstawiony na listingu 12.17 wprowadza metodę `filter()` używając słowa kluczowego `instanceof` do pobrania obiektów określonego typu z tablicy elementów danych. Obiekt `DataCollection<Person | Product>` jest tworzony z tablicą zawierającą połączenie obiektów `Person` i `Product`, a nowa metoda `filter()` pozwala na pobieranie obiektów `Product`.

■ **Wskazówka** Zwróć uwagę na parametr typu generycznego metody `filter()` o nazwie `V` zdefiniowany za pomocą słowa kluczowego `extend`, co wskazuje kompilatorowi, że może akceptować jedynie typy możliwe do przypisania typowi generycznemu `T` klasy, co z kolei uniemożliwia kompilatorowi traktowanie `V` jako `any`.

Próba kompilacji tego przykładu powoduje wygenerowanie następującego błędu:

```
src/index.ts(18,58): error TS2693: 'V' only refers to a type, but is being used as
a value here.
```

W języku JavaScript nie istnieje funkcjonalność będąca odpowiednikiem typu generycznego, więc w trakcie procesu kompilacji ta funkcjonalność została usunięta z kodu TypeScriptu. To z kolei oznacza brak w trakcie działania aplikacji dostępnych informacji pozwalających na użycie typu generycznego ze słowem kluczowym `instanceof`.

W sytuacjach wymagających identyfikacji obiektów według typu typ generyczny nie będzie zbyt pomocny i zamiast niego należy skorzystać z funkcji predykatu. W kodzie na listingu 12.18 dodałem do metody `filter()` parametr akceptujący typ funkcji predykatu, która jest następnie używana do wyszukiwania obiektów określonego typu.

Listing 12.18. *Używanie typu funkcji predykatu w kodzie pliku `index.ts` w katalogu `src`*

```
import { City, Person, Product, Employee } from "../dataTypes";

let people = [new Person("Bartek Nowak", "Londyn"),
  new Person("Dorota Petecka", "Nowy Jork")];
let products = [new Product("Buty do biegania", 100), new Product("Czapka", 25)];
let cities = [new City("Londyn", 8136000), new City("Paryż", 2141000)];
let employees = [new Employee("Bartek Nowak", "handlowiec"),
  new Employee("Alicja Janowska", "handlowiec")];

class DataCollection<T> {
  protected items: T[] = [];

  constructor(initialItems: T[]) {
    this.items.push(...initialItems);
  }

  filter<V extends T>(predicate: (target) => target is V): V[] {
    return this.items.filter(item => predicate(item)) as V[];
  }
}

let mixedData = new DataCollection<Person | Product>([...people, ...products]);
function isProduct(target): target is Product {
```

```
    return target instanceof Product;
}
let filteredProducts = mixedData.filter<Product>(isProduct);
filteredProducts.forEach(p => console.log(`Produkt: ${ p.name}, ${p.price}`));
```

Funkcja predykatu dla wymaganego typu jest dostarczana w postaci argumentu metody `filter()` przy użyciu funkcjonalności JavaScriptu dostępnej podczas wykonywania kodu. W ten sposób dysponujesz metodą pozwalającą na pobieranie niezbędnych obiektów. Kod przedstawiony na listingu 12.18 powoduje wygenerowanie następujących danych wyjściowych:

```
Produkt: Buty do biegania, 100
Produkt: Czapka, 25
```

Definiowanie metody statycznej w klasie generycznej

Tylko metody i właściwości egzemplarza mają typ generyczny, który może być inny dla poszczególnych obiektów. Metoda statyczna jest dostępna za pomocą klasy, jak pokazałem w przykładzie na listingu 12.19.

Listing 12.19. Definiowanie metody statycznej w kodzie pliku `index.ts` w katalogu `src`

```
import { City, Person, Product, Employee } from "../dataTypes";

let people = [new Person("Bartek Nowak", "Londyn"),
  new Person("Dorota Petecka", "Nowy Jork")];
let products = [new Product("Buty do biegania", 100), new Product("Czapka", 25)];
let cities = [new City("Londyn", 8136000), new City("Paryż", 2141000)];
let employees = [new Employee("Bartek Nowak", "handlowiec"),
  new Employee("Alicja Janowska", "handlowiec")];

class DataCollection<T> {
  protected items: T[] = [];

  constructor(initialItems: T[]) {
    this.items.push(...initialItems);
  }

  filter<V extends T>(predicate: (target) => target is V): V[] {
    return this.items.filter(item => predicate(item)) as V[];
  }

  static reverse(items: any[]) {
    return items.reverse();
  }
}

let mixedData = new DataCollection<Person | Product>([...people, ...products]);

function isProduct(target): target is Product {
  return target instanceof Product;
}
```

```
let filteredProducts = mixedData.filter<Product>(isProduct);
filteredProducts.forEach(p => console.log(`Produkt: ${p.name}, ${p.price}`));
```

```
let reversedCities: City[] = DataCollection.reverse(cities);
reversedCities.forEach(c => console.log(`Miasto: ${c.name}, ${c.population}`));
```

Metoda statyczna `reserve()` jest dostępna poprzez klasę `DataCollection` bez konieczności używania typu argumentu, jak widać w kolejnym fragmencie kodu:

```
...
let reversedCities: City[] = DataCollection.reverse(cities);
...
```

Metoda statyczna może definiować własne parametry typu generycznego, jak pokazałem na listingu 12.20.

Listing 12.20. Dodawanie parametru typu w kodzie pliku `index.ts` w katalogu `src`

```
import { City, Person, Product, Employee } from "./dataTypes";

let people = [new Person("Bartek Nowak", "Londyn"),
  new Person("Dorota Petecka", "Nowy Jork")];
let products = [new Product("Buty do biegania", 100), new Product("Czapka", 25)];
let cities = [new City("Londyn", 8136000), new City("Paryż", 2141000)];
let employees = [new Employee("Bartek Nowak", "handlowiec"),
  new Employee("Alicja Janowska", "handlowiec")];

class DataCollection<T> {
  protected items: T[] = [];

  constructor(initialItems: T[]) {
    this.items.push(...initialItems);
  }

  filter<V extends T>(predicate: (target) => target is V): V[] {
    return this.items.filter(item => predicate(item)) as V[];
  }

  static reverse<ArrayType>(items: ArrayType[]): ArrayType[] {
    return items.reverse();
  }
}

let mixedData = new DataCollection<Person | Product>([...people, ...products]);

function isProduct(target): target is Product {
  return target instanceof Product;
}

let filteredProducts = mixedData.filter<Product>(isProduct);
filteredProducts.forEach(p => console.log(`Produkt: ${p.name}, ${p.price}`));

let reversedCities = DataCollection.reverse<City>(cities);
reversedCities.forEach(c => console.log(`Miasto: ${c.name}, ${c.population}`));
```

Metoda `reverse()` definiuje parametr typu określający typ przetwarzanej tablicy. Wywołanie metody odbywa się za pomocą klasy `DataCollection`, a argument typu jest podawany po nazwie metody:

```
...
let reversedCities = DataCollection.reverse<City>(cities);
...
```

Parametry typu definiowane przez metody statyczne są odrębne od tych zdefiniowanych przez klasę i przeznaczonych do używania przez jej metody i właściwości egzemplarza. Przykład zaprezentowany na listingu 12.20 powoduje wygenerowanie następujących danych wyjściowych:

```
Produkt: Buty do biegania, 100
Produkt: Czapka, 25
Miasto: Paryż, 2141000
Miasto: Londyn, 8136000
```

Definiowanie interfejsu generycznego

Interfejs może być zdefiniowany z parametrami typu generycznego, co pozwala na zdefiniowanie funkcjonalności bez konieczności określania poszczególnych typów. Kod przedstawiony na listingu 12.21 definiuje interfejs z parametrem typu generycznego.

Listing 12.21. Definiowanie interfejsu generycznego w kodzie pliku `index.ts` w katalogu `src`

```
import { City, Person, Product, Employee } from "../dataTypes";

type shapeType = { name: string };

interface Collection<T extends shapeType> {
  add(...newItems: T[]): void;
  get(name: string): T;
  count: number;
}
```

Interfejs `Collection<T>` ma parametr typu generycznego o nazwie `T` pozwalający na używanie tej samej składni co w przypadku parametrów typu klasy. Parametry typu są używane przez metody `add()` i `get()` oraz zostały ograniczone w celu zapewnienia, że będą mogły być stosowane jedynie typy zawierające właściwość `name`.

Interfejs z parametrem typu generycznego opisuje zbiór operacji abstrakcyjnych, ale nie określa typów, na których mogą być one przeprowadzane. Dzięki temu typy mogą być wskazywane przez interfejsy pochodne lub implementacje klas. Kod przedstawiony na listingu 12.21 nie powoduje wygenerowania żadnych danych wyjściowych.

Rozszerzanie interfejsu generycznego

Interfejs generyczny może być rozszerzony w dokładnie taki sam sposób jak zwykły interfejs, a opcje przeznaczone do pracy z jego parametrami typu są takie same jak stosowane podczas rozszerzania

klasy generycznej. Na listingu 12.22 możesz zobaczyć zbiór interfejsów rozszerzających interfejs `Collection<T>`.

Listing 12.22. Rozszerzenie interfejsu generycznego w kodzie pliku `index.ts` w katalogu `src`

```
import { City, Person, Product, Employee } from "../dataTypes";

type shapeType = { name: string };

interface Collection<T extends shapeType> {
    add(...newItems: T[]): void;
    get(name: string): T;
    count: number;
}

interface SearchableCollection<T extends shapeType> extends Collection<T> {
    find(name: string): T | undefined;
}

interface ProductCollection extends Collection<Product> {
    sumPrices(): number;
}

interface PeopleCollection<T extends Product | Employee> extends Collection<T> {
    getNames(): string[];
}
```

Kod przedstawiony na listingu 12.22 nie powoduje wygenerowania żadnych danych wyjściowych.

Implementacja interfejsu generycznego

Gdy klasa implementuje interfejs generyczny, musi implementować wszystkie metody i właściwości interfejsu, choć ma pewien wybór w zakresie obsługi parametrów typu, co dokładnie przedstawię w kolejnych punktach. Część tych opcji jest podobna do stosowanych podczas rozszerzania klas generycznych i interfejsów.

Przekazywanie parametru typu generycznego

Najprostsze podejście polega na implementacji metod i właściwości interfejsu bez zmiany parametru typu, tworząc klasę generyczną, która bezpośrednio implementuje interfejs, jak pokazałem na listingu 12.23.

Listing 12.23. Implementowanie interfejsu w kodzie pliku `index.ts` w katalogu `src`

```
import { City, Person, Product, Employee } from "../dataTypes";

type shapeType = { name: string };

interface Collection<T extends shapeType> {
```

```

    add(...newItems: T[]): void;
    get(name: string): T;
    count: number;
}

class ArrayCollection<DataType extends shapeType> implements Collection<DataType> {
    private items: DataType[] = [];

    add(...newItems): void {
        this.items.push(...newItems);
    }

    get(name: string): DataType {
        return this.items.find(item => item.name === name);
    }

    get count(): number {
        return this.items.length;
    }
}

let peopleCollection: Collection<Person> = new ArrayCollection<Person>();
peopleCollection.add(new Person("Bartek Nowak", "Londyn"),
    new Person("Dorota Petecka", "Nowy Jork"));
console.log(`Wielkość kolekcji: ${peopleCollection.count}`);

```

Klasa `ArrayCollection<DataType>` używa słowa kluczowego `implements` do zadeklarowania zgodności z interfejsem. Ten interfejs ma parametr typu generycznego, więc klasa `ArrayCollection<DataType>` musi definiować zgodny z nim parametr. Skoro parametr dla interfejsu musi mieć właściwość `name`, to samo dotyczy parametru typu klasy, zatem dla interfejsu i klasy użyłem tego samego aliasu typu, aby w ten sposób zapewnić spójność.

Klasa `ArrayCollection<DataType>` wymaga argumentu typu podczas tworzenia obiektu i może działać poprzez interfejs `Collection<T>`, jak widać w kolejnym fragmencie kodu:

```

...
let peopleCollection: Collection<Person> = new ArrayCollection<Person>();
...

```

Argument typu określa odpowiedni typ dla klasy i implementowanego interfejsu, więc obiekt `ArrayCollection<DataType>` implementuje interfejs `Collection<Person>`. Kod przedstawiony na listingu 12.23 powoduje wygenerowanie następujących danych wyjściowych:

```

Wielkość kolekcji: 2

```

Ograniczenie lub określenie na stałe parametru typu generycznego

Klasa może dostarczyć implementację interfejsu przeznaczoną dla określonego typu lub podzbioru typów obsługiwanych przez interfejs. Takie rozwiązanie widać w kodzie przedstawionym na listingu 12.24.

Listing 12.24. Implementowanie interfejsu w kodzie pliku *index.ts* w katalogu *src*

```
import { City, Person, Product, Employee } from "../dataTypes";

type shapeType = { name: string };

interface Collection<T extends shapeType> {
    add(...newItems: T[]): void;
    get(name: string): T;
    count: number;
}

class PersonCollection implements Collection<Person> {
    private items: Person[] = [];

    add(...newItems: Person[]): void {
        this.items.push(...newItems);
    }

    get(name: string): Person {
        return this.items.find(item => item.name === name);
    }

    get count(): number {
        return this.items.length;
    }
}

let peopleCollection: Collection<Person> = new PersonCollection();
peopleCollection.add(new Person("Bartek Nowak", "Londyn"),
    new Person("Dorota Petecka", "Nowy Jork"));
console.log(`Wielkość kolekcji: ${peopleCollection.count}`);
```

Klasa `PersonCollection` implementuje interfejs `Collection<Product>`. Po skompilowaniu i uruchomieniu kodu przedstawionego na listingu 12.24 zostaną wygenerowane następujące dane wyjściowe:

```
Wielkość kolekcji: 2
```

Tworzenie implementacji interfejsu abstrakcyjnego

Klasa abstrakcyjna może dostarczyć częściową implementację interfejsu, która następnie będzie dokończona przez podklasy. Klasa abstrakcyjna ma ten sam zbiór opcji przeznaczonych do pracy z parametrami typu, jaki jest dostępny dla zwykłej klasy: przekazanie w niezmienionej postaci do podklasy, nałożenie kolejnych ograniczeń lub też na stałe określenie konkretnego typu. W kodzie przedstawionym na listingu 12.25 pokazałem klasę abstrakcyjną otrzymującą argument w postaci interfejsu typu generycznego.

Listing 12.25. Definiowanie klasy abstrakcyjnej w kodzie pliku *index.ts* w katalogu *src*

```
import { City, Person, Product, Employee } from "../dataTypes";

type shapeType = { name: string };
```

```
interface Collection<T extends shapeType> {
    add(...newItems: T[]): void;
    get(name: string): T;
    count: number;
}

abstract class ArrayCollection<T extends shapeType> implements Collection<T> {
    protected items: T[] = [];

    add(...newItems: T[]): void {
        this.items.push(...newItems);
    }

    abstract get(searchTerm: string): T;

    get count(): number {
        return this.items.length;
    }
}

class ProductCollection extends ArrayCollection<Product> {
    get(searchTerm: string): Product {
        return this.items.find(item => item.name === searchTerm);
    }
}

class PersonCollection extends ArrayCollection<Person> {
    get(searchTerm: string): Person {
        return this.items.find(item =>
            item.name === searchTerm || item.city === searchTerm);
    }
}

let peopleCollection: Collection<Person> = new PersonCollection();
peopleCollection.add(new Person("Bartek Nowak", "Londyn"),
    new Person("Dorota Petecka", "Nowy Jork"));
let productCollection: Collection<Product> = new ProductCollection();
productCollection.add(new Product("Buty do biegania", 100), new Product("Czapka", 25));
[peopleCollection, productCollection].forEach(c => console.log(`Wielkość: ${c.count}`));
```

Klasa `ArrayCollection<T>` jest abstrakcyjna i dostarcza częściową implementację interfejsu `Collection<T>`, pozostawiając podklasom dostarczenie metody `get()`. Klasy `ProductCollection` i `PersonCollection` rozszerzają `ArrayCollection<T>`, zawężając parametr typu generycznego do określonych typów i implementując metodę `get()` do użycia właściwości typu, na którym operują. Kod przedstawiony na listingu 12.25 powoduje wygenerowanie następujących danych wyjściowych:

```
Wielkość: 2
Wielkość: 2
```

Podsumowanie

W tym rozdziale przedstawiłem wprowadzenie do typu generycznego i zaprezentowałem rozwiązywany przez niego problem. Poznałeś relacje zachodzące między argumentami i parametrami typu generycznego, a także różne sposoby, w jakie można ograniczać lub na stałe definiować typy generyczne. Wyjaśniłem, że typy generyczne mogą być używane z klasami zwykłymi, abstrakcyjnymi i interfejsami. Zobaczyłeś również, że funkcje i metody mogą mieć typy generyczne, które są rozwiązywane w momencie ich użycia. W następnym rozdziale przedstawię bardziej zaawansowane funkcje typów generycznych oferowane przez TypeScript.

ROZDZIAŁ 13.



Zaawansowane typy generyczne

W rozdziale będę kontynuował omawianie funkcji typów generycznych dostarczanej przez TypeScript i skoncentruję się na jej bardziej zaawansowanych możliwościach. Wyjaśnię, jak używać typów generycznych z kolekcjami i iteratorami, a także wprowadzę funkcjonalność typów indeksu i mapowania typu. Zaprezentuję również najelastyczniejsze funkcje typów generycznych, czyli typy warunkowe. Streszczenie materiału przedstawionego w rozdziale znajdziesz w tabeli 13.1.

Tabela 13.1. Streszczenie materiału przedstawionego w rozdziale

Problem	Rozwiązanie	Listing
Używanie kolekcji klas z zachowaniem bezpieczeństwa typów	Podczas tworzenia kolekcji podaj argument typu generycznego	3 i 4
Używanie iteratorów z zachowaniem bezpieczeństwa typów	Użyj interfejsów TypeScriptu obsługiwanych przez argumenty typu generycznego	Od 5 do 7
Definiowanie typu, którego wartość może przyjmować jedynie nazwę właściwości	Użyj zapytania typu indeksu	Od 8 do 14
Przekształcanie typu	Użyj mapowania typu	Od 15 do 22
Programowe wybieranie typu	Użyj typów warunkowych	Od 23 do 32

W tabeli 13.2 wymieniłem opcje kompilatora TypeScriptu użyte w rozdziale.

Przygotowanie projektu

W rozdziale będę kontynuował używanie projektu *types* utworzonego w rozdziale 7. i uaktualnionego w kolejnych. Aby przygotować się do tego rozdziału, należy zastąpić zawartość pliku *index.ts* w katalogu *src* kodem przedstawionym na listingu 13.1.

Tabela 13.2. Opcje kompilatora TypeScriptu użyte w rozdziale

Opcja	Opis
declaration	Ta opcja generuje pliki deklaracji typu, które pomagają zrozumieć, w jaki sposób zostały ustalone typy dla kodu JavaScriptu. Wspomniane pliki zostaną szczegółowo omówione w rozdziale 14.
downlevelIteration	Ta opcja włącza obsługę iteratorów, gdy kod jest przeznaczony dla starszych wersji języka JavaScript
outDir	Ta opcja określa katalog, w którym zostaną umieszczone wygenerowane pliki JavaScriptu
rootDir	Ta opcja określa katalog główny używany przez kompilator do wyszukiwania plików TypeScriptu
target	Ta opcja określa docelową wersję języka JavaScript, dla której kompilator wygeneruje kod

Listing 13.1. Nowa zawartość pliku *index.ts* w katalogu *src*

```
import { City, Person, Product, Employee } from "../dataTypes";

let products = [new Product("Buty do biegania", 100), new Product("Czapka", 25)];

type shapeType = { name: string };

class Collection<T extends shapeType> {
    constructor(private items: T[] = []) {}

    add(...newItems: T[]): void {
        this.items.push(...newItems);
    }

    get(name: string): T {
        return this.items.find(item => item.name === name);
    }

    get count(): number {
        return this.items.length;
    }
}

let productCollection: Collection<Product> = new Collection(products);
console.log(`Liczba produktów: ${ productCollection.count }`);
let p = productCollection.get("Czapka");
console.log(`Produkt: ${ p.name }, ${ p.price }`);
```

Otwórz nowe okno powłoki, przejdź do katalogu *types*, a następnie z jego poziomu wydaj polecenie przedstawione na listingu 13.2, uruchamiające kompilator TypeScriptu w trybie automatycznego kompilowania kodu po wykryciu zmiany w dowolnym pliku i wykonywania kodu po jego skompilowaniu.

Listing 13.2. Uruchamianie kompilatora TypeScriptu

```
$ npm start
```

■ **Wskazówka** Przykładowy projekt dla tego rozdziału, a także projekty dla pozostałych rozdziałów książki zostały umieszczone w archiwum dostępnym pod adresem <https://ftp.helion.pl/przyklady/tys4o2.zip>.

Kompilator przeprowadzi kompilację kodu w pliku *index.ts*, wykona wygenerowany kod JavaScriptu, a następnie przejdzie do trybu monitorowania i wyświetli następujące dane wyjściowe:

```
7:31:10 AM - Starting compilation in watch mode...
7:31:11 AM - Found 0 errors. Watching for file changes.
Liczba produktów: 2
Produkt: Czapka, 25
```

Używanie kolekcji generycznych

TypeScript zapewnia obsługę kolekcji JavaScriptu za pomocą parametrów typu generycznego, co pozwala klasie generycznej na bezpieczne używanie kolekcji zgodnie z opisem przedstawionym w tabeli 13.3. Klasy kolekcji JavaScriptu pokrótce omówiłem w rozdziale 4.

Tabela 13.3. Typy kolekcji generycznych

Nazwa	Opis
Map<K, V>	Opisuje kolekcję Map, której typem klucza jest K, a typem wartości jest V
ReadonlyMap<K, V>	Opisuje kolekcję Map, która nie może być modyfikowana
Set<T>	Opisuje kolekcję Set, której typem wartości jest T
ReadonlySet<T>	Opisuje kolekcję Set, która nie może być modyfikowana

Na listingu 13.3 pokazałem, jak klasa generyczna używa parametrów typu z kolekcją.

Listing 13.3. Używanie kolekcji w kodzie pliku *index.ts* w katalogu *src*

```
import { City, Person, Product, Employee } from "../dataTypes";

let products = [new Product("Buty do biegania", 100), new Product("Czapka", 25)];

type shapeType = { name: string };

class Collection<T extends shapeType> {
    private items: Set<T>;

    constructor(initialItems: T[] = []) {
        this.items = new Set<T>(initialItems);
    }
}
```

```

    add(...newItems: T[]): void {
        newItems.forEach(newItem => this.items.add(newItem));
    }

    get(name: string): T {
        return [...this.items.values()].find(item => item.name === name);
    }
    get count(): number {
        return this.items.size;
    }
}

let productCollection: Collection<Product> = new Collection(products);
console.log(`Liczba produktów: ${ productCollection.count }`);
let p = productCollection.get("Czapka");
console.log(`Produkt: ${ p.name }, ${ p.price }`);

```

Klasa `Collection<T>` została zmieniona na `Set<T>` w celu przechowywania elementów i wykorzystuje dla kolekcji parametr typu generycznego. Kompilator TypeScript używa parametru typu, by uniknąć dodawania danych innego typu do zbioru, a podczas pobierania obiektów z kolekcji nie ma konieczności stosowania wartownika typu. To samo podejście można zastosować z obiektem typu `Map`, jak pokazałem na listingu 13.4.

Listing 13.4. *Używanie kolekcji Map w kodzie pliku index.ts w katalogu src*

```

import { City, Person, Product, Employee } from "../dataTypes";

let products = [new Product("Buty do biegania", 100), new Product("Czapka", 25)];

type shapeType = { name: string };

class Collection<T extends shapeType> {
    private items: Map<string, T>;

    constructor(initialItems: T[] = []) {
        this.items = new Map<string, T>();
        this.add(...initialItems);
    }

    add(...newItems: T[]): void {
        newItems.forEach(newItem => this.items.set(newItem.name, newItem));
    }

    get(name: string): T {
        return this.items.get(name);
    }

    get count(): number {
        return this.items.size;
    }
}

let productCollection: Collection<Product> = new Collection(products);
console.log(`Liczba produktów: ${ productCollection.count }`);
let p = productCollection.get("Czapka");
console.log(`Produkt: ${ p.name }, ${ p.price }`);

```

Klasa generyczna nie musi dostarczać parametrów typu generycznego dla kolekcji i zamiast tego może wskazywać konkretny typ. W omawianym przykładzie typ `Map` jest używany do przechowywania obiektów z wykorzystaniem klucza w postaci właściwości `name`. Właściwość `name` może być stosowana bezpiecznie, ponieważ jest częścią ograniczenia nałożonego na parametr typu o nazwie `T`. Kod przedstawiony na listingu 13.4 powoduje wygenerowanie następujących danych wyjściowych:

```
Liczba produktów: 2
Produkt: Czapka, 25
```

Używanie iteratorów generycznych

Jak wyjaśniłem w rozdziale 4., iteratory pozwalają na wyliczanie sekwencji wartości, a obsługa iteratorów jest często dostępną funkcjonalnością klas operujących na innych typach, np. kolekcji. Do opisywania iteratorów i wyniku ich działania TypeScript oferuje interfejsy wymienione w tabeli 13.4.

Tabela 13.4. Interfejsy iteratora w języku TypeScript

Interfejs	Opis
<code>Iterator<T></code>	Ten interfejs opisuje iterator, którego metoda <code>next()</code> zwraca obiekt <code>IteratorResult<T></code>
<code>IteratorResult<T></code>	Ten interfejs opisuje wynik wygenerowany przez iterator z właściwościami <code>done</code> i <code>value</code>
<code>Iterable<T></code>	Ten interfejs definiuje obiekt, który ma właściwość <code>Symbol.iterator</code> i bezpośrednio obsługuje iterację
<code>IterableIterator<T></code>	Ten interfejs łączy interfejsy <code>Iterator<T></code> i <code>Iterable<T></code> w celu opisanego obiektu, który ma właściwość <code>Symbol.iterator</code> oraz definiuje metodę <code>next()</code> i właściwość <code>result</code>

Na listingu 13.5 możesz zobaczyć przykład użycia interfejsów `Iterator<T>` i `IteratorResult<T>` w celu zapewnienia dostępu do zawartości `Map<string, T>` wykorzystywanej do przechowywania obiektów przez klasę `Collection<T>`.

Listing 13.5. Iterowanie obiektów w kodzie pliku `index.ts` w katalogu `src`

```
import { City, Person, Product, Employee } from "../dataTypes";

let products = [new Product("Buty do biegania", 100), new Product("Czapka", 25)];

type shapeType = { name: string };

class Collection<T> extends shapeType {
    private items: Map<string, T>;

    constructor(initialItems: T[] = []) {
        this.items = new Map<string, T>();
    }
}
```

```

    this.add(...initialItems);
  }

  add(...newItems: T[]): void {
    newItems.forEach(newItem => this.items.set(newItem.name, newItem));
  }

  get(name: string): T {
    return this.items.get(name);
  }

  get count(): number {
    return this.items.size;
  }

  values(): Iterator<T> {
    return this.items.values();
  }
}

let productCollection: Collection<Product> = new Collection(products);
console.log(`Liczba produktów: ${ productCollection.count }`);

let iterator: Iterator<Product> = productCollection.values();
let result: IteratorResult<Product> = iterator.next();
while (!result.done) {
  console.log(`Produkt: ${result.value.name}, ${ result.value.price}`);
  result = iterator.next();
}

```

Metoda `values()` definiowana przez klasę `Collection<T>` zwraca `Iterator<T>`. Gdy ta metoda jest wywoływana w obiekcie `Collection<Product>`, zwracany przez nią iterator za pomocą metody `next()` generuje obiekty `IteratorResult<Product>`. Właściwość `result` każdego obiektu `IteratorResult<Product>` będzie zwracała obiekt `Product`, pozwalając na iterację przez zarządzane obiekty. Kod przedstawiony na listingu 13.5 powoduje wygenerowanie następujących danych wyjściowych:

```

Liczba produktów: 2
Produkt: Buty do biegania, 100
Produkt: Czapka, 25

```

Używanie iteratorów z JavaScriptem ES5 i wcześniejszymi wersjami

Iteratory zostały wprowadzone w standardzie JavaScript ES6. Jeżeli używasz iteratorów w projekcie i chcesz wygenerować kod przeznaczony dla wcześniejszych wersji JavaScriptu, wówczas musisz przypisać wartość `true` właściwości kompilatora `downlevelIteration`.

Łączenie iteratora i obiektu możliwego do iteracji

Interfejs `IterableIterator<T>` może być używany do opisywania obiektów, które mogą być iterowane, a także definiuje właściwość `Symbol.iterator`. Obiekt implementujący ten interfejs można wykorzystać w wyliczeniu w bardziej elegancki sposób, jak pokazałem na listingu 13.6.

Listing 13.6. Używanie interfejsu `IterableIterator<T>` w kodzie pliku `index.ts` w katalogu `src`

```
import { City, Person, Product, Employee } from "../dataTypes";

let products = [new Product("Buty do biegania", 100), new Product("Czapka", 25)];

type shapeType = { name: string };

class Collection<T extends shapeType> {
  private items: Map<string, T>;

  constructor(initialItems: T[] = []) {
    this.items = new Map<string, T>();
    this.add(...initialItems);
  }

  add(...newItems: T[]): void {
    newItems.forEach(newItem => this.items.set(newItem.name, newItem));
  }

  get(name: string): T {
    return this.items.get(name);
  }

  get count(): number {
    return this.items.size;
  }

  values(): IterableIterator<T> {
    return this.items.values();
  }
}

let productCollection: Collection<Product> = new Collection(products);
console.log(`Liczba produktów: ${ productCollection.count }`);

[...productCollection.values()].forEach(p =>
  console.log(`Produkt: ${p.name}, ${ p.price}`));
```

Metoda `values()` zwraca obiekt typu `IterableIterator`, co jest możliwe, ponieważ wynik działania metody `Map()` definiuje wszystkie elementy składowe wymienione przez interfejs. Połączony interfejs pozwala na bezpośrednie iterowanie wyniku działania metody `values()`. W kodzie na omawianym listingu operator rozwinęcia został użyty do wypełnienia tablicy, a następnie wyliczenia jej zawartości za pomocą metody `forEach()`. Kod przedstawiony na listingu 13.6 powoduje wygenerowanie następujących danych wyjściowych:

```
Liczba produktów: 2
Produkt: Buty do biegania, 100
Produkt: Czapka, 25
```

Tworzenie klasy umożliwiającej iterację

Klasa definiująca właściwość `Symbol.iterator` może implementować interfejs `Iterable<T>` pozwalający na iterację bez konieczności wywoływania metody lub odczytywania właściwości, jak pokazałem w przykładzie na listingu 13.7.

Listing 13.7. Tworzenie klasy pozwalającej na iterację w kodzie pliku `index.ts` w katalogu `src`

```
import { City, Person, Product, Employee } from "../dataTypes";

let products = [new Product("Buty do biegania", 100), new Product("Czapka", 25)];

type shapeType = { name: string };

class Collection<T extends shapeType> implements Iterable<T> {
  private items: Map<string, T>;

  constructor(initialItems: T[] = []) {
    this.items = new Map<string, T>();
    this.add(...initialItems);
  }

  add(...newItems: T[]): void {
    newItems.forEach(newItem => this.items.set(newItem.name, newItem));
  }

  get(name: string): T {
    return this.items.get(name);
  }

  get count(): number {
    return this.items.size;
  }

  [Symbol.iterator]() : Iterator<T> {
    return this.items.values();
  }
}

let productCollection: Collection<Product> = new Collection(products);
console.log(`Liczba produktów: ${ productCollection.count }`);

[...productCollection].forEach(p => console.log(`Produkt: ${p.name}, ${ p.price}`));
```

Nowa właściwość implementuje interfejs `Iterable<T>` i wskazuje na zdefiniowanie właściwości `Symbol.iterator`, która zwraca obiekt `Iterator<T>` możliwy do wykorzystania podczas iteracji. Kod przedstawiony na listingu 13.7 powoduje wygenerowanie następujących danych wyjściowych:

```
Liczba produktów: 2
Produkt: Buty do biegania, 100
Produkt: Czapka, 25
```

Używanie typów indeksu

Klasa `Collection<T>` ogranicza akceptowane typy przez wykorzystanie kształtu typu gwarantującego, że wszystkie używane obiekty będą miały właściwość `name` stosowaną w charakterze klucza pozwalającego na przechowywanie obiektów w egzemplarzu `Map` i pobieranie ich z niego.

TypeScript oferuje zbiór powiązanych ze sobą funkcji pozwalających na zdefiniowanie właściwości przez dowolny obiekt, który będzie używany jako klucz i jednocześnie zostanie zapewnione bezpieczeństwo typu. Te funkcje mogą być trudne do opanowania, więc najpierw pokażę sposób ich działania, a dopiero później wykorzystam je do usprawnienia klasy `Collection<T>`.

Używanie zapytania typu indeksu

Słowo kluczowe `keyof`, określane mianem operatora *zapytania typu indeksu*, zwraca unię nazw właściwości typu, używając do tego omówionej w rozdziale 9. funkcjonalności literału wartości typu. Przykład zastosowania tego słowa kluczowego w klasie `Product` widać na listingu 13.8.

Listing 13.8. Używanie operatora zapytania typu indeksu w kodzie pliku `index.ts` w katalogu `src`

```
import { City, Person, Product, Employee } from "../dataTypes";
```

```
let myVar: keyof Product = "name";
myVar = "price";
myVar = "someOtherName";
```

Adnotacją typu dla zmiennej `myVar` jest `keyof Product`, czyli unia nazw właściwości zdefiniowanych przez klasę `Product`. W efekcie zmiennej `myVar` można przypisać jedynie wartości w postaci ciągu tekstowego właściwości `name` i `price`, ponieważ są to nazwy dwóch właściwości zdefiniowanych przez klasę `Product` w pliku `dataTypes.ts` utworzonym w rozdziale 12.

```
...
export class Product {
  constructor(public name: string, public price: number) {}
}
...
```

Próba przypisania innej wartości zmiennej `myVar`, jak w ostatnim poleceniu na listingu 13.8, spowoduje wygenerowanie błędu podczas kompilacji:

```
src/index.ts(34,1): error TS2322: Type '"someOtherName"' is not assignable to type '"name" | "price"'.
  ↳| "price" |.
```

Słowo kluczowe `keyof` może zostać użyte do ograniczenia parametrów typu generycznego, aby mogły być stosowane jedynie w celu dopasowania właściwości innych typów, jak pokazałem na listingu 13.9.

Listing 13.9. Ograniczanie parametru typu generycznego w kodzie pliku `index.ts` w katalogu `src`

```
import { City, Person, Product, Employee } from "../dataTypes";
```

```
function getValue<T, K extends keyof T>(item: T, keyname: K) {
  console.log(`Wartość: ${item[keyname]}`);
```

```

}

let p = new Product("Buty do biegania", 100);
getValue(p, "name");
getValue(p, "price");

let e = new Employee("Bartek Nowak", "handlowiec");
getValue(e, "name");
getValue(e, "role");

```

W omawianym przykładzie została zdefiniowana funkcja o nazwie `getValue()`, której parametr typu `K` został ograniczony za pomocą `typeof T`, co oznacza, że `K` może być nazwą tylko jednej właściwości definiowanej przez `T`, niezależnie od typu używanego dla `T` podczas wywołania funkcji. Gdy funkcja `getValue()` jest używana z obiektem `Product`, parametrem `keyname` może być jedynie `name` lub `price`. Natomiast jeśli funkcja `getValue()` jest używana z obiektem `Employee`, parametrem `keyname` może być tylko `name` lub `role`. W obu przypadkach parametr `keyname` można wykorzystać w celu bezpiecznego pobrania lub zdefiniowania wartości odpowiedniej właściwości obiektu `Product` lub `Employee`. Kod przedstawiony na listingu 13.9 powoduje wygenerowanie następujących danych wyjściowych:

```

Wartość: Buty do biegania
Wartość: 100
Wartość: Bartek Nowak
Wartość: handlowiec

```

Jawne dostarczanie parametrów typu generycznego dla typów indeksu

Metoda `getValue()` została na listingu 13.9 wywołana bez argumentów typu generycznego, co pozwala kompilatorowi na określenie typu na podstawie argumentów funkcji. Jawne określenie typu argumentów pokazuje aspekt używania operatora zapytania typu indeksu, jak pokazałem w przykładzie na listingu 13.10.

Listing 13.10. *Jawne używanie argumentów typu w kodzie pliku `index.ts` w katalogu `src`*

```

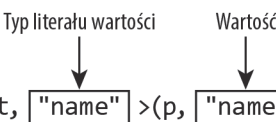
import { City, Person, Product, Employee } from "../dataTypes";

function getValue<T, K extends keyof T>(item: T, keyname: K) {
    console.log(`Wartość: ${item[keyname]}`);
}

let p = new Product("Buty do biegania", 100);
getValue<Product, "name">(p, "name");
getValue(p, "price");
let e = new Employee("Bartek Nowak", "handlowiec");
getValue(e, "name");
getValue(e, "role");

```

Można odnieść wrażenie, że właściwość wymagana w tym przykładzie została podana dwukrotnie, ale `name` ma różne zastosowania w zmodyfikowanym poleceniu, jak pokazałem na rysunku 13.1.



```
getValue<Product, "name">(p, "name");
```

Rysunek 13.1. Typ indeksu i wartość

Argument typu generycznego `name` to literał wartości określający jeden z typów `keyof Product` i używany przez kompilator TypeScriptu do sprawdzenia typu. Argument funkcji `name` to wartość w postaci ciągu tekstowego, która jest używana przez środowisko uruchomieniowe JavaScriptu podczas wykonywania kodu. Kod przedstawiony na listingu 13.10 powoduje wygenerowanie następujących danych wyjściowych:

```
Wartość: Buty do biegania
Wartość: 100
Wartość: Bartek Nowak
Wartość: handlowiec
```

Używanie zindeksowanego operatora dostępu

Zindeksowany operator dostępu jest wykorzystywany w celu pobrania typu jednej lub więcej właściwości, jak widać na listingu 13.11.

Listing 13.11. Używanie zindeksowanego operatora dostępu w kodzie pliku `index.ts` w katalogu `src`

```
import { City, Person, Product, Employee } from "../dataTypes";

function getValue<T, K extends keyof T>(item: T, keyname: K) {
    console.log(`Wartość: ${item[keyname]}`);
}
```

```
type priceType = Product["price"];
type allTypes = Product[keyof Product];
```

```
let p = new Product("Buty do biegania", 100);
getValue<Product, "name">(p, "name");
getValue(p, "price");
```

```
let e = new Employee("Bartek Nowak", "handlowiec");
getValue(e, "name");
getValue(e, "role");
```

Zindeksowany operator dostępu jest wyrażany za pomocą nawiasu kwadratowego po typie, więc np. `Product["price"]` to liczba (`number`), ponieważ jest to typ właściwości `price` zdefiniowanej przez klasę `Product`. Zindeksowany operator dostępu działa z literałami wartości typów, co oznacza, że może być stosowany w zapytaniach typu indeksu:

```
...
type allTypes = Product[keyof Product];
...
```

Wyrażenie `keyof Product` zwraca literał unii wartości typu z nazwami właściwości zdefiniowanymi przez klasę `Product`, czyli `"name" | "price"`. Zindeksowany operator dostępu zwraca unię w postaci typów tych właściwości, np. `Product[keyof Product]` to `string | number`, która w rzeczywistości jest unią typów właściwości `name` i `price`.

■ **Wskazówka** Typy zwracane przez zindeksowany operator dostępu są nazywane *typami wyszukiwania*.

Zindeksowany operator dostępu jest najczęściej używany z typami generycznymi, co pozwala na bezpieczne obsługiwanie typów, nawet jeśli konkretne typy, które mają być zastosowane, pozostają nieznane. Spójrz na przykład przedstawiony na listingu 13.12.

Listing 13.12. Używanie zindeksowanego operatora dostępu z typem generycznym w kodzie pliku *index.ts* w katalogu *src*

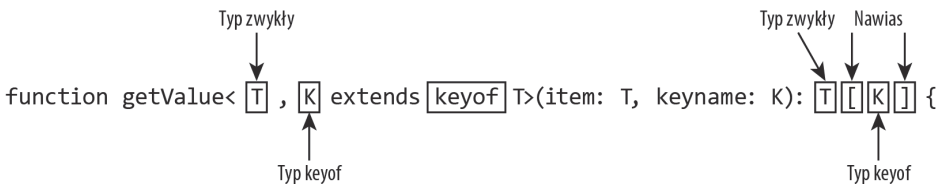
```
import { City, Person, Product, Employee } from "../dataTypes";

function getValue<T, K extends keyof T>(item: T, keyname: K): T[K] {
    return item[keyname];
}

let p = new Product("Buty do biegania", 100);
console.log(getValue<Product, "name">(p, "name"));
console.log(getValue(p, "price"));

let e = new Employee("Bartek Nowak", "handlowiec");
console.log(getValue(e, "name"));
console.log(getValue(e, "role"));
```

Zindeksowany operator dostępu jest wyrażany za pomocą zwykłego typu, typu `keyof` i nawiasu kwadratowego, jak pokazałem na rysunku 13.2.



Rysunek 13.2. Zindeksowany operator dostępu

Zindeksowany operator dostępu na listingu 13.12, `T[K]`, wskazuje kompilatorowi, że wynik działania funkcji `getValue()` będzie typu właściwości, której nazwa została podana za pomocą argumentu `keyof`. W takim przypadku to kompilator ma ustalić typ na podstawie argumentu typu generycznego użytego podczas wywołania funkcji. Dla obiektu `Product` oznacza to, że argument `name` wygeneruje wynik typu `string`, a argument `price` wygeneruje wynik typu `number`. Kod przedstawiony na listingu 13.12 powoduje wygenerowanie następujących danych wyjściowych:

```
Buty do biegania
100
Bartek Nowak
handlowiec
```

Używanie typu indeksu dla klasy Collection<T>

Używanie typu indeksu pozwala na zmianę klasy `Collection<T>` w taki sposób, aby przechowywała obiekty dowolnego typu, a nie tylko zdefiniowanego przez właściwość `name`. Na listingu 13.13 przedstawiłem zmiany konieczne do wprowadzenia w klasie, która teraz będzie wykorzystywać zapytanie typu indeksu w celu ograniczenia właściwości konstruktora `propertyName` do nazw właściwości definiowanych przez parametr typu generycznego `T`. Zapewnia to klucz, według którego obiekty mogą być przechowywane w egzemplarzu `Map`.

Listing 13.13. Używanie typu indeksu w klasie kolekcji w kodzie pliku `index.ts` w katalogu `src`

```
import { City, Person, Product, Employee } from "../dataTypes";
```

```
let products = [new Product("Buty do biegania", 100), new Product("Czapka", 25)];
```

```
//type shapeType = { name: string};
```

```
class Collection<T, K extends keyof T> implements Iterable<T> {
    private items: Map<T[K], T>;

    constructor(initialItems: T[] = [], private propertyName: K) {
        this.items = new Map<T[K], T>();
        this.add(...initialItems);
    }

    add(...newItems: T[]): void {
        newItems.forEach(newItem =>
            this.items.set(newItem[this.propertyName], newItem));
    }

    get(key: T[K]): T {
        return this.items.get(key);
    }

    get count(): number {
        return this.items.size;
    }

    [Symbol.iterator](): Iterator<T> {
        return this.items.values();
    }
}
```

```
let productCollection: Collection<Product, "name">
    = new Collection(products, "name");
console.log(`Liczba produktów: ${ productCollection.count }`);
```

```
let itemByKey = productCollection.get("Czapka");
console.log(`Element: ${ itemByKey.name}, ${ itemByKey.price}`);
```

Klasa została zmodyfikowana i zawiera teraz dodatkowy parametr typu generycznego `K`, ograniczony do `keyof T`, który jest typem danych obiektów przechowywanych w kolekcji. Nowy egzemplarz `Collection<T, K>` jest tworzony w przedstawiony tutaj sposób:

```
...
let productCollection: Collection<Product, "name">
    = new Collection(products, "name");
...
```

Kod przedstawiony na listingu 13.13 powoduje wygenerowanie następujących danych wyjściowych:

```
Liczba produktów: 2
Element: Czapka, 25
```

Zagęszczenie nawiasów ostrych i kwadratowych na listingu 13.13 może być trudne do zrozumienia, gdy po raz pierwszy spotykasz się z używaniem typów indeksu. Aby nieco pomóc w zrozumieniu tego kodu, w tabeli 13.5 wymienilem typy znaczące i parametry konstruktora, a także typy stosowane dla obiektu `Collection<Product, "name">` utworzonego w tym przykładzie.

Tabela 13.5. Ważne typy używane w klasie `Collection<T>`

Nazwa typu	Opis
T	Typ obiektów przechowywanych w klasie kolekcji, która jest podawana przez pierwszy argument typu generycznego. W przypadku obiektu tworzonego na omawianym listingu to <code>Product</code>
K	Nazwa właściwości klucza ograniczonego do nazw właściwości definiowanych przez T. Wartość dla tego typu jest dostarczana przez drugi argument typu generycznego, którym w omawianym przykładzie jest <code>name</code>
T[K]	Typ właściwości klucza pobieranej za pomocą zindeksowanego operatora dostępu. Ta wartość jest używana do określenia typu klucza podczas tworzenia obiektu <code>Map</code> i do ograniczenia typu parametrów. Typem właściwości <code>Product.name</code> dla obiektu tworzonego przez omawiany kod jest <code>string</code>
propertyName	Nazwa właściwości klucza, która jest wymagana jako wartość możliwa do wykorzystania przez środowisko uruchomieniowe JavaScriptu po usunięciu informacji typu generycznego w TypeScriptie. W przypadku obiektu tworzonego przez omawiany kod wartością jest <code>name</code> i odpowiada ona typowi generycznemu K

Wynik zastosowania typu indeksu na listingu 13.13 jest taki, że dowolna właściwość może zostać użyta do przechowywania obiektów, które z kolei mogą być dowolnego typu. Na listingu 13.14 zmienił się sposób tworzenia egzemplarza klasy `Collection<T, K>`, a właściwość `price` została użyta jako klucz. W kodzie przedstawionym na listingu zostały pominięte argumenty typu generycznego, więc kompilator będzie musiał ustalić wymagane typy.

Listing 13.14. Zmienianie właściwości klucza w kodzie pliku `index.ts` w katalogu `src`

```
...
let productCollection = new Collection(products, "price");
console.log(`Liczba produktów: ${ productCollection.count }`);
```

```
let itemByKey = productCollection.get(100);
console.log(`Element: ${ itemByKey.name}, ${ itemByKey.price}`);
...
```

Typ argumentu metody `get()` zmienił się, aby został dopasowany do typu właściwości klucza, co pozwoli na pobieranie obiektów za pomocą argumentu typu `number`. Kod przedstawiony na listingu 13.14 powoduje wygenerowanie następujących danych wyjściowych:

```
Liczba produktów: 2
Element: Buty do biegania, 100
```

Używanie mapowania typu

Mapowanie typu jest tworzone przez zastosowanie transformacji na właściwości istniejącego typu. Najlepszym sposobem na zrozumienie tego, jak działa mapowanie typu, jest utworzenie przykładowego mapowania, które będzie odpowiedzialne za przetwarzanie typu, ale nie wprowadzi żadnych zmian, jak pokazałem na listingu 13.15.

Listing 13.15. Używanie typu mapowanego w kodzie pliku `index.ts` w katalogu `src`

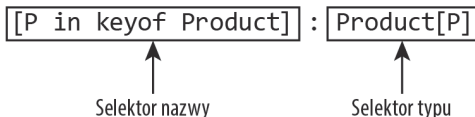
```
import { City, Person, Product, Employee } from "../dataTypes";
```

```
type MappedProduct = {
  [P in keyof Product] : Product[P]
};
```

```
let p: MappedProduct = { name: "Kajak", price: 275};
console.log(`Typ mapowany: ${p.name}, ${p.price}`);
```

Mapowanie typu to wyrażenie pobierające nazwy właściwości przeznaczonych do dołączenia w mapowanym typie oraz typ dla każdej z nich, jak pokazałem na rysunku 13.3.

```
type MappedProduct = {
  [P in keyof Product] : Product[P]
};
```



Rysunek 13.3. Przykład typu mapowanego

Selektor nazwy właściwości definiuje parametr typu (w omawianym przykładzie to `P`) oraz wykorzystuje słowo kluczowe `in` do wyliczenia typów w unii wartości literalnych. Unia typu może zostać wyrażona bezpośrednio, np. `"name" | "price"`, lub za pomocą `keyof`.

Kompilator TypeScriptu tworzy w typie mapowanym nową właściwość dla każdego typu unii. Typ poszczególnych właściwości jest określany przez selektor typu, który można pobrać z typu źródłowego za pomocą zindeksowanego operatora dostępu z `P` jako literałem wartości do wyszukania.

Typ `MappedProduct` na listingu 13.15 używa `keyof` do pobrania właściwości zdefiniowanych przez klasę `Product` oraz przez zindeksowany operator typu do pobrania typu tych właściwości. Wynikiem jest odpowiednik następującego typu:

```
type MappedProduct = {
  name: string;
  price: number;
}
```

Kod przedstawiony na listingu 13.15 powoduje wygenerowanie następujących danych wyjściowych:

```
Typ mapowany: Kajak, 275
```

Zmiana mapowanych nazw i typów

W poprzednim przykładzie podczas mapowania zostały zachowane nazwy i typy. Jednak mapowanie jest znacznie elastyczniejsze, więc pozwala na zmianę nazwy i typu mapowanej właściwości, jak pokazałem na listingu 13.16.

Listing 13.16. Zmiana nazwy i typu mapowanej właściwości w kodzie pliku *index.ts* w katalogu *src*

```
import { City, Person, Product, Employee } from "../dataTypes";

type MappedProduct = {
  [P in keyof Product] : Product[P]
};

let p: MappedProduct = { name: "Kajak", price: 275};
console.log(`Typ mapowany: ${p.name}, ${p.price}`);

type AllowStrings = {
  [P in keyof Product] : Product[P] | string
}
let q: AllowStrings = { name: "Kajak", price: "jabłko" };
console.log(`Typ zmieniony # 1: ${q.name}, ${q.price}`);

type ChangeNames = {
  [P in keyof Product as `${P}Property`] : Product[P]
}

let r: ChangeNames = { nameProperty: "Kajak", priceProperty: 12 };
console.log(`Typ zmieniony # 2: ${r.nameProperty}, ${r.priceProperty}`);
```

Typ `AllowStrings` został utworzony w mapowaniu definiującym unię typów `string` i pierwotnego typu właściwości w następujący sposób:

```
...
[P in keyof Product] : Product[P] | string
...
```

Wynikiem będzie typ odpowiadający następującemu:

```
type AllowStrings = {
  name: string;
  price: number | string;
}
```

Typ `ChangeNames` jest tworzony z mapowaniem zmieniającym nazwę poszczególnych właściwości przez dodanie wyrażenia `Property` w następujący sposób:

```
...
[P in keyof Product as `${P}Property`] : Product[P]
...
```

Słowo kluczowe `as` jest powiązane z wyrażeniem definiującym nazwę właściwości. W omawianym przykładzie do zmodyfikowania istniejącej nazwy został użyty szablon ciągu tekstowego, a wynikiem jest typ odpowiadający następującemu:

```
type ChangeNames = {
  nameProperty: string;
  priceProperty: number;
}
```

Kod zamieszczony na listingu 13.16 po skompilowaniu i uruchomieniu powoduje wygenerowanie następujących danych wyjściowych:

```
Typ: Kajak, 275
Typ zmieniony # 1: Kajak, jabłka
Typ zmieniony # 2: Kajak, 12
```

Używanie parametru typu generycznego z typem mapowanym

Typy mapowane stają się znacznie użyteczniejsze, gdy definiują parametry typu generycznego, jak pokazałem na listingu 13.17.

Listing 13.17. *Używanie parametru typu generycznego w kodzie pliku `index.ts` w katalogu `src`*

```
import { City, Person, Product, Employee } from "../dataTypes";
```

```
type Mapped<T> = {
  [P in keyof T] : T[P]
};
```

```
let p: Mapped<Product> = { name: "Kajak", price: 275};
console.log(`Typ mapowany: ${p.name}, ${p.price}`);
```

```
let c: Mapped<City> = { name: "Londyn", population: 8136000};
console.log(`Typ mapowany: ${c.name}, ${c.population}`);
```

Typ `Mapped<T>` definiuje parametr typu generycznego o nazwie `T`, który jest typem przeznaczonym do transformacji. Parametr typu jest użyty w nazwie oraz w selektorach typu, więc dowolny typ można mapować za pomocą parametru typu generycznego. W przykładzie na listingu 13.17 typ mapowany `Mapped<T>` został użyty dla klas `Product` i `City`, a efektem działania omawianego kodu jest wygenerowanie następujących danych wyjściowych:

```
Typ mapowany: Kajak, 275
Typ mapowany: Londyn, 8136000
```

Mapowanie a konstruktor i metody

Mapowanie operuje wyłącznie na właściwościach. Po zastosowaniu dla klasy mapowanie typu prowadzi do powstania kształtu typu zawierającego właściwości, ale pozbawionego konstruktora i implementacji metod. Na przykład następująca klasa:

```
class MyClass {
  constructor(public name: string) {}
  getName(): string {
    return this.name;
  }
}
```

zostaje na listingu 13.17 mapowana na typ `Mapping<T>`:

```
{
  name: string;
  getName: () => string;
}
```

Mapowanie typu powoduje wygenerowanie kształtu, który może być stosowany dla literałów obiektów, implementowany przez klasy lub rozszerzany przez interfejsy. Jednak wynikiem mapowania typu nie jest klasa.

Zmiana modyfikowalności i opcjonalności właściwości

Typ mapowany może zmieniać właściwości, aby stały się opcjonalne lub wymagane, a także może dodawać lub usuwać słowo kluczowe `readonly`, jak pokazałem na listingu 13.18.

Listing 13.18. Zmiana właściwości w pliku `index.ts` w katalogu `src`

```
import { City, Person, Product, Employee } from "../dataTypes";
```

```
type MakeOptional<T> = {
  [P in keyof T]? : T[P]
};

type MakeRequired<T> = {
  [P in keyof T]-? : T[P]
};

type MakeReadOnly<T> = {
  readonly [P in keyof T] : T[P]
};

type MakeReadWrite<T> = {
  -readonly [P in keyof T] : T[P]
};

type optionalType = MakeOptional<Product>;
type requiredType = MakeRequired<optionalType>;
type readOnlyType = MakeReadOnly<requiredType>;
type readWriteType = MakeReadWrite<readOnlyType>;
```

```
let p: readWriteType = { name: "Kajak", price: 275};
console.log(`Typ mapowany: ${p.name}, ${p.price}`);
```

Znak zapytania zostaje umieszczony po nazwie selektora, aby właściwość w typie mapowanym była opcjonalna. Znaki minusa i zapytania są używane do określenia właściwości jako wymaganej. Natomiast właściwość będzie oznaczona jako tylko do odczytu lub tylko do zapisu przez poprzedzenie nazwy selektora słowem kluczowym odpowiednio `readonly` lub `-readonly`.

Typy mapowane zmieniają wszystkie właściwości definiowane przez typ i przeprowadzają ich transformację, aby typ wygenerowany przez `MakeOptional<T>` po zastosowaniu w klasie np. `Product` był odpowiednikiem dla następującego:

```
type optionalType = {
  name?: string;
  price?: number;
}
```

Typy wygenerowane przez mapowanie mogą być przekazywane innym mapowaniom, tworząc w ten sposób łańcuch transformacji. W omawianym przykładzie typ wygenerowany przez mapowanie `MakeOptional<T>` zostaje następnie transformowany przez mapowanie `MakeRequired<T>`, którego dane wyjściowe są z kolei przekazywane do mapowania `MakeReadOnly<T>`, a później do mapowania `MakeReadWrite<T>`. W efekcie właściwość jest najpierw opcjonalna, później wymagana, dalej tylko do odczytu i na końcu do odczytu i zapisu.

Używanie wbudowanych typów mapowanych

TypeScript oferuje wbudowane typy mapowane, z których część odpowiada transformacjom przedstawionym na listingu 13.18, a inne będą omówione w dalszej części rozdziału.

Mapowania wbudowane wymienilem w tabeli 13.6.

Tabela 13.6. Wbudowane mapowania typu dla właściwości opcjonalnych i tylko do odczytu

Nazwa mapowania	Opis
<code>Partial<T></code>	To mapowanie powoduje, że właściwość jest opcjonalna
<code>Required<T></code>	To mapowanie powoduje, że właściwość jest wymagana
<code>ReadOnly<T></code>	To mapowanie dodaje do właściwości słowo kluczowe <code>readonly</code>
<code>Pick<T, K></code>	To mapowanie wybiera określone właściwości do utworzenia nowego typu zgodnie z opisem przedstawionym w sekcji „Mapowanie określonych właściwości”
<code>Omit<T, klucze></code>	To mapowanie wybiera określone właściwości do utworzenia nowego typu zgodnie z opisem przedstawionym w sekcji „Mapowanie określonych właściwości”
<code>Record<T, K></code>	To mapowanie tworzy typ bez transformacji istniejącego zgodnie z opisem w sekcji „Tworzenie typu z użyciem mapowania”

Nie ma wbudowanego mapowania pozwalającego na usunięcie słowa kluczowego `readonly`. Na listingu 13.19 pokazałem, jak zastąpić mapowanie niestandardowe mapowaniem dostarczonym przez TypeScript.

Listing 13.19. Używanie wbudowanego w TypeScript mapowania w kodzie pliku *index.ts* w katalogu *src*

```
import { City, Person, Product, Employee } from "../dataTypes";

// type MakeOptional<T> = {
//   [P in keyof T]?: T[P]
// };

// type MakeRequired<T> = {
//   [P in keyof T]-?: T[P]
// };

// type MakeReadOnly<T> = {
//   readonly [P in keyof T]: T[P]
// };

type MakeReadWrite<T> = {
  -readonly [P in keyof T]: T[P]
};

type optionalType = Partial<Product>;
type requiredType = Required<optionalType>;
type readOnlyType = Readonly<requiredType>;
type readWriteType = MakeReadWrite<readOnlyType>

let p: readWriteType = { name: "Kajak", price: 275 };
console.log(`Typ mapowany: ${p.name}, ${p.price}`);
```

Mapowania wbudowane działają w dokładnie taki sam sposób jak zdefiniowane na listingu 13.19. Kod przedstawiony na listingu 13.19 powoduje wygenerowanie następujących danych wyjściowych:

```
Typ mapowany: Kajak, 275
```

Mapowanie określonych właściwości

Zapytanie typu indeksu dla typu mapowanego może być wyrażone w postaci parametru typu generycznego, który następnie może być stosowany do wybierania określonych właściwości do mapowania według nazw, jak pokazałem w przykładzie na listingu 13.20.

Listing 13.20. Mapowanie określonych właściwości w kodzie pliku *index.ts* w katalogu *src*

```
import { City, Person, Product, Employee } from "../dataTypes";

type SelectProperties<T, K extends keyof T> = {
  [P in K]: T[P]
};

let p1: SelectProperties<Product, "name"> = { name: "Kajak" };
let p2: Pick<Product, "name"> = { name: "Kajak" };
let p3: Omit<Product, "price"> = { name: "Kajak" };
console.log(`Własny typ mapowany: ${p1.name}`);
console.log(`Wbudowany typ mapowany (Pick): ${p2.name}`);
console.log(`Wbudowany typ mapowany (Omit): ${p3.name}`);
```

Mapowanie `SelectProperties` definiuje dodatkowy parametr typu generycznego o nazwie `K`, który jest ograniczony za pomocą `keyof` w taki sposób, że mogą być określane tylko typy odpowiadające właściwościom zdefiniowanym przez parametr typu `T`. Nowy parametr typu zostaje użyty w nazwie selektora mapowania, a poszczególne właściwości będą mogły być uwzględniane w mapowaniu, np.:

```
...
let p1: SelectProperties<Product, "name"> = { name: "Kajak" };
...
```

To mapowanie pobiera właściwość `name` zdefiniowaną w klasie `Product`. Wiele właściwości można wyrazić jako unię typów, a TypeScript dostarcza wbudowane mapowanie `Pick<T, K>` pełniące tę samą funkcję.

```
...
let p2: Pick<Product, "name"> = { name: "Kajak" };
...
```

Mapowanie `Pick` określa klucze, które pozostaną zachowane w typie mapowanym. Natomiast mapowanie `Omit` działa odwrotnie i wyklucza jeden lub więcej kluczy.

```
...
let p3: Omit<Product, "price"> = { name: "Kajak" };
...
```

Te mapowania powodują wygenerowanie takiego samego wyniku jak wcześniejsze przykłady. Dlatego też kod przedstawiony na listingu 13.20 powoduje wygenerowanie następujących danych wyjściowych:

```
Własny typ mapowany: Kajak
Wbudowany typ mapowany (Pick): Kajak
Wbudowany typ mapowany (Omit): Kajak
```

Łączenie transformacji w pojedyncze mapowanie

Na listingu 13.19 pokazałem, jak połączyć mapowania w celu utworzenia łańcucha transformacji. Jednak mapowania mogą wprowadzać wiele zmian we właściwościach, co przedstawiłem na listingu 13.21.

Listing 13.21. Łączenie transformacji w kodzie pliku `index.ts` w katalogu `src`

```
import { City, Person, Product, Employee } from "../dataTypes";

type CustomMapped<T, K extends keyof T> = {
  readonly[P in K]?: T[P]
};

type BuiltInMapped<T, K extends keyof T> = Readonly<Partial<Pick<T, K>>>;

let p1: CustomMapped<Product, "name"> = { name: "Kajak" };
let p2: BuiltInMapped<Product, "name" | "price">
  = { name: "Kamizelka ratunkowa", price: 48.95};
console.log(`Własny typ mapowany: ${p1.name}`);
console.log(`Wbudowany typ mapowany: ${p2.name}, ${p2.price}`);
```

W przypadku niestandardowego mapowania typu znak zapytania i słowo kluczowe `readonly` mogą być zastosowane w tej samej transformacji, którą można ograniczyć w taki sposób, aby zezwalała na pobieranie właściwości według nazw. Mapowania można również łączyć ze sobą, jak pokazałem na przykładzie połączenia mapowań `Pick`, `Partial` i `Readonly`. Kod przedstawiony na listingu 13.21 powoduje wygenerowanie następujących danych wyjściowych:

Własny typ mapowany: Kajak
 Wbudowany typ mapowany: Kamizelka ratunkowa, 48.95

Tworzenie typu z użyciem mapowania

Ostatnia funkcjonalność dostarczana przez mapowanie typu to możliwość tworzenia nowych typów zamiast przekształcania wskazanych. Na listingu 13.22 zaprezentowałem podstawowy sposób na wykorzystanie tej funkcjonalności — utworzenie typu zawierającego właściwości `name` i `city`.

Listing 13.22. Tworzenie typu w kodzie pliku `index.ts` w katalogu `src`

```
import { City, Person, Product, Employee } from "../dataTypes";

type CustomMapped<K extends keyof any, T> = {
  [P in K]: T
};

let p1: CustomMapped<"name" | "city", string> = { name: "Bartek", city: "Londyn"};
let p2: Record<"name" | "city", string> = { name: "Alicja", city: "Paryż"};

console.log(`Własny typ mapowany: ${p1.name}, ${p1.city}`);
console.log(`Wbudowany typ mapowany: ${p2.name}, ${p2.city}`);
```

Pierwszy parametr typu generycznego jest ograniczony za pomocą słów kluczowych `keyof any`, co oznacza, że może być podany literał wartości unii typu i będzie on zawierał nazwy właściwości wymaganych dla nowego typu. Drugi parametr typu generycznego jest wykorzystywany do określenia typu właściwości, które są tworzone i używane w przedstawiony tutaj sposób:

```
...
let p1: CustomMapped<"name" | "city", string> = { name: "Bartek", city: "Londyn"};
...
```

Mapowanie powoduje wygenerowanie dwóch właściwości w postaci ciągu tekstowego: `name` i `city`. TypeScript zawiera wbudowane mapowanie `Record` wykonujące to samo zadanie.

```
...
let p2: Record<"name" | "city", string> = { name: "Alicja", city: "Paryż"};
...
```

To jest funkcjonalność mapowania, z której korzystam w moich projektach. Pokazuje ona, że mapowania są znacznie elastyczniejsze, niż mogłoby się wydawać, a literały wartości typów ograniczane przez `keyof any` mogą akceptować dowolne połączenia nazw właściwości. Kod przedstawiony na listingu 13.22 powoduje wygenerowanie następujących danych wyjściowych:

Własny typ mapowany: Bartek, Londyn
 Wbudowany typ mapowany: Alicja, Paryż

Używanie typów warunkowych

Typy warunkowe to wyrażenia zawierające parametry typu generycznego, które są wykorzystywane w celu pobierania nowych typów. Na listingu 13.23 widać prosty przykład zastosowania typu warunkowego.

Listing 13.23. Używanie typu warunkowego w kodzie pliku *index.ts* w katalogu *src*

```
import { City, Person, Product, Employee } from "../dataTypes";
```

```
type resultType<T extends boolean> = T extends true ? string : number;
```

```
let firstVal: resultType<true> = "Wartość w postaci ciągu tekstowego";
```

```
let secondVal: resultType<false> = 100;
```

```
let mismatchCheck: resultType<false> = "Wartość w postaci ciągu tekstowego";
```

Typ warunkowy ma parametr typu generycznego i wyrażenie trójargumentowe pobierające typ wyniku, jak pokazałem na rysunku 13.4.

```
type resultType< T extends boolean> = T extends true ? string : number ;
```

Rysunek 13.4. Typ warunkowy

Typ warunkowy to miejsce zarezerwowane dla jednego z typów wyniku, które nie będą wybrane aż do chwili użycia parametru typu generycznego, co z kolei pozwoli na obliczenie wartości wyrażenia z wykorzystaniem jednego z wybranych typów wyniku.

W omawianym przykładzie typ warunkowy `resultType<T>` jest miejscem zarezerwowanym dla typów `string` i `number`. Dlatego też argument dla typu generycznego `T` będzie określał, czy typ warunkowy zostanie ustalony jako `string`, czy jako `number`. Parametr typu generycznego, `T`, został ograniczony i może akceptować jedynie wartości typu `boolean`, a wyrażenie przyjmie wartość `true`, jeśli argument dostarczony dla `T` będzie literałem wartości typu `true`. Dlatego też jeśli `T` ma wartość `true`, wówczas wartością `resultType<T>` będzie `string`.

```
...
let firstVal: resultType<true> = "Wartość w postaci ciągu tekstowego";
let stringTypeCheck: string = firstVal;
...
```

Kompilator określa typ warunkowy i wie, że adnotacja typu dla `firstVal` będzie ustalona jako `string`, co pozwala na przypisanie `firstVal` literału wartości w postaci ciągu tekstowego. Gdy argumentem typu generycznego jest `false`, wówczas typem warunkowym będzie `number`.

```
...
let secondVal: resultType<false> = 100;
let numberTypeCheck: number = secondVal;
...
```

Kompilator gwarantuje bezpieczeństwo typu podczas pracy z typami warunkowymi. W ostatnim poleceniu na listingu 13.23 typ warunkowy będzie określony jako `number`, choć jest

mu przypisywana wartość w postaci ciągu tekstowego, co prowadzi do wygenerowania przez kompilator następującego komunikatu błędu:

```
error TS2322: Type '"Wartość w postaci ciągu tekstowego"' is not assignable to type 'number'.
```

Niebezpieczeństwa związane z używaniem typów warunkowych

Typ warunkowy to funkcja zaawansowana, z której należy korzystać z zachowaniem ostrożności. Przygotowanie typu warunkowego może być trudnym procesem i często przypomina sztuczki, ponieważ zmuszasz kompilator do przejścia przez serię wyrażeń, aby wreszcie otrzymać oczekiwany wynik.

Wraz ze wzrostem stopnia skomplikowania typu warunkowego rośnie również niebezpieczeństwo pominięcia niektórych permutacji typu, a tym samym utworzenia wyniku zbyt luźnego, powstania luki w operacji sprawdzania typu bądź też utworzenia wyniku zbyt restrykcyjnego i powodowania błędów kompilatora dla poprawnych wartości.

Podczas używania typów warunkowych pamiętaj, że jedynie opisujesz połączenia typów dla kompilatora TypeScriptu, a informacje o typie zostaną usunięte w trakcie kompilacji. Gdy typ warunkowy stanie się bardziej skomplikowany i będzie obejmował więcej wariantów, wówczas powinieneś zatrzymać się na chwilę i zastanowić, czy istnieje prostszy sposób na osiągnięcie tego samego wyniku.

Zagnieżdżanie typów warunkowych

Bardziej zaawansowane warianty typów mogą być opisywane przez zagnieżdżanie typów warunkowych. Wynikiem typu zagnieżdżonego może być inny typ warunkowy, a kompilator będzie podążał za łańcuchem wyrażeń aż do momentu dotarcia do wyniku niebędącego typem warunkowym, jak pokazałem na listingu 13.24.

Listing 13.24. Zagnieżdżanie typów warunkowych w kodzie pliku *index.ts* w katalogu *src*

```
import { City, Person, Product, Employee } from "../dataTypes";

type resultType<T extends boolean> = T extends true ? string : number;

type references = "Londyn" | "Bartek" | "Kajak";

type nestedType<T extends references>
  = T extends "Londyn" ? City : T extends "Bartek" ? Person : Product;

let firstVal: nestedType<"Londyn"> = new City("Londyn", 8136000);
let secondVal: nestedType<"Bartek"> = new Person("Bartek", "Londyn");
let thirdVal: nestedType<"Kajak"> = new Product("Kajak", 275);
```

Typ `nestedType<T>` jest zagnieżdżonym typem warunkowym przeznaczonym do wyboru jednego spośród trzech typów na podstawie parametru typu generycznego. Jak wspomniałem w ramce we wcześniejszej części rozdziału, opanowanie pracy z typem warunkowym może być trudnym zadaniem, zwłaszcza gdy te typy są zagnieżdżone.

Używanie typu warunkowego w klasie generycznej

Typ warunkowy może być używany do wyrażania relacji między typami parametrów metody lub funkcji oraz generowanymi wynikami, jak pokazałem na listingu 13.25. To zwięźlejsza alternatywa dla omówionego w rozdziale 8. przeciążania typu funkcji, choć typ warunkowy jest koncepcją trudniejszą do poznania i zrozumienia.

Listing 13.25. Definiowanie typu generycznego w kodzie pliku `index.ts` w katalogu `src`

```
import { City, Person, Product, Employee } from "../dataTypes";

type resultType<T extends boolean> = T extends true ? string : number;

class Collection<T> {
  private items: T[];

  constructor(...initialItems: T[]) {
    this.items = initialItems || [];
  }

  total<P extends keyof T, U extends boolean>(propName: P, format: U)
    : resultType<U> {
    let totalValue = this.items.reduce((t, item) =>
      t += Number(item[propName]), 0);
    return format ? `${totalValue.toFixed()}` : totalValue as any;
  }
}

let data = new Collection<Product>(new Product("Kajak", 275), new Product("Kamizelka
ratunkowa", 48.95));

let firstVal: string = data.total("price", true);
console.log(`Wartość sformatowana: ${firstVal}`);
let secondVal: number = data.total("price", false);
console.log(`Wartość niesformatowana: ${secondVal}`);
```

Klasa `Collection<T>` używa tablicy do przechowywania obiektów, których typ jest określony za pomocą parametru typu generycznego o nazwie `T`. Metoda `total()` definiuje dwa parametry typu generycznego. Pierwszy, `P`, określa właściwość używaną do utworzenia wartości całkowitej. Drugi, `U`, określa to, czy wynik powinien zostać sformatowany. Wynik działania metody `total()` jest typu warunkowego i konkretny typ zostanie ustalony na podstawie wartości dostarczonej dla parametru typu `U`.

```
...
total<P extends keyof T, U extends boolean>(propName: P, format: U): resultType<U> {
...

```

Użycie typu warunkowego oznacza, że wynik działania metody `total()` jest określany przez argument dostarczony dla parametru typu `U`. Skoro kompilator potrafi ustalić typ `U` na podstawie wartości dostarczonej dla argumentu `format`, jak to wyjaśniłem w rozdziale 12., tę metodę można wywołać w następujący sposób:

```
...
let firstVal: string = data.total("price", true);
...

```

Gdy argument dla parametru `format` ma wartość `true`, typ warunkowy określi `string` jako typ wyniku metody `total()`. Spowoduje to dopasowanie typu danych wskazanego przez implementację metody.

```
...
return format ? `${totalValue.toFixed()} zł` : totalValue as any;
...
```

Gdy argument dla parametru `format` ma wartość `false`, typ warunkowy określi `number` jako typ wyniku metody `total()`. Spowoduje to, że metoda zwróci niesformatowaną wartość.

```
...
return format ? `${totalValue.toFixed()} zł` : totalValue as any;
...
```

Zwrot wartości w metodzie używającej typu warunkowego

W chwili powstawania książki kompilator TypeScriptu miał trudność w korelacji typu danych wartości zwracanych przez metody i funkcje, gdy używane są typy warunkowe. Dlatego też w kodzie na listingu 13.25 została użyta asercja typu w metodzie `total()` mająca na celu wskazanie kompilatorowi, aby wynik był traktowany jako typu `any`. Bez takiej adnotacji typu kompilator spowoduje wygenerowanie błędu.

Kod przedstawiony na listingu 13.25 powoduje wygenerowanie następujących danych wyjściowych:

```
Wartość sformatowana: 324 zł
Wartość niesformatowana: 323.95
```

Używanie typów warunkowych z uniami typów

Typy warunkowe mogą być używane do filtrowania unii typów, pozwalając na łatwe wybieranie lub wykluczanie typów ze zbioru tworzącego unię, jak pokazałem w przykładzie na listingu 13.26.

Listing 13.26. Filtrowanie unii typu w kodzie pliku `index.ts` w katalogu `src`

```
import { City, Person, Product, Employee } from "../dataTypes";

type Filter<T, U> = T extends U ? never : T;

function FilterArray<T, U>(data: T[],
    predicate: (item) => item is U): Filter<T, U>[] {
    return data.filter(item => !predicate(item)) as any;
}

let dataArray = [new Product("Kajak", 275), new Person("Bartek", "Londyn"),
    new Product("Kamizelka ratunkowa", 27.50)];

function isProduct(item: any): item is Product {
    return item instanceof Product;
}

let filteredData: Person[] = FilterArray(dataArray, isProduct);
filteredData.forEach(item => console.log(`Osoba: ${item.name}`));
```

Gdy typ warunkowy jest dostarczany z unią typów, kompilator TypeScriptu przekazuje warunek do poszczególnych typów w unii i tworzy tzw. *dystrybucyjny typ warunkowy*. Ten efekt jest stosowany w przypadku używania typu warunkowego jako unii typów, co widać w następującym przykładzie:

```
...
type filteredUnion = Filter<Product | Person, Product>
...
```

Kompilator TypeScriptu stosuje typ warunkowy oddzielnie dla poszczególnych typów w unii, a następnie tworzy unię wyników, np.:

```
...
type filteredUnion = Filter<Product, Product> | Filter<Person, Product>
...
```

Typ warunkowy `Filter<T, U>` przyjmuje wartość `never`, gdy pierwszy parametr typu jest taki sam jak drugi, co prowadzi do wygenerowania następującego wyniku:

```
...
type filteredUnion = never | Person
...
```

Nie ma możliwości zdefiniowania unii z `never`, więc kompilator pomija ten typ w unii, a wynik w postaci `Filter<Product | Person, Product>` jest odpowiednikiem następującego typu:

```
...
type filteredUnion = Person
...
```

Typ warunkowy odrzuca wszelkie typy niemożliwe do przypisania `Person` i zwraca pozostałe typy w unii. Metoda `FilterArray<T, U>` zajmuje się filtrowaniem tablicy za pomocą funkcji predykatu i zwraca typ `Filter<T, U>`. Kod przedstawiony na listingu 13.26 powoduje wygenerowanie następujących danych wyjściowych:

Osoba: Bartek

Używanie wbudowanych dystrybucyjnych typów warunkowych

TypeScript oferuje zestaw wbudowanych typów warunkowych, które są stosowane do filtrowania unii (tabela 13.7). Pozwala to na wykonywanie wielu zadań bez konieczności definiowania typów niestandardowych.

Tabela 13.7. Wbudowane dystrybucyjne typy warunkowe

Nazwa typu warunkowego	Opis
<code>Exclude<T, U></code>	Ten typ wyklucza typy, które mogą być przypisane do <code>U</code> z poziomu <code>T</code> . Jest to odpowiednik typu <code>Filter<T, U></code> przedstawionego na listingu 13.26
<code>Extract<T, U></code>	Ten typ pobiera typy, które mogą być przypisane do <code>U</code> z poziomu <code>T</code>
<code>NonNullable<T></code>	Ten typ wyklucza <code>null</code> i <code>undefined</code> z <code>T</code>

Używanie typów warunkowych podczas mapowania typów

Typy warunkowe mogą być łączone z mapowaniem typów, co pozwala na stosowanie różnych transformacji na właściwościach w typie. W efekcie otrzymujesz znacznie większą elastyczność niż w przypadku stosowania samej funkcjonalności. Na listingu 13.27 możesz się zapoznać z przykładem mapowania typu używającego typu warunkowego.

Listing 13.27. Definiowanie mapowania typu w typie warunkowym w kodzie pliku *index.ts* w katalogu *src*

```
import { City, Person, Product, Employee } from "../dataTypes";

type changeProps<T, U, V> = {
  [P in keyof T]: T[P] extends U ? V: T[P]
};

type modifiedProduct = changeProps<Product, number, string>;

function convertProduct(p: Product): modifiedProduct {
  return { name: p.name, price: `${p.price.toFixed(2)} ` };
}

let kayak = convertProduct(new Product("Kajak", 275));
console.log(`Produkt: ${kayak.name}, ${kayak.price}`);
```

Mapowanie `changeProps<T, U, V>` pobiera właściwość typu `U` i zmienia ją na typ `V` w typie mapowanym. To polecenie ma zastosowanie do mapowania klasy `Product` i określa, że właściwość `number` powinna być we właściwości `string`:

```
...
type modifiedProduct = changeProp<Product, number, string>;
...
```

Typ mapowany definiuje właściwości `name` i `price` będące typu `string`. Typ `modifiedResult` jest używany jako wynik działania funkcji `convertProduct()` akceptującej obiekt `Product` i zwracającej obiekt zgodny z kształtem typu mapowanego przez sformatowanie właściwości `price`. Kod przedstawiony na listingu 13.27 powoduje wygenerowanie następujących danych wyjściowych:

```
Produkt: Kajak, 275.00
```

Identyfikowanie właściwości określonego typu

Bardzo często zachodzi potrzeba ograniczenia parametru typu, aby mógł być używany tylko do określenia właściwości będącej konkretnego typu. Na przykład klasa `Collection<T>` w kodzie przedstawionym na listingu 13.25 definiuje metodę `total()` akceptującą nazwę właściwości i powinna być ograniczona do właściwości typu `number`. Takie ograniczenie można nałożyć przez połączenie funkcjonalności przedstawionych we wcześniejszej części rozdziału, jak pokazałem na listingu 13.28.

Listing 13.28. Identyfikowanie właściwości w kodzie pliku *index.ts* w katalogu *src*

```
import { City, Person, Product, Employee } from "./dataTypes";

type unionOfTypeNames<T, U> = {
  [P in keyof T] : T[P] extends U ? P : never;
};

type propertiesOfType<T, U> = unionOfTypeNames<T, U>[keyof T];

function total<T, P extends propertiesOfType<T, number>>(data: T[],
  propName: P): number {
  return data.reduce((t, item) => t += Number(item[propName]), 0);
}

let products = [new Product("Kajak", 275), new Product("Kamizelka ratunkowa", 48.95)];
console.log(`Wartość całkowita: ${total(products, "price")}`);
```

Metoda identyfikowania właściwości jest nietypowa, więc ten proces podzieliłem na dwa polecenia, aby można go było łatwiej zinterpretować. Pierwszym krokiem jest użycie mapowania typu zawierającego polecenie warunkowe.

```
...
type unionOfTypeNames<T, U> = {
  [P in keyof T] : T[P] extends U ? P : never;
};
...
```

Polecenie warunkowe sprawdza typ poszczególnych właściwości. Jeżeli właściwość nie ma typu docelowego, wówczas ten typ zostanie zmieniony na `never`. Jeżeli właściwość ma oczekiwany typ, wtedy jej typ będzie zmieniony na literał wartości będący nazwą właściwości. Dlatego też mapowanie `unionOfTypeNames<Product, number>` powoduje wygenerowanie następującego typu mapowanego:

```
...
{
  name: never,
  price: "price"
}
...
```

Ten dziwny typ mapowany dostarcza dane wejściowe do drugiego etapu procesu, który polega na wykorzystaniu zindeksowanego operatora dostępu w celu pobrania unii typów właściwości zdefiniowanych w typie mapowania, np.:

```
...
type propertiesOfType<T, U> = unionOfTypeNames<T, U>[keyof T];
...
```

Dla typu mapowanego utworzonego przez `unionOfTypeNames<Product, number>` zindeksowany operator dostępu powoduje wygenerowanie następującej unii:

```
...
never | "price"
...
```

Jak wcześniej wspomniałem, typ `never` jest automatycznie usuwany z unii i pozostawia unię literalów wartości typów będących wymaganymi typami właściwości. Nazwy właściwości unii mogą zostać użyte do ograniczenia parametrów typu generycznego.

```
...
function total<T, P extends propertiesOfT<T, number>>(data: T[],
  propName: P): number {
  return data.reduce((t, item) => t += Number(item[propName]), 0);
}
...
```

Parametr `propName` funkcji `total()` można wykorzystać jedynie z nazwami właściwości `number` w typie `T`, jak pokazałem w kolejnym fragmencie kodu:

```
...
console.log(`Wartość całkowita: ${total(products, "price")}`);
...
```

Ten przykład pokazuje, jak elastyczna jest funkcjonalność typu generycznego w TypeScriptie, a także ilustruje, jak nietypowe kroki mogą być wymagane w celu osiągnięcia zamierzonego efektu. Kod przedstawiony na listingu 13.28 powoduje wygenerowanie następujących danych wyjściowych:

```
Wartość całkowita: 323.95
```

Automatyczne ustalanie typów dodatkowych w warunkach

Mogą występować tarcia między potrzebą akceptacji szerokiej gamy typów za pomocą parametru typu generycznego a potrzebą poznania szczegółów dotyczących tych typów. W kodzie na listingu 13.29 została przedstawiona funkcja akceptująca tablicę lub pojedynczy obiekt danego typu.

Listing 13.29. Definiowanie funkcji w kodzie pliku `index.ts` w katalogu `src`

```
import { City, Person, Product, Employee } from "../dataTypes";

function getValue<T, P extends keyof T>(data: T, propName: P): T[P] {
  if (Array.isArray(data)) {
    return data[0][propName];
  } else {
    return data[propName];
  }
}

let products = [new Product("Kajak", 275), new Product("Kamizelka ratunkowa", 48.95)];
console.log(`Wartość w postaci tablicy: ${getValue(products, "price")}`);
console.log(`Pojedyncza wartość całkowita: ${getValue(products[0], "price")}`);
```

Ten kod nie zostanie skompilowany, ponieważ parametry generyczne nie przechwytyują prawidłowo relacji zachodzących między typami. Jeżeli funkcja `total()` otrzymuje tablicę za pomocą parametru `data`, zwraca wartość właściwości określonej przez parametr `propName`

pierwszego elementu tablicy. Jeżeli funkcja otrzymuje pojedynczy obiekt za pomocą parametru `data`, wtedy zwraca wartość `propName` dla tego obiektu. Parametr `propName` został ograniczony za pomocą `keyof`, co stanowi problem podczas używania tablicy, ponieważ `keyof` zwraca unię nazw właściwości zdefiniowanych przez obiekt tablicy JavaScriptu, a nie właściwości typu znajdującego się w tablicy, o czym możesz się przekonać z komunikatu błędu wygenerowanego przez kompilator.

```
src/index.ts(12,48): error TS2345: Argument of type '"price"' is not assignable to parameter of type 'number | keyof Product[]'.
```

Słowo kluczowe `infer` w TypeScriptie może być stosowane do ustalania typów, które nie zostały jawnie wymienione w parametrach typu warunkowego. Na przykład można nakazać kompilatorowi ustalenie typu obiektów znajdujących się w tablicy, jak pokazałem na listingu 13.30.

Listing 13.30. Ustalenie typu tablicy w kodzie pliku `index.ts` w katalogu `src`

```
import { City, Person, Product, Employee } from "../dataTypes";

type targetKeys<T> = T extends (infer U)[] ? keyof U: keyof T;

function getValue<T, P extends targetKeys<T>>(data: T, propName: P): T[P] {
    if (Array.isArray(data)) {
        return data[0][propName];
    } else {
        return data[propName];
    }
}

let products = [new Product("Kajak", 275), new Product("Kamizelka ratunkowa", 48.95)];
console.log(`Wartość w postaci tablicy: ${getValue(products, "price")}`);
console.log(`Pojedyncza wartość całkowita: ${getValue(products[0], "price")}`);
```

Typ jest ustalany za pomocą słowa kluczowego `infer` i wprowadza typ generyczny, którego typ zostanie określony przez kompilator w chwili rozwiązania typu warunkowego, jak pokazałem na rysunku 13.5.

```
type targetKeys<T> = T extends (infer U)[] ? keyof U: keyof T;
```

Rysunek 13.5. Automatyczne ustalanie typu w typie warunkowym

W kodzie przedstawionym na listingu 13.30 typ `U` zostaje określony, jeśli `T` jest tablicą. Typ `U` będzie ustalony przez kompilator na podstawie parametru typu generycznego `T` podczas jego rozwiązywania. Efekt jest taki, że typy `targetKeys<Product>` i `targetKeys<Product[]>` prowadzą do powstania unii `"name" | "price"`. Typ warunkowy można wykorzystać do ograniczenia

właściwości funkcji `getValue<T, P>()`, zapewniając tym samym spójne typy zarówno dla pojedynczych obiektów, jak i dla tablic. Kod przedstawiony na listingu 13.30 powoduje wygenerowanie następujących danych wyjściowych:

Wartość w postaci tablicy: 275
 Pojedyncza wartość całkowita: 275

Ustalanie typu funkcji

Kompilator może również ustalić typ w przypadku typu generycznego akceptującego funkcję, jak pokazałem na listingu 13.31.

Listing 13.31. Używanie inferencji typu dla funkcji w kodzie pliku `index.ts` w katalogu `src`

```
import { City, Person, Product, Employee } from "../dataTypes";

type Result<T> = T extends (...args: any) => infer R ? R : never;

function processArray<T,
  Func extends (T) => any>(data: T[], func: Func): Result<Func>[] {
  return data.map(item => func(item));
}

let selectName = (p: Product) => p.name;

let products = [new Product("Kajak", 275), new Product("Kamizelka ratunkowa", 48.95)];
let names: string[] = processArray(products, selectName);
names.forEach(name => console.log(`Nazwa: ${name}`));
```

Typ warunkowy `Result<T>` używa słowa kluczowego `infer` do pobrania typu wyniku z funkcji akceptującej obiekt typu `T` i generującej wynik `any`. Zastosowanie inferencji typu pozwala na używanie funkcji przetwarzającej określony typ i jednocześnie gwarantuje, że wynikiem funkcji `processArray()` jest typ określony na podstawie wyniku działania funkcji dostarczonej dla parametru `func`. Funkcja `selectName()` zwraca wartość `string` właściwości `name` obiektu `Product`, a inferencja oznacza, że `Result<(... args:Product) => string>` zostaje prawidłowo zidentyfikowany jako `string`, pozwalając funkcji `processArray()` na zwrot wyniku w postaci `string[]`. Kod przedstawiony na listingu 13.31 powoduje wygenerowanie następujących danych wyjściowych:

Nazwa: Kajak
 Nazwa: Kamizelka ratunkowa

Proces inferencji w typie warunkowym może być trudny do zrozumienia, a TypeScript dostarcza serię wbudowanych typów warunkowych, które są użyteczne podczas pracy z funkcjami. Oferowane przez TypeScript typy wymieniałem w tabeli 13.8.

Typy warunkowe `ConstructorParameters<T>` i `InstanceType<T>` operują na funkcjach konstruktora i są najbardziej użyteczne podczas opisywania typów funkcji tworzących obiekty, których typ będzie podany jako parametr typu generycznego, jak pokazałem na listingu 13.32.

Tabela 13.8. Wbudowane w TypeScript typy warunkowe stosujące inferencję typu

Nazwa typu warunkowego	Opis
Parameters<T>	Typ warunkowy pobierający typy poszczególnych parametrów funkcji wyrażone jako krotka
ReturnType<T>	Typ warunkowy pobierający typ wyniku działania funkcji, odpowiednik Result<T> na listingu 13.31
ConstructorParameters<T>	Typ warunkowy pobierający typy poszczególnych parametrów funkcji konstruktora wyrażone jako krotka, co pokażę w dalszej części rozdziału
InstanceType<T>	Typ warunkowy zwracający typ wyniku działania funkcji konstruktora

Listing 13.32. Używanie wbudowanych typów warunkowych w kodzie pliku index.ts w katalogu src

```
import { City, Person, Product, Employee } from "../dataTypes";

function makeObject<T extends new (...args: any) => any>
    (constructor: T, ...args: ConstructorParameters<T>) : InstanceType<T> {
    return new constructor(...args as any[]);
}

let prod: Product = makeObject(Product, "Kajak", 275);
let city: City = makeObject(City, "Londyn", 8136000);

[prod, city].forEach(item => console.log(`Nazwa: ${item.name}`));
```

Funkcja makeObject() tworzy obiekt na podstawie klasy, nie mając wcześniej żadnych informacji o tym, która klasa jest wymagana. Typy warunkowe ConstructorParameters<T> i InstanceType<T> przeprowadzają inferencję parametrów i konstruktora klasy dostarczonej jako pierwszy parametr typu generycznego. Gwarantuje to, że funkcja makeObject() otrzymuje odpowiedni typ do utworzenia obiektu, dokładnie odzwierciedlający typ tworzonego obiektu. Kod przedstawiony na listingu 13.32 powoduje wygenerowanie następujących danych wyjściowych:

```
Nazwa: Kajak
Nazwa: Londyn
```

Podsumowanie

W tym rozdziale przedstawiłem oferowaną przez TypeScript zaawansowaną funkcjonalność związaną z typami generycznymi. Stosowanie tej funkcjonalności nie jest wymagane w każdym obiekcie, choć będzie ona nieoceniona, gdy te najbardziej podstawowe funkcje nie są w stanie opisać typów wymaganych przez aplikację. W następnym rozdziale wyjaśnię, jak TypeScript radzi sobie z kodem JavaScriptu, gdy stanowi on część projektu, a także gdy znajduje się w pakietach firm trzecich wymaganych do prawidłowego działania aplikacji.

ROZDZIAŁ 14.



Praca z JavaScriptem

Ogólnie rzecz biorąc, projekt TypeScriptu zawiera pewną ilość czystego kodu JavaScriptu, ponieważ aplikacja jest utworzona w językach TypeScript i JavaScript lub dlatego, że projekt wykorzystuje pakiety JavaScriptu firm trzecich zainstalowane za pomocą menedżera pakietów Node.js. W tym rozdziale zamierzam przedstawić oferowane przez TypeScript funkcje przeznaczone do pracy z kodem JavaScriptu. Streszczenie materiału przedstawionego w rozdziale znajdziesz w tabeli 14.1.

Tabela 14.1. Streszczenie materiału przedstawionego w rozdziale

Problem	Rozwiązanie	Listing
Dodanie plików JavaScriptu do projektu	Włącz opcje kompilatora <code>allowJs</code> i <code>checkJs</code>	Od 9 do 13
Określenie, czy plik JavaScriptu jest sprawdzany przez kompilator TypeScriptu	Użyj poleceń <code>@ts-check</code> i <code>@ts-nocheck</code>	14
Opisywanie typów JavaScriptu	Użyj poleceń <code>JSDoc</code> lub utwórz plik deklaracji	Od 15 do 22
Opisywanie kodu JavaScriptu utworzonego przez inną firmę	Zaktualizuj konfigurację kompilatora i utwórz plik deklaracji	Od 22 do 26
Opisywanie kodu innej firmy bez tworzenia pliku deklaracji	Użyj pakietu zawierającego plik deklaracji lub zainstaluj publicznie dostępny pakiet deklaracji	Od 27 do 35
Generowanie pliku deklaracji przeznaczonego do używania w innych projektach	Włącz opcję kompilatora <code>declaration</code>	Od 36 do 39

W tabeli 14.2 wymieniałem opcje kompilatora TypeScriptu użyte w rozdziale.

Tabela 14.2. Opcje kompilatora TypeScriptu użyte w rozdziale

Opcja	Opis
<code>allowJs</code>	Ta opcja powoduje dodanie wszystkich plików JavaScriptu w procesie kompilacji
<code>baseUrl</code>	Ta opcja określa katalog główny używany podczas rozwiązywania zależności modułów
<code>checkJs</code>	Ta opcja nakazuje kompilatorowi sprawdzenie kodu JavaScriptu pod kątem najczęściej popełnianych błędów
<code>declaration</code>	Ta opcja generuje pliki deklaracji typu, które opisują typy przeznaczone do używania w innych projektach
<code>esModuleInterop</code>	Ta opcja dodaje kod pomocniczy umożliwiający importowanie funkcjonalności z modułów nie definiujących domyślnie eksportowanych funkcji i jest używana z opcją <code>allowSyntheticDefaultImports</code>
<code>outDir</code>	Ta opcja określa katalog, w którym zostaną umieszczone wygenerowane pliki JavaScriptu
<code>paths</code>	Ta opcja określa katalogi używane podczas rozwiązywania zależności modułu
<code>rootDir</code>	Ta opcja określa katalog główny używany przez kompilator do wyszukiwania plików TypeScriptu
<code>target</code>	Ta opcja określa docelową wersję języka JavaScript, dla której kompilator wygeneruje kod

Przygotowanie projektu

Aby przygotować się do tego rozdziału, w dogodnym miejscu utwórz katalog o nazwie *usingjs*. Następnie przejdź do powłoki, wejdź do katalogu *usingjs* i wydaj w nim polecenia przedstawione na listingu 14.1. Drugie polecenie inicjalizuje katalog do użycia i nakazuje menedżerowi pakietów Node.js utworzenie pliku o nazwie *package.json*, w którym będą wymienione pliki dodane do projektu.

■ **Wskazówka** Przykładowy projekt dla tego rozdziału, a także projekty dla pozostałych rozdziałów książki zostały umieszczone w archiwum dostępnym pod adresem <https://ftp.helion.pl/przyklady/tys4o2.zip>.

Listing 14.1. Tworzenie pliku *package.json* w przykładowym projekcie

```
$ cd usingjs
$ npm init --yes
```

Z poziomu katalogu *usingjs* wydaj polecenia przedstawione na listingu 14.2 i tym samym dodaj pakiety wymagane w projekcie dla tego rozdziału.

Listing 14.2. Dodawanie pakietów do projektu

```
$ npm install --save-dev typescript@4.2.2
$ npm install --save-dev tsc-watch@4.2.9
```

Aby skonfigurować kompilator TypeScriptu, dodaj do katalogu *usingjs* plik o nazwie *tsconfig.json* z zawartością przedstawioną na listingu 14.3.

Listing 14.3. Zawartość pliku *tsconfig.json* w katalogu *usingjs*

```
{
  "compilerOptions": {
    "target": "es2020",
    "outDir": "./dist",
    "rootDir": "./src",
    "module": "commonjs"
  }
}
```

Te ustawienia konfiguracyjne nakazują kompilatorowi TypeScriptu wygenerowanie kodu przeznaczonego dla najnowszych implementacji JavaScriptu, w katalogu *src* kompilator będzie szukał plików TypeScriptu, a w katalogu *dist* umieści pliki z wygenerowanym kodem. Ustawienie *module* wskazuje kompilatorowi, że moduły CommonJS są wymagane — jest to format obsługiwany przez Node.js.

Jeżeli chcesz przygotować konfigurację pozwalającą na łatwe uruchamianie kompilatora, dodaj do pliku *package.json* opcję konfiguracyjną przedstawioną na listingu 14.4.

Listing 14.4. Konfigurowanie menedżera pakietów Node.js w pliku *package.json* w katalogu *usingjs*

```
{
  "name": "usingjs",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "tsc-watch --onsuccess \"node dist/index.js\""
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "tsc-watch": "^4.2.9",
    "typescript": "^4.2.2"
  }
}
```

Dodawanie kodu TypeScriptu do przykładowego projektu

W katalogu projektu *usingjs* twórz podkatalog *src* i umieść w nim plik o nazwie *product.ts* z kodem przedstawionym na listingu 14.5.

Listing 14.5. Zawartość pliku *product.ts* w katalogu *src*

```
export class Product {
    constructor(public id: number,
                public name: string,
                public price: number) {
        // Polecenia nie są wymagane.
    }
}

export enum SPORT {
    Running, Soccer, Watersports, Other
}

export class SportsProduct extends Product {
    private _sports: SPORT[];

    constructor(public id: number,
                public name: string,
                public price: number,
                ...sportArray: SPORT[]) {
        super(id, name, price);
        this._sports = sportArray;
    }

    usedForSport(s: SPORT): boolean {
        return this._sports.includes(s);
    }

    get sports(): SPORT[] {
        return this._sports;
    }
}
```

Ten plik został użyty do zdefiniowania prostej klasy *Product* rozszerzonej przez *SportsProduct*, która dodaje funkcjonalność charakterystyczną dla sprzętu sportowego. Następnie w katalogu *src* umieść plik o nazwie *cart.ts* z kodem przedstawionym na listingu 14.6.

Listing 14.6. Zawartość pliku *cart.ts* w katalogu *src*

```
import { SportsProduct } from "../product";

class CartItem {
    constructor(public product: SportsProduct,
                public quantity: number) {
        // Polecenia nie są wymagane.
    }

    get totalPrice(): number {
        return this.quantity * this.product.price;
    }
}

export class Cart {
```

```

private items = new Map<number, CartItem>();

constructor(public customerName: string) {
    // Polecenia nie są wymagane.
}

addProduct(product: SportsProduct, quantity: number): number {
    if (this.items.has(product.id)) {
        let item = this.items.get(product.id);
        item.quantity += quantity;
        return item.quantity;
    } else {
        this.items.set(product.id, new CartItem(product, quantity));
        return quantity;
    }
}

get totalPrice(): number {
    return [...this.items.values()].reduce((total, item) =>
        total += item.totalPrice, 0);
}

get itemCount(): number {
    return [...this.items.values()].reduce((total, item) =>
        total += item.quantity, 0);
}
}

```

Ten plik definiuje klasę `Cart` monitorującą za pomocą egzemplarza `Map` obiekty `SportsProduct` wybierane przez użytkownika. Aby zdefiniować punkt wejścia do projektu, w katalogu `usingjs/src` umieść plik o nazwie `index.ts` z zawartością przedstawioną na listingu 14.7.

Listing 14.7. Zawartość pliku `index.ts` w katalogu `src`

```

import { SportsProduct, SPORT } from "../product";
import { Cart } from "../cart";

let kayak = new SportsProduct(1, "Kajak", 275, SPORT.Watersports);
let hat = new SportsProduct(2, "Czapka", 22.10, SPORT.Running, SPORT.Watersports);
let ball = new SportsProduct(3, "Piłka", 19.50, SPORT.Soccer);

let cart = new Cart("Bartek");
cart.addProduct(kayak, 1);
cart.addProduct(hat, 1);
cart.addProduct(hat, 2);

console.log(`Liczba produktów w koszyku: ${cart.itemCount}`);
console.log(`Wartość produktów w koszyku: ${cart.totalPrice.toFixed(2)} zł`);

```

Kod w pliku `index.ts` tworzy kilka obiektów typu `SportsProduct`, używa ich do wypełnienia koszyka na zakupy (egzemplarz `Cart`), a następnie wyświetla w konsoli szczegóły związane z koszykiem na zakupy.

Otwórz nowe okno powłoki, przejdź do katalogu *usingjs*, a następnie z jego poziomu wydaj polecenie przedstawione na listingu 14.8, uruchamiające kompilator TypeScriptu w trybie automatycznego kompilowania kodu po wykryciu zmiany w dowolnym pliku i wykonywania kodu po jego skompilowaniu.

Listing 14.8. *Uruchamianie kompilatora TypeScriptu*

```
$ npm start
```

Kompilator przeprowadzi kompilację kodu w pliku *index.ts*, wykona wygenerowany kod JavaScriptu, a następnie przejdzie do trybu monitorowania i wyświetli następujące dane wyjściowe:

```
7:23:34 AM - Starting compilation in watch mode...
7:23:36 AM - Found 0 errors. Watching for file changes.
Liczba produktów w koszyku: 4
Wartość produktów w koszyku: 341.30 zł
```

Praca z JavaScriptem

W zaprezentowanych dotychczas przykładach przyjąłem założenie, że praca odbywa się jedynie z kodem TypeScriptu. Bardzo często jest to niemożliwe, ponieważ TypeScript został wprowadzony już w trakcie pracy nad projektem lub zachodzi konieczność pracy z kodem JavaScriptu opracowanym we wcześniej zrealizowanych projektach.

Projekt może zawierać kod TypeScriptu i JavaScriptu, co wymaga jedynie drobnych zmian w konfiguracji kompilatora oraz kilku kroków opcjonalnych opisujących typy używane przez kod JavaScriptu. Aby zademonstrować ten proces, będzie potrzebny pewien kod JavaScriptu. W katalogu *src* umieść plik o nazwie *formatters.js* z kodem przedstawionym na listingu 14.9.

-
- **Uwaga** Rozszerzenie pliku przedstawionego na listingu 14.9 to *.js*, ponieważ zawiera on czysty kod JavaScriptu. W przykładach zaprezentowanych w tym podrozdziale bardzo ważne znaczenie ma używanie odpowiednich rozszerzeń.
-

Listing 14.9. *Zawartość pliku *formatters.js* w katalogu *src**

```
export function sizeFormatter(thing, count) {
    writeMessage(`Liczba elementów egzemplarza ${thing} wynosi: ${count}`);
}

export function costFormatter(thing, cost) {
    writeMessage(`Wartość egzemplarza ${thing} wynosi: ${cost.toFixed(2)} zł`, true);
}

function writeMessage(message) {
    console.log(message);
}
```


W omawianym przykładzie JavaScript eksportuje dwie funkcje formatujące, które wyświetlają komunikaty w konsoli. Jeżeli chcesz ten kod JavaScriptu wykorzystać w aplikacji, dodaj do pliku *index.ts* polecenia zaprezentowane na listingu 14.10.

Listing 14.10. *Używanie funkcji JavaScriptu w kodzie pliku *index.ts* w katalogu *src**

```
import { SportsProduct, SPORT } from "./product";
import { Cart } from "./cart";
import { sizeFormatter, costFormatter } from "./formatters";

let kayak = new SportsProduct(1, "Kajak", 275, SPORT.Watersports);
let hat = new SportsProduct(2, "Czapka", 22.10, SPORT.Running, SPORT.Watersports);
let ball = new SportsProduct(3, "Piłka", 19.50, SPORT.Soccer);

let cart = new Cart("Bartek");
cart.addProduct(kayak, 1);
cart.addProduct(hat, 1);
cart.addProduct(hat, 2);

sizeFormatter("Cart", cart.itemCount);
costFormatter("Cart", cart.totalPrice);
```

Po zapisaniu zmian w pliku *index.ts* kompilator wykona działanie bez zgłaszania jakichkolwiek problemów, natomiast po wykonaniu kodu zostanie wyświetlony następujący komunikat błędu:

```
internal/modules/cjs/loader.js:613
  throw err;
  ^
Error: Cannot find module 'dist\index.js'
```

Kompilator TypeScriptu nie miał problemu z odszukaniem kodu JavaScriptu, ale nie skopiował go do katalogu *dist*, co oznacza, że środowisko uruchomieniowe Node.js nie było w stanie odnaleźć niezbędnego kodu po uruchomieniu aplikacji.

Dołączanie kodu JavaScriptu w trakcie kompilacji

Kompilator TypeScriptu używa plików JavaScriptu do rozwiązywania zależności podczas kompilacji, choć nie dodaje ich do generowanych danych wyjściowych. Aby zmienić to zachowanie, należy w pliku *tsconfig.json* przypisać opcji *allowJs* wartość *true*, jak pokazałem na listingu 14.11.

Listing 14.11. *Zmiana konfiguracji kompilatora w pliku *tsconfig.json* w katalogu *usingjs**

```
{
  "compilerOptions": {
    "target": "es2020",
    "outDir": "./dist",
    "rootDir": "./src",
    "module": "commonjs",
    "allowJs": true
  }
}
```

Ta zmiana powoduje uwzględnienie podczas kompilacji plików JavaScriptu znajdujących się w katalogu *src*. Wprawdzie pliki JavaScriptu nie zawierają funkcjonalności TypeScriptu, ale kompilator skonwertuje te pliki JavaScriptu w taki sposób, aby odpowiadały wersji JavaScriptu podanej w ustawieniu *target* i formatowi modułu podanemu we właściwości *module*. W omawianym przykładzie żadna z funkcji użytych w pliku *formatters.js* nie ulegnie zmianie, ponieważ wartością właściwości *target* jest *es2020*, choć kompilator dostosuje eksportowane funkcje do formatu modułów *CommonJS*. Jeżeli przeanalizujesz plik *formatters.js* wygenerowany w katalogu *dist*, zobaczysz zmiany wprowadzone przez kompilator.

```
...
"use strict";
Object.defineProperty(exports, "__esModule", { value: true });
function sizeFormatter(thing, count) {
    writeMessage(`Liczba elementów egzemplarza ${thing} wynosi: ${count}`);
}
exports.sizeFormatter = sizeFormatter;
function costFormatter(thing, cost) {
    writeMessage(`Wartość egzemplarza ${thing} wynosi: ${cost.toFixed(2)} zł`, true);
}
exports.costFormatter = costFormatter;
function writeMessage(message) {
    console.log(message);
}
...
```

Skonfigurowanie kompilatora TypeScriptu w taki sposób, aby uwzględniał również pliki JavaScriptu, pozwala na łatwe łączenie kodu TypeScriptu i JavaScriptu, a ponadto gwarantuje prawidłowe wersjonowanie funkcjonalności JavaScriptu.

Sprawdzanie typu kodu JavaScriptu

Kompilator TypeScriptu będzie sprawdzał kod JavaScriptu pod kątem najczęściej popełnianych błędów, o ile opcji konfiguracyjnej *checkJs* została przypisana wartość *true*, jak pokazałem na listingu 14.12. Nie jest to funkcjonalność tak bardzo rozbudowana jak w przypadku stosowanej dla plików kodu TypeScriptu, ale przynajmniej może pomóc we wskazaniu potencjalnych problemów.

Listing 14.12. Konfigurowanie kompilatora w pliku *tsconfig.json* w katalogu *usingjs*

```
{
  "compilerOptions": {
    "target": "es2020",
    "outDir": "./dist",
    "rootDir": "./src",
    "module": "commonjs",
    "allowJs": true,
    "checkJs": true
  }
}
```

Kompilator nie wykryje zmiany wartości właściwości `checkJs` aż do chwili jego ponownego uruchomienia. Po zapisaniu pliku `tsconfig.json` naciśnij klawisze `Ctrl+C` w celu zatrzymania kompilatora, a następnie z poziomu katalogu `usingjs` wydaj polecenie przedstawione na listingu 14.13, które spowoduje ponowne uruchomienie kompilatora.

Listing 14.13. Uruchamianie kompilatora TypeScriptu

```
$ npm start
```

Zdefiniowana w pliku `formatters.js` funkcja `costFormatter()` wywoła zdefiniowaną w tym samym pliku funkcję `writeMessage()` z większą liczbą argumentów niż parametrów. Jest to dozwolone w JavaScriptcie, ponieważ ten język nie nakłada żadnych ograniczeń na liczbę argumentów stosowanych podczas wywoływania funkcji. Jednak kompilator TypeScriptu zgłosi problem, ponieważ jest to jeden z częściej popełnianych błędów.

```
src/formatters.js(6,60): error TS2554: Expected 0-1 arguments, but got 2.
```

Ta funkcjonalność jest użyteczna tylko wtedy, gdy masz możliwość modyfikowania plików JavaScriptu w celu usunięcia problemów zgłaszanych przez kompilator. Być może spotkasz się z następującą sytuacją: masz kod powodujący zgłoszenie błędu przez kompilator TypeScriptu, ale nie możesz go zmienić, ponieważ ten kod jest zgodny z wymaganiami pewnej biblioteki opracowanej przez podmiot zewnętrzny. Jeżeli masz pliki JavaScriptu, z których część można edytować, a części nie wolno, wówczas rozwiązaniem jest dodawanie komentarzy określających pliki przeznaczone do sprawdzenia. W tabeli 14.3 wymienię komentarze stosowane na początku plików JavaScriptu.

Tabela 14.3. Komentarze kontrolujące sprawdzanie plików JavaScriptu

Nazwa	Opis
<code>//@ts-check</code>	Polecenie nakazuje kompilatorowi sprawdzenie zawartości pliku JavaScriptu nawet w przypadku, gdy opcja <code>checkJs</code> w pliku <code>tsconfig.json</code> ma przypisaną wartość <code>false</code>
<code>//@ts-nocheck</code>	Polecenie nakazuje kompilatorowi zignorowanie zawartości pliku JavaScriptu nawet w przypadku, gdy opcja <code>checkJs</code> w pliku <code>tsconfig.json</code> ma przypisaną wartość <code>true</code>

Na listingu 14.14 przedstawiłem przykład użycia komentarza `//@ts-nocheck` w pliku `formatters.js`, aby kompilator nie sprawdzał zawartości tego pliku. Wszystkie pozostałe pliki JavaScriptu w projekcie zostaną sprawdzone, o ile nie mają tego samego komentarza.

Listing 14.14. Wyłączenie sprawdzania kodu JavaScriptu w pliku `formatters.js` w katalogu `src`

```
//@ts-nocheck
```

```
export function sizeFormatter(thing, count) {
  writeMessage(`Liczba elementów egzemplarza ${thing} wynosi: ${count}`);
}

export function costFormatter(thing, cost) {
  writeMessage(`Wartość egzemplarza ${thing} wynosi: ${cost.toFixed(2)} zł`, true);
}
```

```
}

function writeMessage(message) {
  console.log(message);
}
```

Kompilator wykryje wprowadzoną zmianę i ponownie przeprowadzi kompilację projektu, ale tym razem bez wskazanego pliku JavaScriptu, po czym wygeneruje następujące dane wyjściowe:

```
Liczba elementów egzemplarza Cart wynosi: 4
Wartość egzemplarza Cart wynosi: 341.30 zł
```

Opisywanie typów używanych w kodzie JavaScriptu

Kompilator JavaScriptu umieści kod JavaScriptu w projekcie, choć nie oznacza to dostępności informacji o typach statycznych. Wprawdzie kompilator postara się ustalić typy używane w kodzie JavaScriptu, ale w przypadku problemów skorzysta z rozwiązania awaryjnego i wybierze typ `any`, zwłaszcza dla wyników i parametrów funkcji. Zdefiniowana w pliku *formatters.js* funkcja `costFormatter()` będzie potraktowana, jakby została zdefiniowana z następującymi adnotacjami typu:

```
...
export function costFormatter(thing: any, cost: any): any {
  ...
```

Dodanie kodu JavaScriptu do projektu może doprowadzić do powstania luk w systemie sprawdzania typów i tym samym podkopać zalety używania TypeScriptu. Kompilator nie jest w stanie ustalić tego, że funkcja `costFormatter()` będzie otrzymywała wartość typu `number`, co można łatwo sprawdzić przez dodanie do pliku *index.ts* polecenia dostarczającego wartość typu `string`, jak pokazałem na listingu 14.15.

Listing 14.15. *Używanie błędnego typu w kodzie pliku *index.ts* w katalogu *src**

```
import { SportsProduct, SPORT } from "../product";
import { Cart } from "../cart";
import { sizeFormatter, costFormatter } from "../formatters";

let kayak = new SportsProduct(1, "Kajak", 275, SPORT.Watersports);
let hat = new SportsProduct(2, "Czapka", 22.10, SPORT.Running, SPORT.Watersports);
let ball = new SportsProduct(3, "Piłka", 19.50, SPORT.Soccer);

let cart = new Cart("Bartek");
cart.addProduct(kayak, 1);
cart.addProduct(hat, 1);
cart.addProduct(hat, 2);

sizeFormatter("Cart", cart.itemCount);
costFormatter("Cart", `${cart.totalPrice}`);
```

To nowe polecenie wywołuje funkcję `costFormatter()` z dwoma argumentami typu `string`. Kompilator TypeScriptu nie jest w stanie ustalić, że to spowoduje problem, i przeprowadza bezbłędną

kompilację kodu źródłowego. Jednak po uruchomieniu tego kodu metoda `costFormatter()` wywołuje metodę `toFixed()` bez sprawdzenia, czy otrzymana została wartość typu `number`, co prowadzi do powstania następującego błędu podczas działania aplikacji:

```
formatters.js:9: writeMessage(`Wartość ${thing} wynosi: ${cost.toFixed(2)} zł`, true);
TypeError: cost.toFixed is not a function
```

Ten błąd można usunąć przez dostarczenie kompilatorowi informacji o typach opisujących kod JavaScriptu, co z kolei pozwoli na przeprowadzenie sprawdzenia tego kodu podczas kompilacji. Istnieją dwa podejścia w zakresie opisywania typów w kodzie JavaScriptu. Przedstawię je w kolejnych sekcjach.

Używanie komentarzy do opisywania typów

Kompilator TypeScriptu może pobierać informacje o typie zamieszczone w komentarzach JSDoc. JSDoc to popularny język znaczników używany do stosowania adnotacji w postaci komentarzy dla kodu JavaScriptu. Na listingu 14.16 przedstawiłem przykład zastosowania w kodzie pliku *formatters.js* adnotacji JSDoc.

-
- **Wskazówka** Większość edytorów kodu źródłowego pomaga w generowaniu komentarzy JSDoc. Na przykład Visual Studio Code reaguje na utworzenie komentarza i automatycznie generuje listę parametrów funkcji.
-

Listing 14.16. Używanie komentarzy JSDoc w kodzie pliku *formatters.js* w katalogu *src*

```
// @ts-nocheck

export function sizeFormatter(thing, count) {
  writeMessage(`Liczba elementów ${thing} wynosi: ${count}`);
}

/**
 * Sformatowanie wartości uznawanej za walutową.
 * @param { string } thing — nazwa elementu.
 * @param { number } cost — wartość przypisana elementowi.
 */
export function costFormatter(thing, cost) {
  writeMessage(`Wartość ${thing} wynosi: ${cost.toFixed(2)} zł`, true);
}

function writeMessage(message) {
  console.log(message);
}
```

Specyfikacja JSDoc pozwala na wskazywanie typów parametrom funkcji. Komentarz JSDoc na listingu 14.16 wskazuje, że funkcja `costFormatter()` oczekuje parametrów typu `string` i `number`. Informacje o typie są standardową częścią JSDoc, choć zwykle mają za zadanie jedynie dostarczyć odpowiedzi.

Kompilator TypeScriptu odczytuje komentarze JSDoc w celu pobrania informacji o typach używanych w kodzie JavaScriptu. Po dodaniu komentarza JSDoc przedstawionego na listingu 14.16 kompilator wygeneruje następujący komunikat błędu:

```
src/index.ts(15,23): error TS2345: Argument of type 'string' is not assignable to parameter of type 'number'.
```

Kompilator odczytał komentarz JSDoc dla funkcji `costFormatter()` i ustalił, że wartość użyta do wywołania funkcji w pliku *index.ts* nie jest odpowiedniego typu danych.

■ **Wskazówka** Pełną listę znaczników JSDoc rozpoznawanych przez kompilator TypeScriptu znajdziesz na stronie <https://www.typescriptlang.org/docs/handbook/type-checking-javascript-files.html#supported-jsdoc>.

Komentarze JSDoc mogą korzystać ze składni TypeScriptu podczas opisywania bardziej skomplikowanych typów, jak pokazałem na listingu 14.17, na którym znajduje się przykład wykorzystujący unię typów.

Listing 14.17. Opis unii typów w kodzie pliku *formatters.js* w katalogu *src*

// @ts-nocheck

```
export function sizeFormatter(thing, count) {
    writeMessage(`Liczba elementów egzemplarza ${thing} wynosi: ${count}`);
}

/**
 * Sformatowanie wartości uznawanej za walutową.
 * @param { string } thing — nazwa elementu.
 * @param { number | string } cost — wartość przypisana elementowi.
 */
export function costFormatter(thing, cost) {
    if (typeof cost === "number") {
        writeMessage(`Wartość egzemplarza ${thing} wynosi: ${cost.toFixed(2)} zł`, true);
    } else {
        writeMessage(`Wartość egzemplarza ${thing} wynosi: ${cost} zł`);
    }
}

function writeMessage(message) {
    console.log(message);
}
```

Funkcja `costFormatter()` została zmodyfikowana w taki sposób, aby mogła akceptować wartości typów `number` i `string` dla parametru `cost`, co zostało odzwierciedlone w komentarzu JSDoc (zwróć uwagę na zapis `number | string`). Po zapisaniu zmian kod zostanie skompilowany i będą wygenerowane następujące dane wyjściowe:

```
Liczba elementów egzemplarza Cart wynosi: 4
Wartość egzemplarza Cart wynosi: 341.3
```

Używanie plików deklaracji typu

Pliki deklaracji są również określane mianem *plików definicji typów* i zapewniają możliwość opisanie kodu JavaScriptu kompilatorowi TypeScriptu bez konieczności modyfikowania pliku kodu źródłowego. Plik deklaracji typów ma rozszerzenie *.d.ts* oraz nazwę odpowiadającą nazwie pliku JavaScriptu. W celu przygotowania pliku deklaracji dla *formatters.js* należy utworzyć plik o nazwie *formatters.d.ts*. Umieść ten plik (z kodem przedstawionym na listingu 14.18) w katalogu *src*.

Listing 14.18. Zawartość pliku *formatters.d.ts* w katalogu *src*

```
export declare function sizeFormatter(thing: string, count: number): void;
export declare function costFormatter(thing: string, cost: number | string): void;
```

Zawartość pliku deklaracji typów odpowiada opisywanemu plikowi kodu źródłowego. Każde polecenie zawiera słowo kluczowe *declare* wskazujące kompilatorowi, że polecenia opisują typy zdefiniowane w innym miejscu. Kod przedstawiony na listingu 14.18 opisuje typy wyniku i parametrów funkcji wyeksportowane z pliku *formatters.js*.

■ **Wskazówka** Pliki deklaracji typów mają pierwszeństwo przed komentarzami JSDoc, jeśli oba podejścia są używane w celu opisanie kodu JavaScriptu.

Gdy używany jest plik deklaracji, musi zawierać opis całej funkcjonalności zdefiniowanej w odpowiadającym mu pliku JavaScriptu, który jest wykorzystany w aplikacji. W takim przypadku plik deklaracji jest jedynym źródłem informacji używanych przez kompilator TypeScriptu, który już nie analizuje pliku JavaScriptu. W omawianym przykładzie przedstawiony na listingu 14.18 plik deklaracji musi opisywać funkcje *sizeFormatter()* i *costFormatter()*, ponieważ obie są wywoływane w kodzie pliku *index.ts*. Funkcjonalność nieopisana w pliku deklaracji pozostanie niedostępna dla kompilatora TypeScriptu. Aby to pokazać, na listingu 14.19 wprowadziłem zmianę w kodzie funkcji *writeMessage()* zdefiniowanej w pliku *formatters.js* — będzie wyeksportowana w celu jej użycia w pozostałej części aplikacji.

■ **Wskazówka** Kompilator uznaje, że zawartość pliku deklaracji jest prawidłowa. Dlatego to programista jest odpowiedzialny za zagwarantowanie, że zostały wybrane typy obsługiwane przez kod JavaScriptu, a wszystkie funkcje w kodzie JavaScriptu są zaimplementowane zgodnie z opisem.

Listing 14.19. Eksportowanie funkcji w kodzie pliku *formatters.js* w katalogu *src*

```
// @ts-nocheck

export function sizeFormatter(thing, count) {
  writeMessage(`Liczba elementów egzemplarza ${thing} wynosi: ${count}`);
}

/**
 * Sformatowanie wartości uznawanej za walutową.
 * @param { string } thing — nazwa elementu.
 * @param { number | string } cost — wartość przypisana elementowi.
 */
```

```
export function costFormatter(thing, cost) {
  if (typeof cost === "number") {
    writeMessage(`Wartość egzemplarza ${thing} wynosi: ${cost.toFixed(2)} zł`, true);
  } else {
    writeMessage(`Wartość egzemplarza ${thing} wynosi: ${cost} zł`);
  }
}

export function writeMessage(message) {
  console.log(message);
}
```

Kod na listingu 14.20 używa w pliku *index.ts* nowo wyeksportowanej funkcji do wyświetlenia prostego komunikatu.

Listing 14.20. Używanie funkcji w kodzie pliku *index.ts* w katalogu *src*

```
import { SportsProduct, SPORT } from "../product";
import { Cart } from "../cart";
import { sizeFormatter, costFormatter, writeMessage } from "../formatters";

let kayak = new SportsProduct(1, "Kajak", 275, SPORT.Watersports);
let hat = new SportsProduct(2, "Czapka", 22.10, SPORT.Running, SPORT.Watersports);
let ball = new SportsProduct(3, "Piłka", 19.50, SPORT.Soccer);

let cart = new Cart("Bartek");
cart.addProduct(kayak, 1);
cart.addProduct(hat, 1);
cart.addProduct(ball, 2);

sizeFormatter("Cart", cart.itemCount);
costFormatter("Cart", `${cart.totalPrice}`);
writeMessage("Przykładowy komunikat");
```

Kompilator przetworzy zmiany wprowadzone w pliku *index.ts* po jego zapisaniu, a następnie wygeneruje przedstawiony tutaj komunikat błędu:

```
src/index.ts(3,40): error TS2305: Module '"../usingjs/src/formatters"' has no exported
  member 'writeMessage'.
```

Kompilator opiera swoje działanie na pliku deklaracji zawierającym opis funkcjonalności modułu *formatters*. Polecenie *declaration* jest w pliku *formatters.d.ts* niezbędne, aby funkcja *writeMessage()* stała się dostępna dla kompilatora (listing 14.21).

Listing 14.21. Dodawanie polecenia w kodzie pliku *formatters.d.ts* w katalogu *src*

```
export declare function sizeFormatter(thing: string, count: number): void;
export declare function costFormatter(thing: string, cost: number | string): void;
export declare function writeMessage(message: string): void;
```

Gdy plik deklaracji zawiera niezbędne polecenie, projekt zostanie prawidłowo skompilowany, a po uruchomieniu wygeneruje następujące dane wyjściowe:

```
Liczba elementów egzemplarza Cart wynosi: 4
Wartość Cart wynosi: 341.3
Przykładowy komunikat
```

Opisywanie kodu JavaScriptu przygotowanego przez podmioty zewnętrzne

Pliki deklaracji mogą być używane do opisywania kodu JavaScriptu dodawanego do projektu w postaci opracowanych przez podmioty zewnętrzne pakietów, które są dołączane do projektu za pomocą menedżera pakietów Node.js. W oknie powłoki przejdź do katalogu *usingjs*, a następnie wydaj polecenie przedstawione na listingu 14.22, które zainstaluje nowy pakiet w przykładowym projekcie.

Listing 14.22. Dodawanie nowego pakietu w przykładowym projekcie

```
$ npm install debug@4.1.1
```

Do projektu został dodany pakiet narzędziowy debug zapewniający możliwość generowania udekorowanych danych wyjściowych w konsoli JavaScriptu. Zdecydowałem się na jego użycie w projekcie, ponieważ jest to pakiet mały, doskonale dopracowany i powszechnie używany podczas programowania w języku JavaScript.

Kompilator spróbuje ustalić typy dla pakietów opracowanych przez podmioty zewnętrzne, ale podobnie jak w przypadku plików JavaScriptu w projekcie szanse na sukces tej operacji są dość ograniczone. Plik deklaracji typów może być tworzony dla pakietów instalowanych w katalogu *node_modules*, choć ta technika jest dość niewygodna. Lepsze podejście polega na wykorzystaniu dostępnych publicznie definicji, co przedstawiłem w następnej sekcji.

Pierwszym krokiem jest ponowna konfiguracja sposobu, w jaki kompilator TypeScriptu rozwiązuje zależności w modułach, jak pokazałem na listingu 14.23.

Listing 14.23. Konfigurowanie kompilatora w pliku *tsconfig.json* w katalogu *usingjs*

```
{
  "compilerOptions": {
    "target": "es2020",
    "outDir": "./dist",
    "rootDir": "./src",
    "module": "commonjs",
    "allowJs": true,
    "checkJs": true,
    "baseUrl": ".",
    "paths": {
      "**": ["types/**"]
    }
  }
}
```

Właściwość *paths* jest używana do określenia katalogów, które kompilator TypeScriptu będzie próbował wykorzystać podczas rozwiązywania poleceń *import* dla modułów. Konfiguracja użyta na listingu 14.23 nakazuje kompilatorowi sprawdzenie wszystkich pakietów w katalogu o nazwie *types*. Gdy używana jest właściwość *paths*, konieczne jest również zdefiniowanie wartości właściwości *baseUrl* — wartość podana na listingu wskazuje kompilatorowi, że katalog wymieniony we właściwości *paths* znajduje się w tym samym katalogu, który zawiera plik *tsconfig.js*.

Następnym krokiem jest utworzenie katalogu *usingjs/types/debug* i umieszczenie w nim pliku o nazwie *index.d.ts*. Dostarcza to kompilatorowi niestandardowe pliki deklaracji. Położenie wskazywane przez właściwość *paths* musi zawierać katalog odpowiadający nazwie modułu lub pakietu, a także plik deklaracji odpowiadający punktowi wejścia pakietu, czyli zwykle *index.js* (w tym przypadku oznacza to plik deklaracji o nazwie *index.d.ts*). Dla pakietu *debug* typy używane przez pakiet zostaną opisane przez plik *types/debug/index.d.ts*. Po utworzeniu wymienionego pliku dodaj do niego kod przedstawiony na listingu 14.24.

Listing 14.24. Zawartość pliku *index.d.ts* w katalogu *types/debug*

```
declare interface Debug {
  (namespace: string): Debugger
}

declare interface Debugger {
  (...args: string[]): void;
  enabled: boolean;
}

declare var debug: { default: Debug };
export = debug;
```

Proces opisywania modułu przygotowanego przez podmiot zewnętrzny może być skomplikowany. Trudność wynika z wielu powodów, m.in. autorzy pakietu mogli nie przewidzieć, że ktokolwiek będzie próbował opisać ich kod za pomocą typów statycznych. Sytuację jeszcze bardziej komplikuje inny fakt: szeroka gama wersji języka JavaScript i formatów modułów oznacza konieczność stosowania wielu sztuczek do przedstawienia TypeScriptu z opisami, które pozwolą na użyteczne i dokładne zaprezentowanie kodu w module.

Dwa interfejsy pokazane na listingu 14.24 prezentują podstawową funkcjonalność pakietu *debug*, która pozwala na przygotowanie i używanie prostego debuggera. Dwa ostatnie polecenia są wymagane do przedstawienia kompilatorowi TypeScriptu wyeksportowanych pakietów.

■ **Wskazówka** Szczegółowe informacje dotyczące pełnego API dostarczanego przez pakiet *debug* znajdziesz na stronie <https://github.com/visionmedia/debug>.

Aby skorzystać z możliwości oferowanych przez pakiet *debug*, dodaj do pliku *index.ts* w katalogu *src* polecenia przedstawione na listingu 14.25.

Listing 14.25. Używanie pakietu w kodzie pliku *index.ts* w katalogu *src*

```
import { SportsProduct, SPORT } from "../product";
import { Cart } from "../cart";
import { sizeFormatter, costFormatter, writeMessage } from "../formatters";
import debug from "debug";

let kayak = new SportsProduct(1, "Kajak", 275, SPORT.Watersports);
let hat = new SportsProduct(2, "Czapka", 22.10, SPORT.Running, SPORT.Watersports);
let ball = new SportsProduct(3, "Piłka", 19.50, SPORT.Soccer);

let cart = new Cart("Bartek");
```

```

cart.addProduct(kayak, 1);
cart.addProduct(hat, 1);
cart.addProduct(hat, 2);

sizeFormatter("Cart", cart.itemCount);
costFormatter("Cart", `${cart.totalPrice}`);

let db = debug("Przykładowa aplikacja.", true);
db.enabled = true;
db("Treść komunikatu: %0", "Przykładowy komunikat");

```

Kompilator TypeScriptu odszuka plik deklaracji i ustali, że funkcja `debug()` została wywołana ze zbyt dużą liczbą argumentów, co spowoduje wygenerowanie następującego komunikatu błędu:

```

...
src/index.ts(18,31): error TS2554: Expected 1 arguments, but got 2.
...

```

Ten błąd nie zostałby zgłoszony bez pliku deklaracji, ponieważ w przypadku czystego kodu JavaScriptu nie istnieje konieczność dopasowania liczby argumentów wywołania funkcji do liczby zdefiniowanych w niej parametrów, jak to wyjaśniłem w rozdziale 8.

Nie musisz specjalnie wprowadzać błędu, aby się przekonać o znalezieniu przez kompilator pliku deklaracji. Zamiast tego otwórz nowe okno powłoki, przejdź do katalogu *usingjs*, a następnie wydaj polecenie przedstawione na listingu 14.26.

Listing 14.26. Uruchamianie kompilatora TypeScriptu

```
$ tsc --traceResolution
```

Argument `traceResolution` — można go używać także jako ustawienia konfiguracyjnego w pliku *tsconfig.json* — nakazuje kompilatorowi wyświetlanie informacji o postępie podczas próby odnalezienia poszczególnych modułów. Wygenerowane dane wyjściowe mogą być naprawdę obszerne, zwłaszcza w skomplikowanych projektach. W omawianym przykładzie zostanie wygenerowany następujący komunikat błędu:

```

===== Module name 'debug' was successfully resolved to
'C:/usingjs/types/debug/index.d.ts'. =====

```

Wprawdzie w Twoim komputerze mogą być podane inne katalogi, ale komunikat będzie potwierdzał to, że kompilator odnalazł niestandardowy plik deklaracji, który następnie wykorzysta do rozwiązywania zależności w pakiecie `debug`.

Nie twórz deklaracji dla pakietów opracowanych przez podmioty zewnętrzne

Plik deklaracji z listingu 14.24 potwierdza możliwość opisywania publicznie dostępnych pakietów. Jednak nie jest to proces, który polecam, i dlatego też nie zamierzam przedstawiać żadnych szczegółów o różnych sposobach, w jakie można opisywać zawartość pakietu.

Po pierwsze, prawidłowe przedstawienie kodu utworzonego przez kogoś innego może być trudne, a przygotowanie właściwego pliku deklaracji wymaga dokładnej analizy pakietu i dogłębnego zrozumienia sposobu jego działania. Po drugie, niestandardowe deklaracje koncentrują się na funkcjonalności wymaganej natychmiast, a same pliki deklaracji są uaktualniane i rozszerzane wraz z pojawianiem się potrzeby użycia kolejnych funkcjonalności, co prowadzi do powstawania trudnych do zrozumienia oraz niełatwych w zarządzaniu plików definicji typów. Po trzecie, każde nowe wydanie pakietu oznacza konieczność ponownego przeanalizowania pliku deklaracji i zagwarantowania, że prawidłowo odzwierciedla API przedstawiany przez pakiet.

Jednak najważniejszym powodem unikania procesu samodzielnego tworzenia plików deklaracji jest istnienie doskonałej biblioteki wysokiej jakości deklaracji tysięcy pakietów JavaScriptu. Ta biblioteka jest dostępna w ramach projektu Definitely Typed, co dokładniej przedstawię w następnej sekcji. Nieustannie zwiększająca się popularność TypeScriptu oznacza, że coraz większa liczba pakietów jest dostarczana z wbudowanymi deklaracjami typów.

Jeżeli jesteś zdecydowany na samodzielne tworzenie plików deklaracji — lub chcesz dołożyć coś od siebie do projektu Definitely Typed — zajrzyj do opracowanego przez Microsoft przewodnika prezentującego proces opisywania pakietu. Ten przewodnik znajdziesz na stronie <https://www.typescriptlang.org/docs/handbook/declaration-files/introduction.html>.

Używanie plików deklaracji pochodzących z projektu Definitely Typed

Projekt Definitely Typed zawiera pliki deklaracji dla tysięcy pakietów JavaScriptu. Wykorzystanie zasobów tego projektu to najbardziej niezawodny i najszybszy sposób na używanie języka TypeScript podczas pracy z pakietami opracowanymi przez podmioty zewnętrzne — zdecydowanie lepsze rozwiązanie niż samodzielne tworzenie plików deklaracji. Do instalowania pochodzących z projektu Definitely Typed plików deklaracji służy polecenie `npm install`. Dlatego też jeśli chcesz zainstalować plik deklaracji dla pakietu debug, z poziomu katalogu *usingjs* w powłocie wydaj polecenie przedstawione na listingu 14.27.

Listing 14.27. Instalowanie pakietu zawierającego deklaracje typów

```
$ npm install --save-dev @types/debug
```

Nazwa używana dla pakietu Definitely Typed to `@types/`, a po niej znajduje się nazwa pakietu wymagającego opisu w postaci definicji. W przypadku pakietu debug pakiet Definitely Typed nosi nazwę `@types/debug`.

■ **Wskazówka** Zwróć uwagę na brak numeru wersji dla pakietu `@types/debug` na listingu 14.27. Podczas instalowania pakietów `@types` wybór wersji pakietu pozostawiam menedżerowi pakietów Node.js.

Kompilator nie użyje pochodzących z projektu Definitely Typed deklaracji, dopóki nie zostanie zmieniona konfiguracja nakazująca szukania definicji w katalogu *types*, jak pokazałem na listingu 14.28.

-
- **Uwaga** Zmiana konfiguracji jest wymagana, ponieważ dla tego samego pakietu projekt zawiera definicje zarówno przygotowane samodzielnie, jak i pochodzące z Definitely Typed. Nie stanowi to problemu w rzeczywistych projektach i zawsze możesz skorzystać z ustawień konfiguracyjnych do wyboru między deklaracjami własnymi lub pochodzącymi z Definitely Typed dla każdego używanego pakietu.
-

Listing 14.28. Konfigurowanie kompilatora w pliku `tsconfig.json` w katalogu `usingjs`

```
{
  "compilerOptions": {
    "target": "es2020",
    "outDir": "./dist",
    "rootDir": "./src",
    "module": "commonjs",
    "allowJs": true,
    "checkJs": true,
    // "baseUrl": ".",
    // "paths": {
    //   "*": ["types/*"]
    // }
  }
}
```

Z poziomu powłoki przejdź do katalogu `usingjs`, a następnie wydaj polecenie przedstawione na listingu 14.29, aby zobaczyć efekt użycia pakietu Definitely Typed.

Listing 14.29. Uruchamianie kompilatora `TypeScriptu`

```
$ tsc --traceResolution
```

Wygenerowane dane wyjściowe potwierdzają, że kompilator odszukał inny plik deklaracji:

```
===== Type reference directive 'debug' was successfully resolved to
'C:/usingjs/node_modules/@types/debug/index.d.ts' with Package ID
'@types/debug/index.d.ts@4.1.5', primary: true. =====
```

Kompilator sprawdza katalog `node_modules/@types` zawierający podkatalogi odpowiadające pakietom, dla których istnieją pliki deklaracji — stosowany jest dokładnie ten sam wzorzec co w przypadku samodzielnie przygotowanych plików deklaracji. (Nie jest wymagana żadna zmiana konfiguracyjna, aby nakazać kompilatorowi wyszukiwanie deklaracji w katalogu `node_modules/@types`).

W wyniku wprowadzonych modyfikacji używany jest plik deklaracji Definitely Typed, który zapewnia pełny opis API przedstawionego przez pakiet `debug`. Na listingu 14.30 poprawiłem liczbę argumentów używanych do wywoływania funkcji `debug()`.

Listing 14.30. Używanie funkcjonalności pakietu w kodzie pliku `index.ts` w katalogu `src`

```
import { SportsProduct, SPORT } from "./product";
import { Cart } from "./cart";
import { sizeFormatter, costFormatter, writeMessage } from "./formatters";
import debug from "debug";
```

```
let kayak = new SportsProduct(1, "Kajak", 275, SPORT.Watersports);
let hat = new SportsProduct(2, "Czapka", 22.10, SPORT.Running, SPORT.Watersports);
let ball = new SportsProduct(3, "Piłka", 19.50, SPORT.Soccer);

let cart = new Cart("Bartek");
cart.addProduct(kayak, 1);
cart.addProduct(hat, 1);
cart.addProduct(ball, 2);

sizeFormatter("Cart", cart.itemCount);
costFormatter("Cart", `${cart.totalPrice}`);

let db = debug("Przykładowa aplikacja.");
db.enabled = true;
db("Treść komunikatu: %0", "Przykładowy komunikat");
```

Po zapisaniu zmian kompilator przeprowadzi kompilację projektu z użyciem polecenia `npm start`, o ile nie zostało wydane już wcześniej. Kompilator wykorzysta nowy plik deklaracji, który będzie zawierał informacje dotyczące nowej metody użytej na listingu. Skompilowany kod spowoduje wygenerowanie następujących danych wyjściowych:

```
Liczba elementów egzemplarza Cart wynosi: 4
Wartość Cart wynosi: 341.3
Przykładowa aplikacja. Treść komunikatu: %0 Przykładowy komunikat +0ms
```

Używanie pakietów zawierających deklaracje typu

W miarę coraz większej popularności języka TypeScript pakiety zaczęły zawierać także pliki deklaracji, co eliminuje konieczność pobierania dodatkowych danych. Najłatwiejszy sposób na sprawdzenie, czy projekt zawiera plik deklaracji, polega na zainstalowaniu pakietu i sprawdzeniu katalogu `node_modules`. Z poziomu powłoki przejdź do katalogu `usingjs`, a następnie wydaj polecenie przedstawione na listingu 14.31, aby w ten sposób dodać pakiet do przykładowego projektu.

Listing 14.31. Dodawanie kolejnego pakietu do projektu

```
$ npm install chalk@4.1.0
```

Działanie pakietu `Chalk` polega na dostarczeniu stylów dla danych wyjściowych generowanych w konsoli. Przeanalizuj zawartość katalogu `node_modules/chalk`, a zobaczysz, że zawiera podkatalog `types` z plikiem `index.d.ts`. Z kolei plik `node_modules/package.json` zawiera właściwość `types` wskazującą kompilatorowi katalog, w którym znajduje się plik deklaracji.

```
...
"types": "types/index.d.ts",
...
```

Aby potwierdzić możliwość znalezienia pliku deklaracji pakietu `Chalk` przez kompilator TypeScriptu, dodaj do pliku `index.ts` w katalogu `src` polecenia przedstawione na listingu 14.32.

Listing 14.32. Dodawanie poleceń w kodzie pliku *index.ts* w katalogu *src*

```
import { SportsProduct, SPORT } from "./product";
import { Cart } from "./cart";
import { sizeFormatter, costFormatter, writeMessage } from "./formatters";
import debug from "debug";
import chalk from "chalk";

let kayak = new SportsProduct(1, "Kajak", 275, SPORT.Watersports);
let hat = new SportsProduct(2, "Czapka", 22.10, SPORT.Running, SPORT.Watersports);
let ball = new SportsProduct(3, "Piłka", 19.50, SPORT.Soccer);

let cart = new Cart("Bartek");
cart.addProduct(kayak, 1);
cart.addProduct(hat, 1);
cart.addProduct(ball, 2);

sizeFormatter("Cart", cart.itemCount);
costFormatter("Cart", `${cart.totalPrice}`);

console.log(chalk.greenBright("Sformatowany komunikat"));
console.log(chalk.notAColor("Sformatowany komunikat"));
```

Jedną z funkcjonalności dostarczanych przez pakiet Chalk polega na kolorowaniu tekstu wyświetlanego w konsoli. Pierwsze polecenie nakazuje pakietowi Chalk zastosowanie koloru zielonego, natomiast drugie wskazuje nieistniejącą właściwość. Po zapisaniu zmian w pliku *index.ts* kompilator wykorzysta plik deklaracji i wygeneruje następujący komunikat błędu:

```
src/index.ts(20,19): error TS2339: Property 'notAColor' does not exist on type 'Chalk & {
↳ supportsColor: ColorSupport; }'.
```

W celu umożliwienia kompilatorowi obsługi funkcjonalności pochodzącej z pakietu Chalk trzeba w pliku konfiguracyjnym *tsconfig.json* dodać opcję przedstawioną na listingu 14.33.

Listing 14.33. Konfiguracja kompilatora w pliku *tsconfig.json* w katalogu *src*

```
{
  "compilerOptions": {
    "target": "es2020",
    "outDir": "./dist",
    "rootDir": "./src",
    "module": "commonjs",
    "allowJs": true,
    "checkJs": true,
    "esModuleInterop": true
  }
}
```

Aby poznać proces używany przez kompilator do odszukania pliku deklaracji, z poziomu katalogu *usings* w powłoce wydaj polecenie przedstawione na listingu 14.34.

Listing 14.34. Uruchamianie kompilatora TypeScriptu

```
$ tsc --traceResolution
```

Dane wyjściowe generowane po użyciu argumentu `traceResolution` są obszerne i jeśli przeczytasz wszystkie komunikaty, wówczas poznasz poszczególne katalogi sprawdzane przez kompilator podczas szukania plików deklaracji, a także efekt ustawień zamieszczonych w pliku `package.json` pakietu Chalk.

```
...
'package.json' has 'types' field 'types/index.d.ts' that references 'C:/usingjs/node_modules/chalk/types/index.d.ts'.
...
File 'C:/usingjs/node_modules/chalk/index.d.ts' exist - use it as a name resolution result.
...
```

Na listingu 14.35 pokazałem usunięcie polecenia prowadzącego do błędu podczas kompilacji, aby przykładowa aplikacja mogła zostać skompilowana i wykonana.

Listing 14.35. Wyłączenie nieprawidłowego polecenia w kodzie pliku `index.ts` w katalogu `src`

```
import { SportsProduct, SPORT } from "./product";
import { Cart } from "./cart";
import { sizeFormatter, costFormatter, writeMessage } from "./formatters";
import debug from "debug";
import chalk from "chalk";

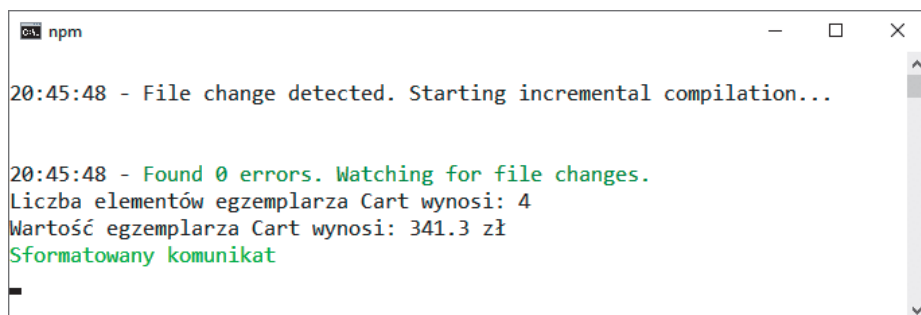
let kayak = new SportsProduct(1, "Kajak", 275, SPORT.Watersports);
let hat = new SportsProduct(2, "Czapka", 22.10, SPORT.Running, SPORT.Watersports);
let ball = new SportsProduct(3, "Piłka", 19.50, SPORT.Soccer);

let cart = new Cart("Bartek");
cart.addProduct(kayak, 1);
cart.addProduct(hat, 1);
cart.addProduct(hat, 2);

sizeFormatter("Cart", cart.itemCount);
costFormatter("Cart", `${cart.totalPrice}`);

console.log(chalk.greenBright("Sformatowany komunikat"));
//console.log(chalk.notAColor("Sformatowany komunikat"));
```

Kod zostanie skompilowany i wykonany, a polecenie sformatowane przez pakiet Chalk będzie wyświetlone w innym kolorze, jak pokazałem na rysunku 14.1.



Rysunek 14.1. Przykład użycia pakietu Chalk

Generowanie plików deklaracji

Jeżeli kod będzie używany przez inne projekty, wówczas kompilatorowi można zlecić zadanie wygenerowania plików deklaracji z czystym kodem JavaScriptu. Efektem będzie zachowanie informacji o typach dla innych programistów TypeScriptu i jednocześnie możliwość stosowania zwykłego kodu JavaScriptu w projekcie.

Kompilator nie wygeneruje plików deklaracji w przypadku użycia opcji `allowJS`, co oznacza konieczność usunięcia zależności od pliku *formatters.js*, więc projekt będzie utworzony całkowicie w języku TypeScript. W katalogu *src* umieść plik *tsFormatters.ts* i dodaj kod przedstawiony na listingu 14.36.

Listing 14.36. Zawartość pliku *tsFormatters.ts* w katalogu *src*

```
export function sizeFormatter(thing: string, count: number): void {
    writeMessage(`Liczba elementów egzemplarza ${thing} wynosi: ${count}`);
}

export function costFormatter(thing: string, cost: number | string): void {
    if (typeof cost === "number") {
        writeMessage(`Wartość egzemplarza ${thing} wynosi: ${cost.toFixed(2)} zł`);
    } else {
        writeMessage(`Wartość egzemplarza ${thing} wynosi: ${cost} zł`);
    }
}

export function writeMessage(message: string): void {
    console.log(message);
}
```

To jest kod JavaScriptu pochodzący z pliku *formatters.js*, choć z adnotacjami typu. Na listingu 14.37 przedstawiłem uaktualnienie pliku *index.ts*, aby zależnością był plik TypeScriptu zamiast JavaScriptu.

■ **Ostrzeżenie** Bardzo ważne znaczenie ma podążanie za zmianami w tym procesie, ponieważ wyłączenie opcji `allowJS` jedynie uniemożliwia kompilatorowi dodawanie pliku JavaScriptu do katalogu wyjściowego. Włączenie wymienionej opcji natomiast nie chroni przed kodem TypeScriptu zależnym od pliku JavaScriptu, co może prowadzić do błędów podczas działania aplikacji, ponieważ środowisko uruchomieniowe JavaScriptu nie będzie w stanie odszukać wszystkich niezbędnych mu plików.

Listing 14.37. Uaktualnianie zależności w kodzie pliku *index.ts* w katalogu *src*

```
import { SportsProduct, SPORT } from "./product";
import { Cart } from "./cart";
import { sizeFormatter, costFormatter, writeMessage } from "./tsFormatters";
import debug from "debug";
import chalk from "chalk";

let kayak = new SportsProduct(1, "Kajak", 275, SPORT.Watersports);
```

```
let hat = new SportsProduct(2, "Czapka", 22.10, SPORT.Running, SPORT.Watersports);
let ball = new SportsProduct(3, "Piłka", 19.50, SPORT.Soccer);

let cart = new Cart("Bartek");
cart.addProduct(kayak, 1);
cart.addProduct(hat, 1);
cart.addProduct(hat, 2);

sizeFormatter("Cart", cart.itemCount);
costFormatter("Cart", `${cart.totalPrice}`);

console.log(chalk.greenBright("Sformatowany komunikat"));
//console.log(chalk.notAColor("Sformatowany komunikat"));
```

Kod przedstawiony na listingu 14.38 zmienia konfigurację kompilatora przez wyłączenie właściwości `allowJS` i `checkJS` oraz włączenie automatycznego generowania plików deklaracji.

Listing 14.38. Konfigurowanie kompilatora w pliku `tsconfig.json` w katalogu `usingjs`

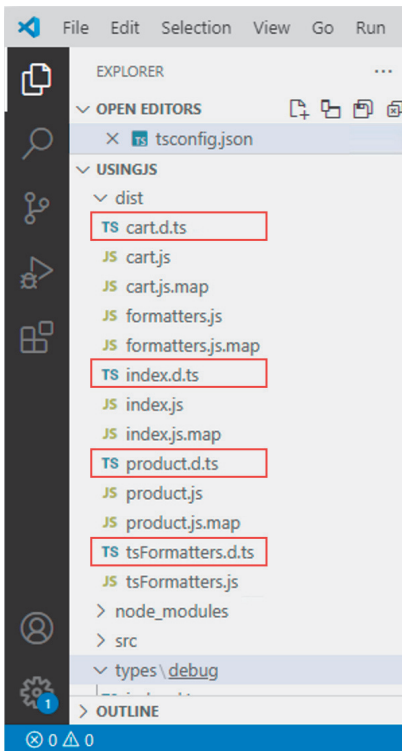
```
{
  "compilerOptions": {
    "target": "es2020",
    "outDir": "./dist",
    "rootDir": "./src",
    "module": "commonjs",
    // "allowJs": true,
    // "checkJs": true,
    // "baseUrl": ".",
    // "paths": {
    //   "*": ["types/*"]
    // },
    "esModuleInterop": true,
    "declaration": true
  }
}
```

Kompilator nie wygeneruje pliku deklaracji aż do jego ponownego uruchomienia. Za pomocą skrótu klawiszowego `Ctrl+C` zatrzymaj kompilator, a następnie z poziomu katalogu `usingjs` w powłoce wydaj polecenie przedstawione na listingu 14.39, aby ponownie uruchomić kompilator.

Listing 14.39. Uruchamianie kompilatora TypeScriptu

```
$ npm start
```

Gdy właściwość `declaration` ma przypisaną wartość `true`, kompilator będzie generował pliki deklaracji w katalogu `dist` opisujące funkcje wyeksportowane przez poszczególne pliki TypeScriptu, jak pokazałem na rysunku 14.2.



Rysunek 14.2. Wygenerowane pliki deklaracji w projekcie

Podsumowanie

W tym rozdziale pokazałem, jak powinieneś pracować z JavaScriptem w projekcie TypeScript. Wyjaśniłem sposób konfiguracji kompilatora w celu przetwarzania i sprawdzania plików JavaScriptu, a także pokazałem, jak używać plików deklaracji do opisywania kodu JavaScriptu kompilatorowi. W następnej części książki przystąpię do utworzenia serii aplikacji internetowych opierających się na języku TypeScript. Zacznę od aplikacji utworzonej tylko za pomocą TypeScriptu, a następnie pokażę wykorzystanie frameworków Angular, React i Vue.js podczas tworzenia aplikacji internetowych w języku TypeScript.

CZĘŚĆ III



Tworzenie aplikacji internetowych



Tworzenie aplikacji internetowej TypeScriptu — część I

W tej części książki pokażę, jak język TypeScript wpisuje się w proces tworzenia aplikacji internetowych z wykorzystaniem trzech popularnych frameworków: Angular, React i Vue.js. We wszystkich przypadkach dokładnie przedstawię proces tworzenia projektu, konfigurowania usługi sieciowej oraz przygotowania prostej aplikacji internetowej. W tym rozdziale zajmę się utworzeniem aplikacji internetowej TypeScriptu bez użycia któregośkolwiek z wymienionych wcześniej frameworków. Dostarczę w ten sposób podstawy pomocne w zrozumieniu funkcjonalności oferowanej przez te frameworki, a także kontekstu stosowania funkcji TypeScriptu.

Wprawdzie nie zalecam tworzenia rzeczywistych aplikacji bez wykorzystania frameworków, ale praca nad taką aplikacją dostarcza wiele informacji na temat TypeScriptu jako nowoczesnego języka programowania, dlatego warto zapoznać się z takim projektem. W tabeli 15.1 wymieniałem opcje kompilatora TypeScriptu użyte w rozdziale.

Tabela 15.1. *Opcje kompilatora TypeScriptu użyte w rozdziale*

Opcja	Opis
jsx	Ta opcja określa sposób przetwarzania elementów HTML-a w plikach JSX/TSX
jsxFactory	Ta opcja określa nazwę funkcji fabryki, która jest używana do zastępowania elementów HTML-a w plikach JSX/TSX
outDir	Ta opcja określa katalog, w którym zostaną umieszczone wygenerowane pliki JavaScriptu
rootDir	Ta opcja określa katalog główny używany przez kompilator do wyszukiwania plików TypeScriptu
target	Ta opcja określa docelową wersję języka JavaScript, dla której kompilator wygeneruje kod

Przygotowanie projektu

Aby przygotować się do tego rozdziału, w dogodnym miejscu utwórz katalog o nazwie *webapp*. Następnie przejdź do powłoki, wejdź do katalogu *webapp* i wydaj w nim polecenia przedstawione na listingu 15.1. Drugie polecenie inicjalizuje katalog do użycia i nakazuje menedżerowi pakietów Node.js utworzenie pliku o nazwie *package.json* zawierającego listę plików dodanych do projektu.

■ **Wskazówka** Przykładowy projekt dla tego rozdziału, a także projekty dla pozostałych rozdziałów książki zostały umieszczone w archiwum dostępnym pod adresem <https://ftp.helion.pl/przyklady/tys4o2.zip>.

Listing 15.1. Utworzenie pliku *package.json* w przykładowym projekcie

```
$ cd webapp
$ npm init --yes
```

W rozdziale zamierzam użyć zestawu narzędzi kompilatora TypeScriptu, aby dzięki temu pokazać najczęściej stosowany sposób pracy nad aplikacją internetową. Takie podejście wymaga lokalnej instalacji pakietu TypeScriptu w projekcie — nie można wykorzystać pakietu zainstalowanego globalnie w rozdziale 1. Z poziomu katalogu *webapp* w powłocie wydaj polecenie przedstawione na listingu 15.2, którego wynikiem jest instalacja pakietu TypeScriptu.

Listing 15.2. Dodawanie pakietu kompilatora TypeScriptu za pomocą menedżera pakietów Node.js

```
$ npm install --save-dev typescript@4.2.2
```

W trakcie pracy nad aplikacją dodam kolejne niezbędne pakiety, lecz na tę chwilę pakiet zawierający kompilator TypeScriptu jest w zupełności wystarczający. Aby skonfigurować kompilator TypeScriptu, należy dodać do katalogu *webapp* plik o nazwie *tsconfig.json* z zawartością przedstawioną na listingu 15.3.

Listing 15.3. Zawartość pliku *tsconfig.json* w katalogu *webapp*

```
{
  "compilerOptions": {
    "target": "es2020",
    "outDir": "./dist",
    "rootDir": "./src"
  }
}
```

Ta konfiguracja wskazuje kompilatorowi, że docelową wersją JavaScriptu jest ES2020, pliki kodu źródłowego znajdują się w katalogu *src*, a wygenerowane pliki aplikacji mają trafić do katalogu *dist*. Aby przygotować punkt wejścia dla aplikacji, utwórz katalog *src* i umieść w nim plik o nazwie *index.ts* z zawartością przedstawioną na listingu 15.4.

Listing 15.4. Zawartość pliku *index.ts* w katalogu *src*

```
console.log("Aplikacja internetowa");
```

Z poziomu katalogu *webapp* w powłoce wydaj polecenie przedstawione na listingu 15.5, które skompiluje plik *index.ts* i uruchomi kod JavaScriptu wygenerowany w wyniku kompilacji.

Listing 15.5. Kompilowanie i wywoływanie wygenerowanego kodu JavaScriptu

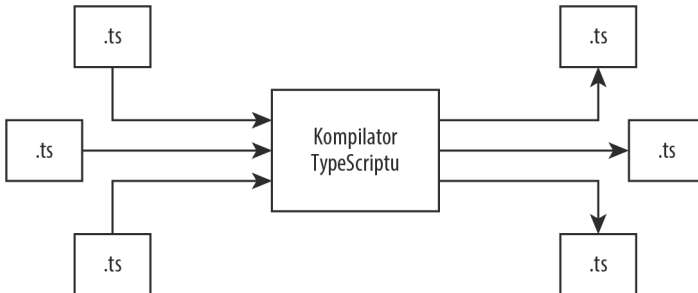
```
$ tsc
$ node dist/index.js
```

Wynikiem wykonania kodu będą następujące dane wyjściowe:

```
Aplikacja internetowa
```

Przygotowanie zestawu narzędzi

Podczas tworzenia aplikacji internetowej zestaw narzędzi wykorzystuje się do kompilowania kodu źródłowego oraz do przygotowania i wykonania aplikacji przez środowisko uruchomieniowe JavaScriptu. W chwili obecnej kompilator TypeScriptu to jedyne narzędzie w omawianym projekcie, jak pokazałem na rysunku 15.1.



Rysunek 15.1. Początkowy zestaw narzędzi omawianego projektu

Narzędzia programistyczne pozostają ukryte, gdy wykorzystywany jest framework taki jak Angular, React lub Vue.js (jak to pokażę w późniejszych rozdziałach). W tym rozdziale zamierzam zainstalować i skonfigurować poszczególne narzędzia oraz pokazać, jak ze sobą współdziałają.

Dodawanie obsługi paczek

Gdy aplikacja jest uruchamiana za pomocą Node.js w katalogu projektu, wszystkie polecenia `import` mogą być rozwiązane poprzez użycie kodu JavaScriptu wygenerowanego przez kompilator TypeScriptu lub pakiety zainstalowane w katalogu *node_modules*.

Środowisko uruchomieniowe JavaScriptu uruchamia aplikację, poczynając od jej punktu wejścia — w omawianym przykładzie jest to plik *index.js* skompilowany na podstawie pliku *index.ts* — i przetwarza zdefiniowane w nim polecenia `import`. Dla każdego polecenia `import` środowisko uruchomieniowe rozwiązuje zależność i wczytuje żądane moduły, którymi mogą być inne pliki JavaScriptu. Każde polecenie `import` zadeklarowane w nowym pliku JavaScriptu będzie przetwarzane w dokładnie taki sam sposób, co pozwala na rozwiązywanie wszystkich zależności, aby można było wykonać kod.

Środowisko uruchomieniowe JavaScriptu nie ma wcześniej żadnych informacji o poleceniach `import`, które mogą się znajdować w poszczególnych plikach kodu źródłowego. Dlatego też wcześniej nie wiadomo, które z plików JavaScriptu będą wymagane. Na szczęście to nie ma znaczenia, ponieważ wyszukiwanie plików w celu rozwiązania zależności jest względnie szybką operacją, a wszystkie pliki lokalne są łatwo dostępne.

Takie podejście nie sprawdza się dobrze w przypadku aplikacji internetowej, ponieważ nie ma ona bezpośredniego dostępu do systemu plików. Zamiast tego pliki muszą być żądane poprzez HTTP — ta operacja może być wolna i kosztowna oraz nie pozwala na łatwe sprawdzanie wielu katalogów podczas rozwiązywania zależności. Dlatego też stosowany jest tzw. *bundler* odpowiedzialny za rozwiązywanie zależności podczas kompilacji i umieszczenie zawartości wszystkich plików używanych przez aplikację w jednym pliku. Dzięki wykorzystaniu programu tworzącego paczkę aplikacji jedno żądanie HTTP pozwala na dostarczenie niezbędnego do uruchomienia aplikacji całego kodu JavaScriptu oraz kodu innego typu, np. CSS, który można umieścić w pojedynczym pliku wygenerowanym przez wspomniane narzędzie typu *bundler*. W wyniku połączenia różnych plików z kodem otrzymujemy tzw. *paczkę*. W trakcie procesu tworzenia paczki kod może być zminimalizowany i skompresowany, co zmniejsza ilość danych koniecznych do przekazania do klienta. Ogromne aplikacje mogą być dzielone na wiele paczek, więc kod lub treść opcjonalna mogą być łączone oddzielnie i wczytywane tylko wtedy, gdy zachodzi potrzeba.

Najczęściej używanym pakietem przeznaczonym do tworzenia paczek jest *webpack* — pakiet o znaczeniu kluczowym w zestawach narzędzi używanych przez frameworki React, Angular i Vue.js, choć nie zawsze będzie zachodziła potrzeba bezpośredniej pracy z nim, o czym przekonasz się w kolejnych rozdziałach. Praca z pakietem *webpack* może być skomplikowana, choć jest wspomagana przez wiele usprawniających pracę z zestawem narzędzi dodatków, które można tworzyć dla niemal dowolnego typu projektu. Z poziomu katalogu *webapp* w powłoce wydaj przedstawione na listingu 15.6 polecenia, które spowodują dodanie pakietów *webpack* do przykładowego projektu.

Listing 15.6. Dodawanie pakietów *webpack* do przykładowego projektu

```
$ npm install --save-dev webpack@5.17.0
$ npm install --save-dev webpack-cli@4.5.0
$ npm install --save-dev ts-loader@8.0.14
```

Pakiet *webpack* zawiera całą funkcjonalność związaną z tworzeniem i obsługą paczek, natomiast pakiet *webpack-cli* zapewnia tę funkcjonalność w powłoce. Do pracy z różnymi rodzajami treści *webpack* używa tzw. *procedur wczytujących* — pakiet *ts-loader* zapewnia

obsługę kompilacji plików TypeScriptu, które następnie są umieszczane w paczce tworzonej przez webpacka. Do konfiguracji webpacka niezbędne jest umieszczenie w katalogu *webapp* pliku o nazwie *webpack.config.js* i zawartości przedstawionej na listingu 15.7.

Listing 15.7. Zawartość pliku *webpack.config.js* w katalogu *webapp*

```
module.exports = {
  mode: "development",
  devtool: "inline-source-map",
  entry: "./src/index.ts",
  output: { filename: "bundle.js" },
  resolve: { extensions: [".ts", ".js"] },
  module: {
    rules: [
      { test: /\.ts$/, use: "ts-loader", exclude: /node_modules/ }
    ]
  }
};
```

Ustawienie entry nakazuje pakietowi webpack podczas rozwiązywania zależności aplikacji rozpoczęcie pracy od pliku *src/index.ts*, natomiast ustawienie output określa, że plik paczki ma otrzymać nazwę *bundle.js*. Pozostałe ustawienia pozwalają na skonfigurowanie webpacka i wykorzystanie *ts-loader* do przetwarzania plików, które mają rozszerzenie *.ts*.

■ **Wskazówka** Szczegółowe informacje oraz pełne omówienie wszystkich opcji konfiguracyjnych obsługiwanych przez pakiet webpack znajdziesz pod adresem <https://webpack.js.org/>.

Z poziomu katalogu *webapp* w powłoce wydaj polecenie przedstawione na listingu 15.8, aby w ten sposób utworzyć plik paczki.

Listing 15.8. Tworzenie pliku w postaci paczki

```
$ npx webpack
```

webpack przetworzy wszystkie zależności projektu i wykorzysta pakiet *ts-loader* do kompilacji znalezionych plików TypeScriptu. Wynikiem działania wymienionego polecenia będzie wygenerowanie następujących danych wyjściowych:

```
asset bundle.js 788 bytes [emitted] (name: main)
./src/index.ts 25 bytes [built] [code generated]
webpack 5.17.0 compiled successfully in 1865 ms
```

Plik *bundle.js* zostanie umieszczony w katalogu *dist*. Wykonanie kodu umieszczonego w paczce następuje po wydaniu polecenia, które wymieniałem na listingu 15.9.

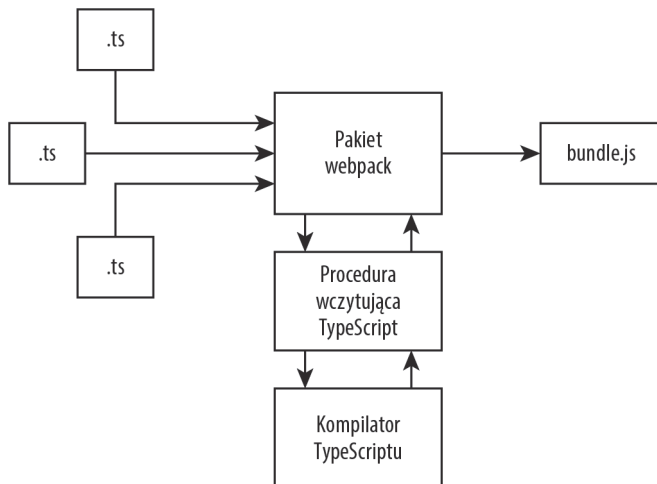
Listing 15.9. Wywoływanie pliku w postaci paczki

```
$ node dist/bundle.js
```

Wprawdzie w tej chwili projekt zawiera tylko jeden plik TypeScriptu, ale paczka jest niezależna od innych plików i pozostaje taka nawet wtedy, gdy aplikacja stanie się znacznie bardziej skomplikowana. Wykonanie kodu znajdującego się w paczce spowoduje wygenerowanie następujących danych wyjściowych:

Aplikacja internetowa

Dodanie pakietu webpack i innych narzędzi wspomagających jego działanie spowodowało zmianę zestawu narzędzi programistycznych projektu, jak pokazałem na rysunku 15.2.



Rysunek 15.2. Dodanie do projektu obsługi paczek

Dodawanie programistycznego serwera WWW

Serwer WWW jest wymagany w celu dostarczenia pliku paczki do przeglądarki WWW, w której zostanie następnie wykonany. WDS, czyli Webpack Dev Server, to serwer HTTP zintegrowany z pakietem webpack i zapewniający obsługę wywoływania automatycznego odświeżenia strony w przeglądarce po zmianie pliku kodu źródłowego i wygenerowaniu nowego pliku paczki. Z poziomu katalogu *webapp* w powłocie wydaj polecenie przedstawione na listingu 15.10, aby zainstalować pakiet WDS.

Listing 15.10. Dodawanie pakietu WDS

```
$ npm install --save-dev webpack-dev-server@3.11.2
```

Kolejnym krokiem jest zmiana konfiguracji pakietu webpack, aby zawierała podstawową konfigurację serwera WDS, jak pokazałem na listingu 15.11.

Listing 15.11. Zmianie konfiguracji w pliku *webpack.config.js* w katalogu *webapp*

```

module.exports = {
  mode: "development",
  devtool: "inline-source-map",
  entry: "./src/index.ts",
  output: { filename: "bundle.js" },
  resolve: { extensions: [".ts", ".js"] },
  module: {
    rules: [
      { test: /\.ts$/, use: "ts-loader", exclude: /node_modules/ }
    ]
  },
  devServer: {
    contentBase: "./assets",
    port: 4500
  }
};

```

Nowe ustawienia konfiguracyjne nakazują serwerowi WDS odszukanie w katalogu *assets* wszystkich plików nieznajdujących się w paczce i nasłuchiwanie żądań HTTP na porcie 4500. Aby dostarczyć serwerowi WDS plik dokumentu HTML-a, który będzie mógł być używany w odpowiedzi udzielanej przeglądarce WWW, w katalogu projektu utwórz podkatalog *assets* i umieść w nim plik o nazwie *index.html* z zawartością przedstawioną na listingu 15.12.

Listing 15.12. Zawartość pliku *index.html* w katalogu *assets*

```

<!DOCTYPE html>
<html>
<head>
  <title>Aplikacja internetowa</title>
  <script src="bundle.js"></script>
</head>
<body>
  <div id="app">Miejsce zarezerwowane dla aplikacji internetowej</div>
</body>
</html>

```

Gdy przeglądarka WWW otrzyma plik HTML-a, wówczas przetworzy jego zawartość i wygeneruje element `<script>`, który z kolei wykona żądanie HTTP do pliku *bundle.js* zawierającego kod JavaScriptu aplikacji.

Uruchomienie serwera WWW następuje po wydaniu z poziomu katalogu *webapp* w powłoce polecenia przedstawionego na listingu 15.13.

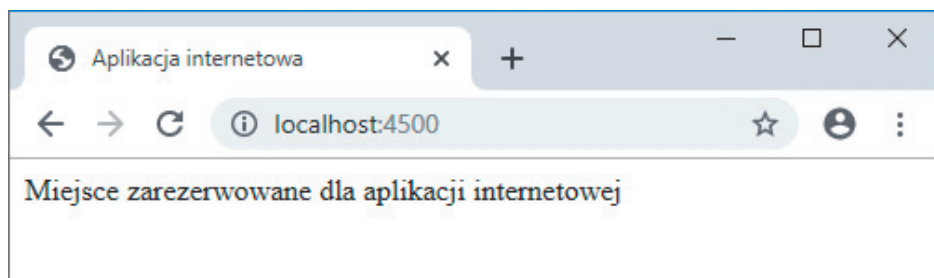
Listing 15.13. Uruchamianie programistycznego serwera WWW

```
$ npx webpack-dev-server
```

Serwer WWW zostanie uruchomiony i nastąpi utworzenie paczki. Jednak katalog *dist* nie jest już używany do przechowywania plików — dane wyjściowe procesu tworzenia paczki są przechowywane w pamięci i wykorzystywane do udzielania odpowiedzi na żądanie HTTP bez konieczności utworzenia pliku na dysku. Po uruchomieniu serwera i utworzeniu paczki zostaną wygenerowane następujące dane wyjściowe:

```
i [wds]: Project is running at http://localhost:4500/
i [wds]: webpack output is served from /
i [wds]: Content not from webpack is served from ./assets i [wds]: asset bundle.js 364 KiB
[emitted] (name: main) runtime modules 430 bytes 3 modules
cacheable modules 335 KiB
  modules by path ./node_modules/webpack-dev-server/client/ 20.9 KiB 10 modules
  modules by path ./node_modules/html-entities/lib/*.js 61 KiB 5 modules
  modules by path ./node_modules/url/ 37.4 KiB 3 modules
  modules by path ./node_modules/querystring/*.js 4.51 KiB
    ./node_modules/querystring/index.js 127 bytes [built] [code generated]
    ./node_modules/querystring/decode.js 2.34 KiB [built] [code generated]
    ./node_modules/querystring/encode.js 2.04 KiB [built] [code generated]
  modules by path ./node_modules/webpack/hot/*.js 1.42 KiB
    ./node_modules/webpack/hot/emitter.js 75 bytes [built] [code generated]
    ./node_modules/webpack/hot/log.js 1.34 KiB [built] [code generated]
  ./node_modules/webpack/hot/ sync nonrecursive ^\\.\\.\\/log$ 170 bytes [built]
[code generated] webpack 5.17.0 compiled successfully in 2087 Microsoft
i [wdm]: Compiled successfully.
```

Dokładne znaczenie wygenerowanych komunikatów nie jest tutaj istotne — przedstawiają one ogólny proces. Po uruchomieniu serwera przejdź do nowego okna przeglądarki WWW i wpisz adres `http://localhost:4500` — podany numer portu został skonfigurowany dla serwera WDS, który nasłuchuje na nim żądań HTTP. Zawartość pliku `index.html` zostanie wyświetlona przez przeglądarkę WWW, jak pokazałem na rysunku 15.3.



Rysunek 15.3. Dokument HTML-a wyświetlony przez przykładową aplikację

W przeglądarce WWW przejdź do narzędzi programistycznych i kliknij kartę *Console*, a zobaczysz dane wyjściowe wygenerowane przez polecenie `console.log()` znajdujące się w pliku `index.ts`:

Aplikacja internetowa

Po uruchomieniu serwera WDS pakiet webpack działa w trybie monitorowania i generuje nową paczkę po wykryciu każdej zmiany w plikach kodu źródłowego. Podczas procesu kompilacji WDS wstrzykuje do pliku JavaScriptu dodatkowy kod nawiązujący połączenie zwrotne z serwerem i oczekujący na sygnał odświeżenia strony w przeglądarce WWW — ten sygnał jest wysyłany dla każdej nowej paczki. Efektem jest automatyczne odświeżenie strony w przeglądarce WWW po wykryciu i przetworzeniu każdej zmiany. Możesz się o tym przekonać przez dodanie polecenia do pliku `index.ts`, jak pokazałem na listingu 15.14.

- **Wskazówka** Funkcjonalność odświeżenia strony w przeglądarce WWW działa tylko dla plików kodu źródłowego i nie dotyczy pliku HTML-a znajdującego się w katalogu *assets*. Zmiana pliku HTML-a przyniesie efekt dopiero po ponownym uruchomieniu serwera WDS.

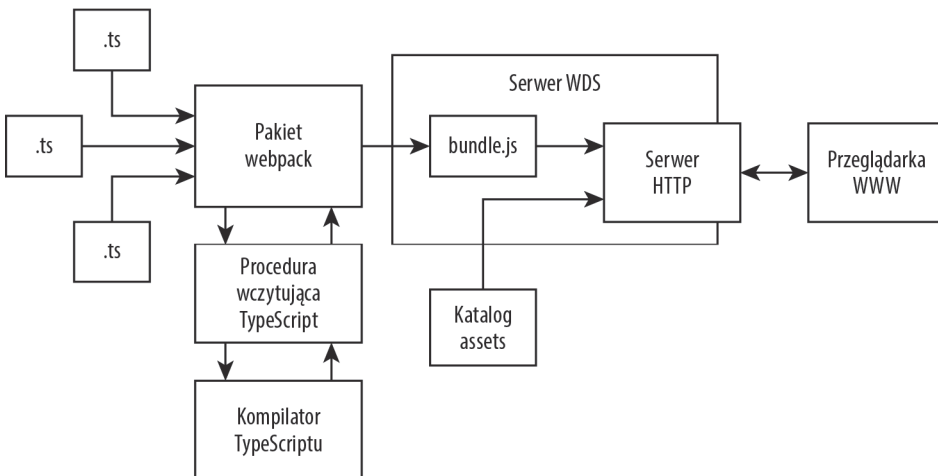
Listing 15.14. Dodawanie nowego polecenia w kodzie pliku *index.ts* w katalogu *src*

```
console.log("Aplikacja internetowa");
console.log("To jest nowe polecenie");
```

Po zapisaniu pliku *index.ts* pakiet webpack powoduje wygenerowanie nowej paczki, a do przeglądarki WWW wysyłany jest sygnał nakazujący odświeżenie strony internetowej — w konsoli narzędzi programistycznych w przeglądarce WWW zostaną wyświetlone następujące dane wyjściowe:

```
Aplikacja internetowa
To jest nowe polecenie
```

Dodanie serwera WDS powoduje rozszerzenie łańcucha zestawu narzędzi programistycznych i łączy aplikację ze środowiskiem uruchomieniowym JavaScriptu dostarczanym przez przeglądarkę WWW, jak pokazałem na rysunku 15.4.



Rysunek 15.4. Dodawanie serwera WDS do zestawu narzędzi programistycznych

Zestaw narzędzi programistycznych zawiera kluczowe elementy dostępne w większości projektów aplikacji internetowych, choć poszczególne części zwykle pozostają w ukryciu. Zwróć uwagę na to, że kompilator TypeScriptu to zwykle tylko jeden element łańcucha pozwalający na zintegrowanie kodu TypeScriptu z szerokim zestawem narzędzi programistycznych JavaScriptu.

Tworzenie modelu danych

Aplikacja będzie otrzymywała listę produktów za pomocą usługi sieciowej i żądania HTTP. Użytkownik będzie miał możliwość wybierania produktów i złożenia zamówienia, które zostanie przekazane do usługi sieciowej za pomocą kolejnego żądania HTTP. Aby przygotować model danych, w katalogu *src* utwórz podkatalog *data* i umieść w nim plik o nazwie *entities.ts* z kodem przedstawionym na listingu 15.15.

Listing 15.15. Zawartość pliku *entities.ts* w katalogu *src/data*

```
export type Product = {
  id: number,
  name: string,
  description: string,
  category: string,
  price: number
};

export class OrderLine {
  constructor(public product: Product, public quantity: number) {
    // Polecenia nie są wymagane.
  }

  get total(): number {
    return this.product.price * this.quantity;
  }
}

export class Order {
  private lines = new Map<number, OrderLine>();

  constructor(initialLines?: OrderLine[]) {
    if (initialLines) {
      initialLines.forEach(ol => this.lines.set(ol.product.id, ol));
    }
  }

  public addProduct(prod: Product, quantity: number) {
    if (this.lines.has(prod.id)) {
      if (quantity === 0) {
        this.removeProduct(prod.id);
      } else {
        this.lines.get(prod.id)!.quantity += quantity;
      }
    } else {
      this.lines.set(prod.id, new OrderLine(prod, quantity));
    }
  }

  public removeProduct(id: number) {
    this.lines.delete(id);
  }
}
```



```

get orderLines(): OrderLine[] {
    return [...this.lines.values()];
}

get productCount(): number {
    return [...this.lines.values()]
        .reduce((total, ol) => total += ol.quantity, 0);
}

get total(): number {
    return [...this.lines.values()].reduce((total, ol) => total += ol.total, 0);
}
}

```

Typy `Product`, `Order` i `OrderLine` są eksportowane, więc mogą być używane poza plikiem kodu źródłowego. Klasa `Order` przedstawia produkty wybrane przez użytkownika, a każdy z nich jest wyrażony jako obiekt typu `OrderLine` łączący produkt i liczbę jego egzemplarzy. `Product` zdefiniowano jako alias typu, ponieważ ułatwi to pracę z danymi pobieranymi zdalnie, gdy do projektu zostanie dodana usługa sieciowa w dalszej części rozdziału. Typy `Order` i `OrderLine` zostały zdefiniowane jako klasy, ponieważ poza kolekcją powiązanych ze sobą właściwości zawierają także funkcje dodatkowe.

Tworzenie źródła danych

Usługa sieciowa będzie dodana do projektu w dalszej części rozdziału. W tym momencie zajmiesz się utworzeniem klasy zapewniającej dostęp do pewnych lokalnych danych testowych. Aby ułatwić przejście od danych lokalnych do zdalnych, zdefiniowana zostanie klasa abstrakcyjna zapewniająca dostęp do funkcji podstawowych i tworząca konkretne implementacje poszczególnych źródeł danych. W katalogu `src/data` umieść plik `abstractDataSource.ts` i wykorzystaj go do implementacji klasy przedstawionej na listingu 15.16.

Listing 15.16. Zawartość pliku `abstractDataSource.ts` w katalogu `src/data`

```

import { Product, Order } from "../entities";

export type ProductProp = keyof Product;

export abstract class AbstractDataSource {
    private _products: Product[];
    private _categories: Set<string>;
    public order: Order;
    public loading: Promise<void>;

    constructor() {
        this._products = [];
        this._categories = new Set<string>();
        this.order = new Order();
        this.loading = this.getData();
    }

    async getProducts(sortProp: ProductProp = "id",

```

```

        category?: string): Promise<Product[]> {
    await this.loading;
    return this.selectProducts(this._products, sortProp, category);
  }

  protected async getData(): Promise<void> {
    this._products = [];
    this._categories.clear();
    const rawData = await this.loadProducts();
    rawData.forEach(p => {
      this._products.push(p);
      this._categories.add(p.category);
    });
  }

  protected selectProducts(prods: Product[],
    sortProp: ProductProp, category?: string): Product[] {
    return prods.filter(p=> category === undefined || p.category === category)
      .sort((p1, p2) => p1[sortProp] < p2[sortProp]
        ? -1 : p1[sortProp] > p2[sortProp] ? 1: 0);
  }

  async getCategories(): Promise<string[]> {
    await this.loading;
    return [...this._categories.values()];
  }

  protected abstract loadProducts(): Promise<Product[]>;
  abstract storeOrder(): Promise<number>;
}

```

Klasa `AbstractDataSource` używa funkcjonalności obietnic w JavaScriptcie do pobierania danych w tle oraz korzysta ze słów kluczowych `async` i `await` do wyrażenia zależnego od tych operacji. Klasa przedstawiona na listingu 15.16 wywołuje w konstruktorze metodę abstrakcyjną `loadProducts()` oraz metody `getProducts()` i `getCategories()` oczekujące na zakończenie operacji pobierania danych w tle, a dopiero następnie generujących dane wyjściowe. Aby utworzyć implementację klasy źródła danych wykorzystującej lokalne dane testowe, w katalogu `src/data` umieść plik `localDataSource.ts` z zawartością przedstawioną na listingu 15.17.

Listing 15.17. Zawartość pliku `localDataSource.ts` w katalogu `src/data`

```

import { AbstractDataSource } from "../abstractDataSource";
import { Product } from "../entities";

export class LocalDataSource extends AbstractDataSource {

  loadProducts(): Promise<Product[]> {
    return Promise.resolve([
      { id: 1, name: "P1", category: "Sporty wodne",
        description: "P1 (Sporty wodne)", price: 3 },
      { id: 2, name: "P2", category: "Sporty wodne",
        description: "P2 (Sporty wodne)", price: 4 },
      { id: 3, name: "P3", category: "Bieganie",
        description: "P3 (Bieganie)", price: 5 },
    ]);
  }
}

```

```

        { id: 4, name: "P4", category: "Szachy",
          description: "P4 (Szachy)", price: 6 },
        { id: 5, name: "P5", category: "Szachy",
          description: "P6 (Szachy)", price: 7 },
      ]);
    }

    storeOrder(): Promise<number> {
      console.log("Złożenie zamówienia");
      console.log(JSON.stringify(this.order));
      return Promise.resolve(1);
    }
  }
}

```

Ta klasa używa metody `Promise.resolve()` do utworzenia obiektu `Promise`, który natychmiast wygeneruje odpowiedź i pozwoli na łatwe wykorzystanie danych testowych. W rozdziale 16. poznasz źródło danych faktycznie przeprowadzające operacje w tle w celu pobrania danych z usługi sieciowej. Aby upewnić się o działaniu podstawowej funkcjonalności modelu danych, należy zastąpić kod w pliku `index.ts` kodem przedstawionym na listingu 15.18.

Listing 15.18. Nowa zawartość pliku `index.ts` w katalogu `src`

```

import { LocalDataSource } from "../data/localDataSource";

async function displayData(): Promise<string> {
  let ds = new LocalDataSource();
  let allProducts = await ds.getProducts("name");
  let categories = await ds.getCategories();
  let chessProducts = await ds.getProducts("name", "Szachy");

  let result = "";

  allProducts.forEach(p => result += `Produkt: ${p.name}, ${p.category}\n`);
  categories.forEach(c => result += (`Kategoria: ${c}\n`));
  chessProducts.forEach(p => ds.order.addProduct(p, 1));
  result += `Wartość całkowita zamówienia: ${ds.order.total.toFixed(2)} zł`;
  return result;
}

displayData().then(res => console.log(res));

```

Po wprowadzeniu zmian i zapisaniu pliku `index.ts` kod zostanie skompilowany, a łańcuch poleceń `import` rozwiązany w taki sposób, aby paczka wygenerowana przez pakiet `webpack` zawierała cały kod JavaScriptu wymagany przez aplikację. Przeglądarka WWW otrzyma sygnał do odświeżenia strony internetowej, a w konsoli JavaScriptu w przeglądarce zostaną wygenerowane następujące dane wyjściowe:

```

Produkt: P1, Sporty wodne
Produkt: P2, Sporty wodne
Produkt: P3, Bieganie
Produkt: P4, Szachy
Produkt: P5, Szachy
Kategoria: Sporty wodne

```

Kategoria: Bieganie

Kategoria: Szachy

Wartość całkowita zamówienia: 13.00 zł

Generowanie treści HTML-a za pomocą API modelu DOM

Niewielu użytkowników chce zaglądać do okna konsoli JavaScriptu przeglądarki WWW, aby tam zobaczyć dane wyjściowe. Przeglądarka WWW oferuje API modelu DOM pozwalający aplikacjom internetowym na współdziałanie z dokumentem HTML-a wyświetlanym użytkownikowi, dynamiczne generowanie treści, a także reagowanie na działania podejmowane przez użytkownika. W celu utworzenia klasy generującej element HTML-a należy dodać do katalogu *src* plik o nazwie *domDisplay.ts* i użyć go do zdefiniowania klasy przedstawionej na listingu 15.19.

Listing 15.19. Zawartość pliku *domDisplay.ts* w katalogu *src*

```
import { Product, Order } from "../data/entities";

export class DomDisplay {

  props: {
    products: Product[],
    order: Order
  }

  getContent(): HTMLElement {
    let elem = document.createElement("h3");
    elem.innerText = this.getElementText();
    elem.classList.add("bg-primary", "text-center", "text-white", "p-2");
    return elem;
  }

  getElementText() {
    return `Liczba produktów: ${this.props.products.length}, `
      + `Wartość całkowita zamówienia: ${ this.props.order.total } zł`;
  }
}
```

Klasa *DomDisplay* definiuje metodę *getContent()*, wynikiem działania której jest obiekt *HTMLElement* typu używanego przez API modelu DOM do przedstawienia elementu HTML-a. Metoda *getContent()* tworzy element `<h3>` i wykorzystuje szablon do zdefiniowania wartości tego elementu. Element jest dodawany do czterech klas, które następnie są używane do zarządzania wyglądem elementu podczas jego wyświetlania. Wartości danych używane w ciągu tekstowym szablonu są dostarczane za pomocą właściwości o nazwie *props*. Jest to konwencja zaadaptowana z frameworka React, którą dokładnie omówię w rozdziale 19.

Dodawanie obsługi stylów Bootstrap CSS

Trzy klasy, do których na listingu 15.19 zostały dołączone elementy `<h3>`, odpowiadają stylom zdefiniowanym przez Bootstrap, czyli wysokiej jakości framework CSS typu *open source*, niezwykle ułatwiający spójne stosowanie stylów w treści HTML-a.

Konfiguracja pakietu `webpack` może zostać za pomocą procedur wczytujących rozszerzona o obsługę dodatkowych typów treści umieszczanych w pliku paczki. Oznacza to możliwość rozbudowy zestawu narzędzi programistycznych o obsługę arkuszy stylów CSS, np. definiujących style Bootstrap zastosowane dla elementu `<h3>`.

Zatrzymaj proces serwera WDS przez naciśnięcie klawiszy `Ctrl+C`, a następnie z poziomu katalogu `webapp` w powłoce wydaj polecenia przedstawione na listingu 15.20, aby w ten sposób zainstalować niezbędne komponenty.

■ **Uwaga** W większości moich projektów używam frameworka Bootstrap CSS, ponieważ praca z nim należy do łatwych, a otrzymane wyniki są dobrej jakości. Więcej informacji na temat dostępnych stylów i opcjonalnych funkcji JavaScriptu w tym frameworku znajdziesz w witrynie <https://getbootstrap.com/>.

Listing 15.20. Dodawanie pakietów do projektu

```
$ npm install bootstrap@4.6.0
$ npm install --save-dev css-loader@5.0.1
$ npm install --save-dev style-loader@2.0.0
```

Pakiet `bootstrap` zawiera style CSS, które chciałbym zastosować w przykładowym projekcie. Z kolei pakiety `css-loader` i `style-loader` zawierają procedury wczytujące przeznaczone do obsługi stylów CSS (oba wymienione pakiety są ważne podczas dołączania stylów CSS do pliku paczki tworzonej przez pakiet `webpack`). W konfiguracji `webpacka` wprowadź zmiany przedstawione na listingu 15.21 i tym samym dodaj obsługę CSS do pliku paczki.

Listing 15.21. Dodawanie procedur wczytujących do pliku `webpack.config.js` w katalogu `webapp`

```
module.exports = {
  mode: "development",
  devtool: "inline-source-map",
  entry: "./src/index.ts",
  output: { filename: "bundle.js" },
  resolve: { extensions: [".ts", ".js", ".css"] },
  module: {
    rules: [
      { test: /\.ts$/, use: "ts-loader", exclude: /node_modules/ },
      { test: /\.css$/, use: ["style-loader", "css-loader"] },
    ]
  },
},
devServer: {
  contentBase: "./assets",
  port: 4500
}
};
```

Na listingu 15.22 przedstawiłem zmodyfikowaną wersję kodu pliku *index.ts* deklarującego zależność od stylów CSS pochodzących z pakietu Bootstrap oraz użycie klasy *DomHeader* do wygenerowania treści HTML-a w przeglądarce WWW.

Listing 15.22. Wyświetlanie zawartości HTML-a przez kod zdefiniowany w pliku *index.ts* w katalogu *src*

```
import { LocalDataSource } from "../data/localDataSource";
import { DomDisplay } from "../domDisplay";
import "bootstrap/dist/css/bootstrap.css";

let ds = new LocalDataSource();

async function displayData(): Promise<HTMLElement> {
    let display = new DomDisplay();
    display.props = {
        products: await ds.getProducts("name"),
        order: ds.order
    }
    return display.getContent();
}

document.onreadystatechange = () => {
    if (document.readyState === "complete") {
        displayData().then(elem => {
            let rootElement = document.getElementById("app");
            rootElement.innerHTML = "";
            rootElement.appendChild(elem);
        });
    }
};
```

API modelu DOM oferuje pełny zestaw funkcjonalności przeznaczonych do pracy z dokumentem HTML-a wyświetlanym przez przeglądarkę WWW. Jednak wynikiem może być dość rozległy i trudny w odczycie kod, zwłaszcza gdy treść przeznaczona do wyświetlenia zależy od wyniku działania zadania w tle, np. pobrania danych z usługi sieciowej.

Kod przedstawiony na listingu 15.22 musi poczekać na zakończenie dwóch zadań, zanim będzie mógł wyświetlić jakąkolwiek treść. Przeglądarka WWW musi zakończyć przetwarzanie dokumentu HTML-a zdefiniowanego w pliku *index.html*, zanim API modelu DOM będzie mógł zostać użyty do przetwarzania zawartości dokumentu. Przeglądarka przetwarza elementy HTML-a w kolejności ich zdefiniowania w dokumencie HTML-a, co oznacza, że kod JavaScriptu będzie wykonany, zanim przeglądarka rozpocznie przetwarzanie elementów w sekcji *<body>* dokumentu. Każda próba modyfikacji dokumentu przed jego pełnym przetworzeniem będzie prowadziła do wygenerowania niespójnych wyników.

■ **Wskazówka** Ustawienia domyślne kompilatora TypeScriptu obejmują pliki deklaracji typów dla API modelu DOM, co pozwala na bezpieczne dla typów używanie funkcjonalności oferowanej przez przeglądarkę WWW.

Kod przedstawiony na listingu 15.22 musi również zaczekać na pobranie danych przez źródło danych. Wprawdzie w omawianym przykładzie klasa `LocalDataSource` używa dostępnych natychmiast lokalnych danych testowych, ale pobieranie danych z usługi sieciowej może się wiązać z pewnym opóźnieniem. Przykładową implementację usługi sieciowej przedstawię w rozdziale 16.

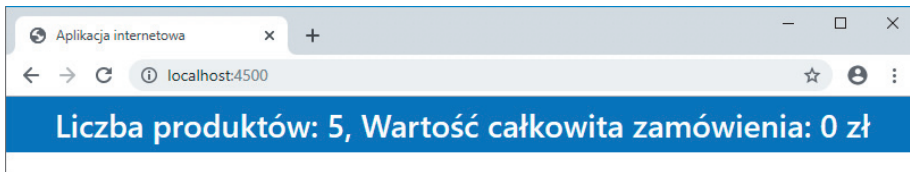
Gdy oba zadania zostaną wykonane, element miejsca zarezerwowanego w pliku `index.html` zostanie usunięty i zastąpiony obiektem `HTMLElement` otrzymanym przez utworzenie obiektu `DomDisplay` i wywołanie jego metody `getContent()`.

Zapisz zmiany w pliku `index.ts` i z poziomu katalogu `webapp` w powłoce wydaj polecenie przedstawione na listingu 15.23, aby uruchomić serwer WDS na podstawie konfiguracji utworzonej na listingu 15.21.

Listing 15.23. *Uruchamianie narzędzi programistycznych*

```
$ npx webpack serve
```

Utworzona zostanie nowa paczka zawierająca style CSS. W przeglądarce WWW przejdź pod adres `http://localhost:4500` — zobaczysz wyświetloną treść HTML-a, dla której zostały zastosowane style Bootstrap CSS, jak pokazałem na rysunku 15.5.



Rysunek 15.5. Wygenerowane elementy HTML-a

-
- **Wskazówka** Dołączone do projektu procedury wczytujące zajmują się obsługą CSS przez dodanie kodu JavaScriptu wykonywanego podczas przetwarzania pliku paczki. API dostarczany przez przeglądarkę WWW wykorzystuje wymieniony kod do tworzenia stylów CSS. Takie podejście oznacza, że plik paczki zawiera jedynie kod JavaScriptu, pomimo dostarczania klientowi różnych typów treści.
-

Używanie formatu JSX do tworzenia treści HTML-a

Wyrażanie elementów HTML-a za pomocą poleceń JavaScriptu jest niewygodne, a bezpośrednie używanie API modelu DOM powoduje wygenerowanie rozwlekłego kodu, który jest trudny do zrozumienia i podatny na błędy, nawet pomimo oferowanej przez język TypeScript obsługi typów statycznych.

Problem nie leży jedynie w samym API modelu DOM — choć nie zawsze był on tworzony z myślą o jak najłatwiejszym sposobie stosowania — ale w trudności związanej z używaniem poleceń odpowiedzialnych za generowanie treści deklaratywnej, takiej jak elementy HTML-a. Znacznie bardziej eleganckie podejście polega na wykorzystaniu formatu JSX, czyli *JavaScript XML*, co pozwala na łatwe łączenie treści deklaratywnej, takiej jak elementy HTML-a, z poleceniami kodu.

JSX ma ścisły związek z programowaniem React — o czym się przekonasz w rozdziale 19. — przy czym kompilator TypeScriptu dostarcza funkcje mogące być używane w dowolnym projekcie.

■ **Uwaga** JSX to nie jedyny sposób na ułatwienie pracy z elementami HTML-a — zdecydowałem się na zastosowanie tego rozwiązania w tym rozdziale, ponieważ jest obsługiwane przez kompilator TypeScriptu. Jeżeli nie lubisz JSX, możesz skorzystać z jednego z wielu dostępnych pakietów szablonów (np. zacznij od wyszukania *szablonów mustache*).

Najlepszym sposobem na zrozumienie JSX jest rozpoczęcie tworzenia kodu w tym formacie. Pliki TypeScriptu zawierają treść JSX zdefiniowaną w plikach z rozszerzeniem *.tsx*, odzwierciedlając w ten sposób połączenie funkcjonalności TypeScriptu i JSX. Do katalogu *src* dodaj plik o nazwie *htmlDisplay.tsx* i zawartości przedstawionej na listingu 15.24.

Listing 15.24. Zawartość pliku *htmlDisplay.tsx* w katalogu *src*

```
import { Product, Order } from "../data/entities";

export class HtmlDisplay {
  props: {
    products: Product[],
    order: Order
  }

  getContent(): HTMLElement {
    return <h3 className="bg-secondary text-center text-white p-2">
      { this.getElementText() }
    </h3>
  }

  getElementText() {
    return `Liczba produktów: ${this.props.products.length}, `
      + `Wartość całkowita zamówienia: ${ this.props.order.total }`;
  }
}
```

Ten plik używa formatu JSX do wygenerowania takiego samego wyniku jak w przypadku użycia zwykłej klasy TypeScriptu. Różnica wiąże się z metodą `getContent()`, która zwraca element HTML-a wyrażony bezpośrednio jako element, zamiast wykorzystać API modelu DOM do utworzenia obiektu i jego skonfigurowania za pomocą właściwości tego obiektu. Zwrócony element `<h3>` jest wyrażony w sposób podobny do stosowanego w dokumencie HTML-a, przy czym tutaj mamy dodatkowe fragmenty kodu JavaScriptu pozwalające wyrażeniom na dynamiczne generowanie treści na podstawie wartości dostarczanych przez właściwość `props`.

Ten plik nie zostanie skompilowany, ponieważ projekt nie został jeszcze skonfigurowany do obsługi JSX. Mimo to możesz zobaczyć, że ten format pozwala tworzyć treści w sposób znacznie bardziej naturalny. W kolejnych sekcjach wyjaśnię, jak są przetwarzane pliki JSX, i pokażę sposób konfiguracji przykładowego projektu w celu ich obsługi.

Sposób działania JSX

Podczas kompilacji pliku TypeScriptu JSX kompilator przetwarza znajdujące się w nim elementy HTML-a w celu ich konwersji na polecenia JavaScriptu. Każdy element jest przetwarzany jako oddzielny znacznik definiujący typ elementu, atrybuty stosowane w danym elemencie oraz właściwą treść tego elementu.

Poszczególne elementy HTML-a są zastępowane wywołaniem funkcji nazywanej *funkcją fabryki*, która będzie odpowiedzialna za utworzenie treści HTML-a w trakcie działania aplikacji. Zgodnie z konwencją ta funkcja fabryki nazywa się `createElement()`, ponieważ taka nazwa jest stosowana we frameworku React. Oznacza to, że klasa przedstawiona na listingu 15.24 zostanie skonwertowana na następującą postać:

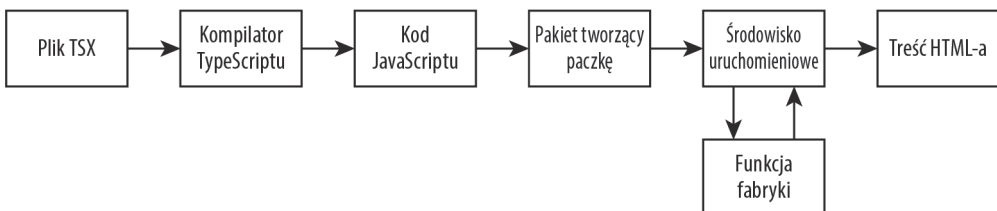
```
...
import { Product, Order } from "../data/entities";

export class HtmlDisplay {
  props: {
    products: Product[],
    order: Order
  }

  getContent() {
    return createElement("h3",
      { className: "bg-secondary text-center text-white p-2" },
      this.getElementText());
  }

  getElementText() {
    return `${this.props.products.length} Products, `
      + `Wartość całkowita zamówienia: ${ this.props.order.total }`;
  }
}
...
```

Poza nazwą kompilator nie ma żadnych informacji o funkcji fabryki. W wyniku operacji konwersji treść HTML-a zostaje zastąpiona przez polecenia kodu, które następnie mogą być w zwykły sposób skompilowane i wykonane przez standardowe środowisko uruchomieniowe JavaScriptu, jak pokazałem na rysunku 15.6.



Rysunek 15.6. Przekształcanie pliku w formacie JSX

Po uruchomieniu aplikacji każde wywołanie funkcji fabryki jest odpowiedzialne za użycie nazwy znacznika, atrybutu i przetworzenie treści przez kompilator w celu utworzenia elementu HTML-a wymaganego przez aplikację.

Właściwość props kontra atrybuty

Elementy w pliku JSX nie są standardowym kodem HTML-a. Podstawowa różnica polega na tym, że atrybuty w elementach używają nazw właściwości JavaScriptu zdefiniowanych przez API modelu DOM zamiast odpowiadających im nazw atrybutów ze specyfikacji HTML-a. Wiele właściwości i atrybutów współdzieli tę samą nazwę, choć istnieją również spore różnice. Największe zamieszanie powoduje atrybut `class`, który jest używany w celu przypisania elementom jednej lub więcej klas, najczęściej w celu nadania stylu tym elementom.

API modelu DOM nie może używać słowa kluczowego `class`, ponieważ w języku JavaScript jest to słowo zarezerwowane, więc klasa jest przypisywana elementowi za pomocą właściwości `className`, jak pokazałem w kolejnym fragmencie kodu:

```
...
<h3 className="bg-secondary text-center text-white p-2">
...
```

Dlatego te klasy TypeScriptu JSX otrzymują wartości danych za pomocą właściwości o nazwie `props`, ponieważ każda właściwość musi być zdefiniowana w obiekcie `HTMLElement` utworzonym przez funkcję fabryki. Pominięcie nazw właściwości w pliku JSX to często popełniany błąd i jednocześnie odpowiednie miejsce do rozpoczęcia sprawdzania kodu w przypadku otrzymania wyników innych niż oczekiwane.

Konfigurowanie kompilatora TypeScriptu i procedury wczytującej pakiet webpack

Kompilator TypeScriptu domyślnie nie przetwarza plików TSX i wymaga dwóch ustawień konfiguracyjnych, które wymieniłem w tabeli 15.2. Wprawdzie mamy jeszcze inne opcje kompilatora dotyczące obsługi JSX, ale dwie wymienione tutaj są wymagane do rozpoczęcia pracy z plikami w formacie JSX.

Tabela 15.2. Ustawienia kompilatora niezbędne do zapewnienia obsługi JSX

Opcja	Opis
<code>jsx</code>	Ta opcja określa sposób przetwarzania elementów HTML-a w pliku TSX. Wartość <code>react</code> zastępuje elementy HTML-a wywołaniami funkcji fabryki i generuje plik JavaScriptu. Wartość <code>react-native</code> powoduje wygenerowanie pliku JavaScriptu i pozostawienie nietkniętych elementów HTML-a. Wartość <code>preserve</code> generuje plik JSX pozostawiający nietknięte elementy HTML-a. Wartość <code>react-jsx</code> używa <code>__jsx</code> jako nazwy funkcji tworzącej elementy
<code>jsxFactory</code>	Ta opcja określa nazwę funkcji fabryki, która jest używana, gdy wartością opcji <code>jsx</code> kompilatora jest <code>react</code>

Na potrzeby przykładowego projektu zamierzam zdefiniować funkcję fabryki o nazwie `createElement()` i wybrać opcję `react` dla ustawienia `jsx`, aby kompilator zastępował treść HTML-a wywołaniami funkcji fabryki, jak pokazałem na listingu 15.25.

Listing 15.25. Konfigurowanie kompilatora w pliku `tsconfig.json` w katalogu `webapp`

```
{
  "compilerOptions": {
    "target": "es2020",
    "outDir": "./dist",
    "rootDir": "./src",
    "jsx": "react",
    "jsxFactory": "createElement"
  }
}
```

Konfiguracja pakietu `webpack` musi być uaktualniona, aby pliki TSX były uwzględniane podczas tworzenia paczki. Zmiany konieczne do wprowadzenia przedstawiłem na listingu 15.26.

Listing 15.26. Konfigurowanie pakietu `webpack` w pliku `webpack.conf.js` w katalogu `webapp`

```
module.exports = {
  mode: "development",
  devtool: "inline-source-map",
  entry: "./src/index.ts",
  output: { filename: "bundle.js" },
  resolve: { extensions: [".ts", ".tsx", ".js", ".css"] },
  module: {
    rules: [
      { test: /\.tsx?$/, use: "ts-loader", exclude: /node_modules/ },
      { test: /\.css$/, use: ["style-loader", "css-loader"] },
    ]
  },
  devServer: {
    contentBase: "./assets",
    port: 4500
  }
};
```

Zmiana ustawienia na `resolve` wskazuje pakietowi `webpack`, że pliki TSX powinny być uwzględniane podczas tworzenia pliku paczki. Natomiast pozostałe zmiany wskazują na obsługę plików TSX przez pakiet `ts-loader`, który będzie używany przez kompilator TypeScriptu.

Tworzenie funkcji fabryki

Kod wygenerowany przez kompilator zastępuje treść HTML-a wywołaniami funkcji fabryki, co pozwala na konwersję kodu JSX na standardowy kod JavaScriptu. Implementacja funkcji fabryki zależy od środowiska, w którym będzie uruchamiana aplikacja, dlatego też np. aplikacja `React` będzie wykorzystywała funkcję fabryki generującą treść możliwą do zarządzania przez `React`. W omawianym przykładzie zamierzam utworzyć funkcję fabryki używającą po prostu API modelu DOM do utworzenia obiektu `HTMLElement`. Oczywiście nie jest to rozwiązanie tak

eleganckie lub efektywne jak dynamiczna obsługa treści przez React lub inny framework, ale pozwala mi na użycie JSX w aplikacji bez konieczności zagłębiania się w szczegóły. W celu zdefiniowania funkcji fabryki w katalogu `src` utwórz podkatalog `tools` i umieść w nim plik o nazwie `jsxFactory.ts` z kodem przedstawionym na listingu 15.27.

Listing 15.27. Zawartość pliku `jsxFactory.ts` w katalogu `src/tools`

```
export function createElement(tag: any, props: Object, ...children : Object[])
    : HTMLElement {

    function addChild(elem: HTMLElement, child: any) {
        elem.appendChild(child instanceof Node ? child
            : document.createTextNode(child.toString()));
    }

    if (typeof tag === "function") {
        return Object.assign(new tag(), { props: props || {} }).getContent();
    }

    const elem = Object.assign(document.createElement(tag), props || {});
    children.forEach(child => Array.isArray(child)
        ? child.forEach(c => addChild(elem, c)) : addChild(elem, child));
    return elem;
}

declare global {
    namespace JSX {
        interface ElementAttributesProperty { props; }
    }
}
```

Funkcja `createElement()` zdefiniowana na listingu 15.27 ma minimalną postać niezbędną do utworzenia elementów HTML-a za pomocą API modelu DOM bez konieczności stosowania jakichkolwiek zaawansowanych funkcji oferowanych przez frameworki wykorzystane w dalszej części książki. Parametr `tag` może być funkcją — w takim przypadku inna klasa używająca JSX zostanie podana jako typ elementu.

■ **Wskazówka** Ostatnia sekcja kodu na listingu 15.27 wskazuje kompilatorowi TypeScriptu, że powinien wykorzystać właściwość `props` do przeprowadzania operacji sprawdzania typu wartości przypisywanych atrybutom elementu JSX w plikach TSX. Opiera się to na funkcjonalności przestrzeni nazw TypeScriptu, czym nie będę się zajmował w tym rozdziale, ponieważ przestrzenie nazw zostały zastąpione przez wprowadzenie standardowych modułów JavaScriptu i nie zaleca się ich dalszego używania.

Używanie klasy JSX

Klasa JSX jest konwertowana na standardowy kod JavaScriptu, co oznacza, że może być używana w dokładnie taki sam sposób jak każda inna klasa TypeScriptu. W kodzie przedstawionym na listingu 15.28 usunąłem zależność od klasy API modelu DOM i zastąpiłem ją klasą JSX.

Listing 15.28. *Używanie klasy JSX w pliku index.ts*

```
import { LocalDataSource } from "../data/localDataSource";
import { HtmlDisplay } from "../htmlDisplay";
import "bootstrap/dist/css/bootstrap.css";

let ds = new LocalDataSource();

async function displayData(): Promise<HTMLElement> {
  let display = new HtmlDisplay();
  display.props = {
    products: await ds.getProducts("name"),
    order: ds.order
  }
  return display.getContent();
}

document.onreadystatechange = () => {
  if (document.readyState === "complete") {
    displayData().then(elem => {
      let rootElement = document.getElementById("app");
      rootElement.innerHTML = "";
      rootElement.appendChild(elem);
    });
  }
};
```

Klasa JSX jest bezpośrednim zamiennikiem dla klasy wykorzystującej API modelu DOM. W dalszej części rozdziału zobaczysz, jak łączyć klasy używające JSX z zastosowaniem jedynie elementów. Mimo to zawsze istnieje pewna granica między klasą zwykłą i zawierającą elementy HTML-a. W omawianym tutaj przykładzie granica leży między plikiem *index* a klasą *HtmlDisplay*.

Importowanie funkcji fabryki w klasie JSX

Ostatnia zmiana pozwalająca na dokończenie konfiguracji JSX to dodanie polecenia `import` do funkcji fabryki w klasie JSX, jak pokazałem na listingu 15.29. Kompilator przeprowadza konwersję elementów HTML-a na wywołania funkcji fabryki, natomiast polecenie `import` jest wymagane do kompilacji skonwertowanego kodu.

Listing 15.29. *Dodawanie polecenia import w kodzie pliku htmlDisplay.tsx w katalogu src*

```
import { createElement } from "../tools/jsxFactory";
import { Product, Order } from "../data/entities";

export class HtmlDisplay {

  props: {
    products: Product[],
    order: Order
  }
}
```

```

    getContent(): HTMLElement {
      return <h3 className="bg-secondary text-center text-white p-2">
        { this.getElementText() }
      </h3>
    }

    getElementText() {
      return `${this.props.products.length} Products, `
        + `Wartość całkowita zamówienia: ${ this.props.order.total }`;
    }
  }

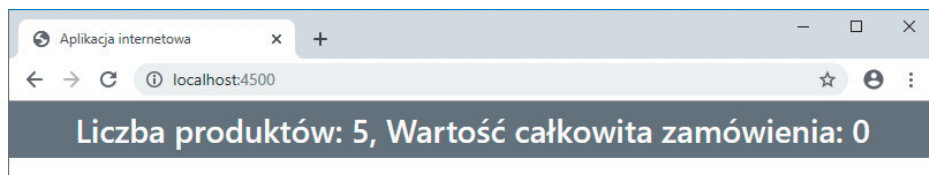
```

Polecenie `import` dla funkcji fabryki jest wymagane we wszystkich plikach TSX. Naciskając klawisze `Ctrl+C`, zatrzymaj narzędzia programistyczne webpacka, a następnie z poziomu katalogu *webapp* w powłoce wykonaj polecenie przedstawione na listingu 15.30, aby w ten sposób ponownie uruchomić te narzędzia z nową konfiguracją.

Listing 15.30. Uruchamianie narzędzi programistycznych

```
$ npx webpack serve
```

Po ponownym utworzeniu pliku paczki w przeglądarce WWW przejdź pod adres `http://localhost:4500` — zobaczysz wyświetloną treść HTML-a, używającą innych stylów koloru niż w poprzednim przykładzie, jak pokazałem na rysunku 15.7.



Rysunek 15.7. Generowanie treści z użyciem JSX

Dodawanie funkcjonalności do aplikacji

Skoro podstawowa struktura aplikacji znajduje się na swoim miejscu, można przystąpić do dodawania niezbędnych funkcjonalności. Zacznę od wyświetlenia produktów filtrowanych według kategorii.

Wyświetlanie filtrowanej listy produktów

W katalogu *src* umieść plik o nazwie *productItem.tsx* z kodem przedstawionym na listingu 15.31, który odpowiada za wyświetlenie szczegółów pojedynczego produktu.

Listing 15.31. Zawartość pliku *productItem.tsx* w katalogu *src*

```

import { createElement } from "../tools/jsxFactory";
import { Product } from "../data/entities";

```

```

export class ProductItem {
  private quantity: number = 1;

  props: {
    product: Product,
    callback: (product: Product, quantity: number) => void
  }

  getContent(): HTMLElement {
    return <div className="card m-1 p-1 bg-light">
      <h4>
        { this.props.product.name }
        <span className="badge badge-pill badge-primary float-right">
          { this.props.product.price.toFixed(2) } zł
        </span>
      </h4>
      <div className="card-text bg-white p-1">
        { this.props.product.description }
        <button className="btn btn-success btn-sm float-right"
          onclick={ this.handleAddToCart } >
          Dodaj do koszyka
        </button>
        <select className="form-control-inline float-right m-1"
          onchange={ this.handleQuantityChange }>
          <option>1</option>
          <option>2</option>
          <option>3</option>
        </select>
      </div>
    </div>
  }

  handleQuantityChange = (ev: Event): void => {
    this.quantity = Number((ev.target as HTMLSelectElement).value);
  }

  handleAddToCart = (): void => {
    this.props.callback(this.props.product, this.quantity);
  }
}

```

Klasa `ProductItem` otrzymuje obiekt `Product` i funkcję wywołania zwrotnego za pomocą właściwości tego obiektu. Metoda `getContent()` generuje elementy HTML-a wyświetlające szczegółowe informacje dotyczące obiektu `Product` z elementem `<select>` pozwalającym na wybór liczby produktów i przyciskiem, którego kliknięcie powoduje dodanie produktów do zamówienia.

Elementy `<select>` i `<button>` zostały skonfigurowane z funkcjami obsługi zdarzeń wykorzystującymi właściwości `onchange` i `onclick`. Metoda obsługująca zdarzenia jest zdefiniowana za pomocą składni funkcji strzałki, jak pokazałem w kolejnym fragmencie kodu:

```

...
handleQuantityChange = (ev: Event): void => {
  this.quantity = Number((ev.target as HTMLSelectElement).value);
}
...

```

Składnia funkcji strzałki gwarantuje, że słowo kluczowe `this` odwołuje się do obiektu `ProductItem`, co pozwala na wykorzystanie właściwości `props` i `quantity`. Jeżeli do obsługi zdarzenia zostałaby zastosowana metoda konwencjonalna, wówczas `this` odwoła się do obiektu opisującego zdarzenie.

Deklaracje TypeScriptu dla obsługi zdarzeń API modelu DOM są niewygodne i wymagają asercji typu dla celu zdarzenia, zanim będzie można uzyskać dostęp do funkcjonalności:

```
...
handleQuantityChange = (ev: Event): void => {
  this.quantity = Number((ev.target as HTMLSelectElement).value);
}
...
```

Aby odczytać wartość właściwości `value` elementu `<select>`, konieczne jest użycie asercji i wskazanie kompilatorowi TypeScriptu, że właściwość `event.target` zwróci obiekt `HTMLSelectElement`.

■ **Wskazówka** Typ `HTMLSelectElement` to jeden ze standardowych typów API modelu DOM. Ten interfejs API został dokładnie omówiony na stronie <https://developer.mozilla.org/en-US/docs/Web/API/HTMLSelectElement>.

W celu wyświetlenia listy przycisków kategorii pozwalających użytkownikowi na filtrowanie treści dodaj do katalogu `src` plik o nazwie `categoryList.tsx` z kodem przedstawionym na listingu 15.32.

Listing 15.32. Zawartość pliku `categoryList.tsx` w katalogu `src`

```
import { createElement } from "../tools/jsxFactory";

export class CategoryList {
  props: {
    categories: string[];
    selectedCategory: string;
    callback: (selected: string) => void
  }

  getContent(): HTMLElement {
    return <div>
      { ["Wszystkie", ...this.props.categories].map(c =>
this.getCategoryButton(c))}
    </div>
  }

  getCategoryButton(cat?: string): HTMLElement {
    let selected = this.props.selectedCategory === undefined
      ? "Wszystkie": this.props.selectedCategory;
    let btnClass = selected === cat ? "btn-primary": "btn-secondary";
    return <button className={ `btn btn-block ${btnClass}` }
      onClick={ () => this.props.callback(cat)}>
      { cat }
    </button>
  }
}
```


Ta klasa wyświetla listę elementów `<button>` ze stylami klas Bootstrap. Właściwość `props` tej klasy dostarcza listę kategorii, dla których zostaną utworzone przyciski, obecnie wybraną kategorię oraz funkcję wywołania zwrotnego wywoływaną po kliknięciu przycisku przez użytkownika.

```
...
return <button className={ `btn btn-block ${btnClass}` }
  onClick={ () => this.props.callback(cat) }>
...

```

Taki wzorzec jest często spotykany podczas stosowania JSX — klasy generują treść HTML-a na podstawie danych otrzymanych za pomocą właściwości `props`. Wymieniona właściwość oferuje również funkcje wywołania zwrotnego wykonywane w odpowiedzi na zdarzenia. W omawianym przykładzie atrybut `onClick` został użyty do wywołania funkcji otrzymanej poprzez `callback`.

Aby wyświetlić listę produktów i przyciski kategorii, dodaj do katalogu `src` plik o nazwie *productList.tsx* zawierający kod przedstawiony na listingu 15.33.

Listing 15.33. Zawartość pliku *productList.tsx* w katalogu `src`

```
import { createElement } from "../tools/jsxFactory";
import { Product } from "../data/entities";
import { ProductItem } from "../productItem";
import { CategoryList } from "../categoryList";

export class ProductList {
  props: {
    products: Product[],
    categories: string[],
    selectedCategory: string,
    addToOrderCallback?: (product: Product, quantity: number) => void,
    filterCallback?: (category: string) => void;
  }

  getContent(): HTMLElement {
    return <div className="container-fluid">
      <div className="row">
        <div className="col-3 p-2">
          <CategoryList categories={ this.props.categories }
            selectedCategory={ this.props.selectedCategory }
            callback={ this.props.filterCallback } />
        </div>
        <div className="col-9 p-2">
          {
            this.props.products.map(p =>
              <ProductItem product={ p }
                callback={ this.props.addToOrderCallback } />
            )
          }
        </div>
      </div>
    </div>
  }
}
```

Metoda `getContent()` w tej klasie opiera swoje działanie na jednej z najbardziej użytecznych funkcjonalności JSX, czyli możliwości stosowania innych klas JSX jako elementów HTML-a, np.:

```
...
<div className="col-3 p-2">
  <CategoryList categories={ this.props.categories }
    selectedCategory={ this.props.selectedCategory }
    callback={ this.props.filterCallback } />
</div>
...
```

Podczas przetwarzania pliku TSX kompilator TypeScriptu wykrywa niestandardowe znaczniki i tworzy polecenie wywołujące funkcję fabryki z odpowiednią klasą. Podczas działania aplikacji następuje utworzenie nowego egzemplarza klasy, atrybuty elementu są przypisywane właściwości `props`, a metoda `getContent()` jest wywoływana w celu pobrania treści, która ma trafić do elementu HTML-a wyświetlanego użytkownikowi.

Wyświetlanie treści i obsługa uaktualnień

Konieczne jest przygotowanie pomostu między funkcjonalnością magazynu danych a klasami JSX wyświetlającymi treść użytkownikowi, aby zagwarantować uaktualnianie treści w celu odzwierciedlenia zmian stanu aplikacji. Frameworki zaprezentowane w dalszej części książki zajmują się obsługą efektywnych uaktualnień i minimalizują ilość pracy, którą przeglądarka WWW musi wykonać, by wyświetlić uaktualnioną treść.

W omawianym przykładzie zamierzam skorzystać z najprostszego podejścia, które polega na usunięciu i ponownym utworzeniu elementów HTML-a wyświetlanych przez przeglądarkę WWW, jak pokazałem na listingu 15.34. Ten kod zawiera zmodyfikowaną wersję klasy `HtmlDisplay`, otrzymującą źródło danych i zarządzającą stanem danych wymaganym do wyświetlenia listy produktów filtrowanych według kategorii.

Listing 15.34. Wyświetlanie zawartości zdefiniowanej w pliku `htmlDisplay.tsx` w katalogu `src`

```
import { createElement } from "../tools/jsxFactory";
import { Product, Order } from "../data/entities";
import { AbstractDataSource } from "../data/abstractDataSource";
import { ProductList } from "../productList";

export class HtmlDisplay {
  private containerElem: HTMLElement;
  private selectedCategory: string;

  constructor() {
    this.containerElem = document.createElement("div");
  }

  props: {
    dataSource: AbstractDataSource;
  }

  async getContent(): Promise<HTMLElement> {
```

```

    await this.updateContent();
    return this.containerElem;
  }

  async updateContent() {
    let products = await this.props.dataSource.getProducts("id",
      this.selectedCategory);
    let categories = await this.props.dataSource.getCategories();
    this.containerElem.innerHTML = "";
    let content = <div>
      <ProductList products={ products } categories={ categories }
        selectedCategory={ this.selectedCategory }
        addToOrderCallback={ this.addToOrder }
        filterCallback={ this.selectCategory } />
    </div>
    this.containerElem.appendChild(content);
  }

  addToOrder = (product: Product, quantity: number) => {
    this.props.dataSource.order.addProduct(product, quantity);
    this.updateContent();
  }

  selectCategory = (selected: string) => {
    this.selectedCategory = selected === "Wszystkie" ? undefined : selected;
    this.updateContent();
  }
}

```

Metody zdefiniowane przez klasę `HtmlDisplay` są używane jako funkcje wywołania zwrótnego dla klasy `ProductList`, która przekazuje je klasom `ProductItem` i `CategoryList`. Po wywołaniu te metody uaktualniają właściwości przechowujące informacje o stanie aplikacji, a następnie wywołują metodę `updateContent()` odpowiedzialną za zastąpienie treści HTML-a wygenerowanej przez klasę.

Aby dostarczyć klasę `HtmlDisplay` z wymaganymi właściwościami, uaktualnij plik `index.ts`, jak pokazałem na listingu 15.35.

Listing 15.35. Zmiana właściwości w kodzie pliku `index.ts` w katalogu `src`

```

import { LocalDataSource } from "../data/localDataSource";
import { HtmlDisplay } from "../htmlDisplay";
import "bootstrap/dist/css/bootstrap.css";

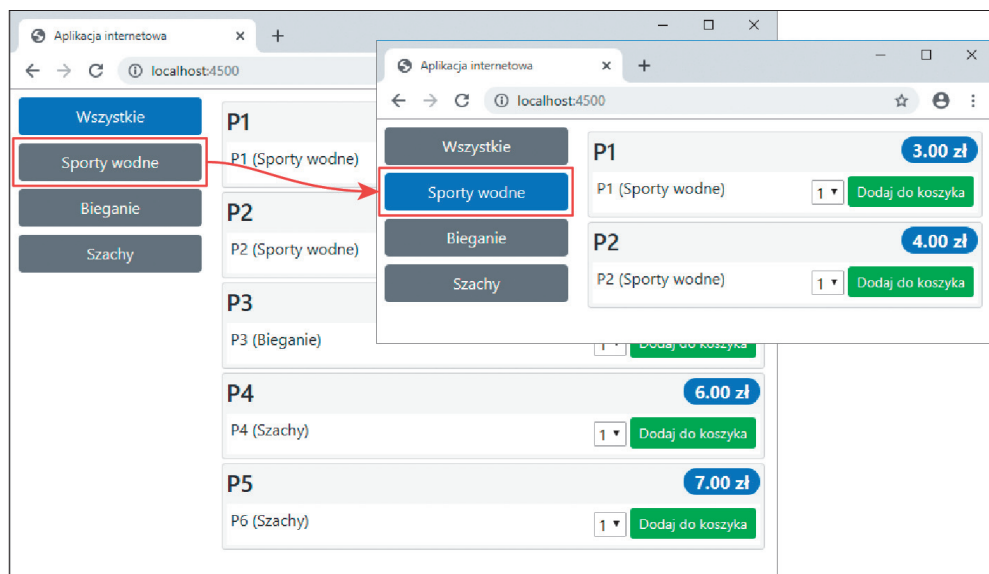
let ds = new LocalDataSource();

function displayData(): Promise<HTMLElement> {
  let display = new HtmlDisplay();
  display.props = {
    dataSource: ds
  }
  return display.getContent();
}

```

```
document.onreadystatechange = () => {
  if (document.readyState === "complete") {
    displayData().then(elem => {
      let rootElement = document.getElementById("app");
      rootElement.innerHTML = "";
      rootElement.appendChild(elem);
    });
  }
};
```

Po zapisaniu zmian nastąpi utworzenie nowego pliku paczki, odświeżenie strony w przeglądarce WWW i wyświetlenie treści pokazanej na rysunku 15.8. Kliknięcie przycisku kategorii powoduje filtrowanie produktów wyświetlanych użytkownikowi.



Rysunek 15.8. Wyświetlenie listy produktów

Podsumowanie

W tym rozdziale pokazałem, jak przygotować prosty i jednocześnie efektywny zestaw narzędzi programistycznych przeznaczony do tworzenia aplikacji internetowych za pomocą kompilatora TypeScriptu i pakietu webpack. Dowiedziałeś się, jak umieścić dane wygenerowane przez kompilator TypeScriptu w pliku paczki, a także jak zapewnić obsługę JSX, co upraszcza pracę z elementami HTML-a. W następnym rozdziale dokończę pracę nad tą aplikacją i przygotuję ją do wdrożenia.



Tworzenie aplikacji internetowej TypeScriptu — część II

W rozdziale dokończę tworzenie aplikacji internetowej TypeScriptu i przygotuję ją do wdrożenia. Przy okazji pokażę sposób, w jaki projekty TypeScriptu są dostosowywane do standardowych procesów wdrażania. W tabeli 16.1 wymienię opcje kompilatora TypeScriptu użyte w rozdziale.

Tabela 16.1. Opcje kompilatora TypeScriptu użyte w rozdziale

Opcja	Opis
emitDecoratorMetadata	Ta opcja powoduje dołączenie metadanych dekoratora w kodzie JavaScriptu wyemitowanym przez kompilator
experimentalDecorators	Ta opcja włącza obsługę dekoratorów
jsx	Ta opcja określa sposób przetwarzania elementów HTML-a w plikach TSX
jsxFactory	Ta opcja określa nazwę funkcji fabryki, która jest używana do zastępowania elementów HTML-a w plikach TSX
moduleResolution	Ta opcja określa styl używany podczas rozwiązywania zależności modułów
outDir	Ta opcja określa katalog, w którym zostaną umieszczone wygenerowane pliki JavaScriptu
rootDir	Ta opcja określa katalog główny używany przez kompilator do wyszukiwania plików TypeScriptu
target	Ta opcja określa docelową wersję języka JavaScript, dla której kompilator wygeneruje kod

Przygotowanie projektu

W rozdziale będę kontynuował używanie projektu *webapp* utworzonego w poprzednim rozdziale. Aby przygotować się do tego rozdziału, z poziomu katalogu *webapp* w powłoce wydaj polecenia przedstawione na listingu 16.1, odpowiadające za dodanie nowych pakietów do projektu.

■ **Wskazówka** Przykładowy projekt dla tego rozdziału, a także projekty dla pozostałych rozdziałów książki zostały umieszczone w archiwum dostępnym pod adresem <https://ftp.helion.pl/przyklady/tys4o2.zip>.

Listing 16.1. Dodawanie nowych pakietów do projektu

```
$ npm install --save-dev json-server@0.16.3
$ npm install --save-dev npm-run-all@4.1.5
```

Pakiet *json-server* to usługa sieciowa typu RESTful dostarczająca dane aplikacji i pozwalająca na zastąpienie lokalnych danych testowych, które były używane w rozdziale 15. Z kolei *npm-run-all* to użyteczne narzędzie umożliwiające uruchamianie wielu pakietów Node.js za pomocą tylko jednego polecenia.

Aby przygotować usługę sieciową z danymi, w pliku *webapp* umieść plik o nazwie *data.js* i zawartości przedstawionej na listingu 16.2.

Listing 16.2. Zawartość pliku *data.js* w katalogu *webapp*

```
module.exports = function () {
  return {
    products: [
      { id: 1, name: "Kajak", category: "Sporty wodne",
        description: "Łódka przeznaczona dla jednej osoby.", price: 275 },
      { id: 2, name: "Kamizelka ratunkowa", category: "Sporty wodne",
        description: "Chroni i dodaje uroku.", price: 48.95 },
      { id: 3, name: "Piłka", category: "Piłka nożna",
        description: "Zatwierdzone przez FIFA rozmiar i waga.", price: 19.50 },
      { id: 4, name: "Flagi narożne", category: "Piłka nożna",
        description: "Nadadzą twojemu boisku profesjonalny wygląd.",
          price: 34.95 },
      { id: 5, name: "Stadion", category: "Piłka nożna",
        description: "Składany stadion na 35 000 osób.", price: 79500 },
      { id: 6, name: "Czapka", category: "Szachy",
        description: "Zwiększa efektywność mózgu o 75%.", price: 16 },
      { id: 7, name: "Niestabilne krzesło", category: "Szachy",
        description: "Zmniejsza szanse przeciwnika.",
          price: 29.95 },
      { id: 8, name: "Ludzka szachownica", category: "Szachy",
        description: "Przyjemna gra dla całej rodziny.", price: 75 },
      { id: 9, name: "Błyszczący król", category: "Szachy",
        description: "Pokryty złotem i wysadzany diamentami król.", price: 1200 }
    ],
    orders: []
  }
}
```

Pakiet `json-server` zostanie skonfigurowany do używania danych przedstawionych na listingu 16.2, co spowoduje ich zerowanie w trakcie każdego uruchomienia aplikacji. (Wymieniony pakiet pozwala również na trwale przechowywanie danych, ale nie jest to aż tak użyteczne w przykładowym projekcie, w którym znana podstawa będzie znacznie lepszym rozwiązaniem).

W celu przeprowadzenia konfiguracji zestawu narzędzi programistycznych uaktualnij sekcję `scripts` pliku `package.json` zgodnie z kodem zamieszczonym na listingu 16.3.

Listing 16.3. Konfigurowanie narzędzi programistycznych w pliku `package.json` w katalogu `webapp`

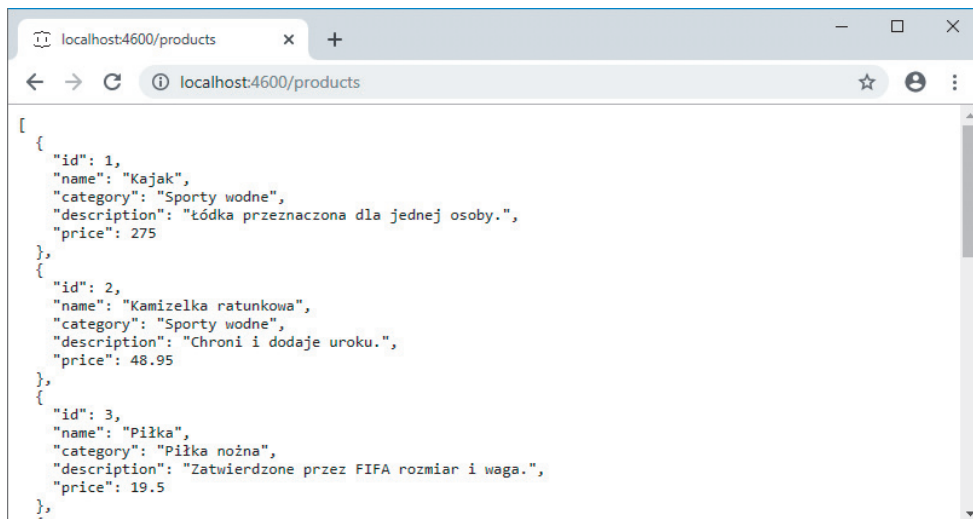
```
...
"scripts": {
  "json": "json-server data.js -p 4600",
  "wds": "webpack serve",
  "start": "npm-run-all -p json wds"
},
...
```

Wprowadzone zmiany pozwalają na to, aby wydanie jednego polecenia spowodowało dostarczenie danych przez usługę sieciową i uruchomienie serwera WWW. Następnie z poziomu katalogu `webapp` w powłoce wydaj polecenie przedstawione na listingu 16.4.

Listing 16.4. Uruchamianie narzędzi programistycznych

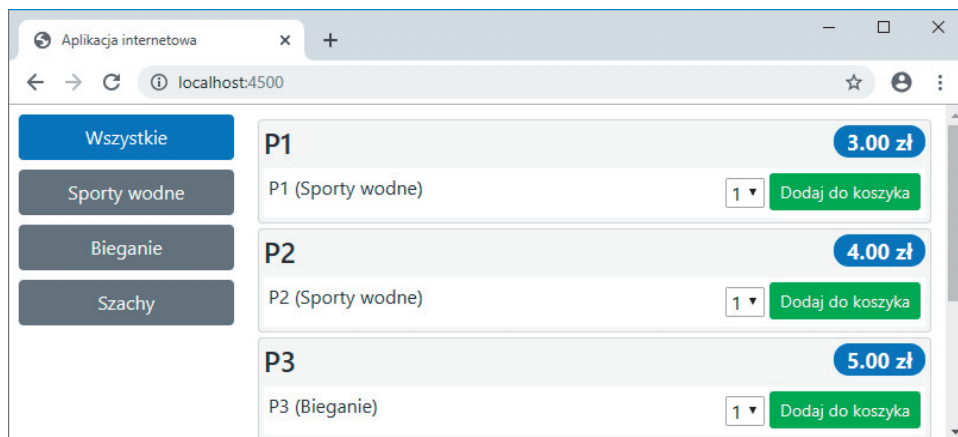
```
$ npm start
```

Usługa sieciowa zostanie uruchomiona, choć żadne dane nie zostały jeszcze zintegrowane z aplikacją. Aby przetestować działanie usługi sieciowej, w przeglądarce WWW przejdź pod adres `http://localhost:4600/products`, co spowoduje wygenerowanie danych wyjściowych pokazanych na rysunku 16.1.



Rysunek 16.1. Pobieranie danych z usługi sieciowej

Pliki TypeScriptu zostaną skompilowane, plik paczki będzie utworzony, a programistyczny serwer WWW rozpocznie nasłuchiwanie żądań HTTP. W nowym oknie przeglądarki WWW przejdź pod adres `http://localhost:4500`, co spowoduje wygenerowanie danych wyjściowych pokazanych na rysunku 16.2.



Rysunek 16.2. Uruchomiona przykładowa aplikacja

Dodawanie usługi sieciowej

W rozdziale 15. w budowanej aplikacji internetowej były wykorzystywane lokalne dane testowe. Takie rozwiązanie uznałem za użyteczne podczas przygotowywania projektu, ponieważ pozwala uniknąć zagłębiania się w szczegóły związane z pobieraniem danych z serwera. Jednak obecnie aplikacja nabiera kształtu, więc nadeszła odpowiednia pora na dodanie usługi sieciowej i rozpoczęcie pracy ze zdalnymi danymi. W nowym oknie powłoki przejdź do katalogu `webapp`, a następnie wydaj polecenie przedstawione na listingu 16.5, aby w ten sposób dodać kolejny pakiet do projektu.

Listing 16.5. Dodawanie kolejnego pakietu do projektu

```
$ npm install axios@0.21.1
```

Dostępnych jest wiele pakietów pozwalających na wykonywanie żądań HTTP w aplikacjach JavaScriptu. Wszystkie wykorzystują API oferowany przez przeglądarkę WWW. W omawianym przykładzie zdecydowałem się na popularny pakiet Axios, ponieważ praca z nim jest łatwa, a ponadto otrzymujemy komplet z deklaracjami TypeScriptu. W celu utworzenia źródła danych korzystającego z żądań HTTP dodaj do katalogu `src/data` plik o nazwie `remoteDataSource.ts` z kodem przedstawionym na listingu 16.6.

Listing 16.6. Zawartość pliku `remoteDataSource.ts` i katalogu `src/data`

```
import { AbstractDataSource } from "../abstractDataSource";
import { Product, Order } from "../entities";
import Axios from "axios";
```



```

const protocol = "http";
const hostname = "localhost";
const port = 4600;

const urls = {
  products: `${protocol}://${hostname}:${port}/products`,
  orders: `${protocol}://${hostname}:${port}/orders`
};

export class RemoteDataSource extends AbstractDataSource {

  loadProducts(): Promise<Product[]> {
    return Axios.get(urls.products).then(response => response.data);
  }

  storeOrder(): Promise<number> {
    let orderData = {
      lines: [...this.order.orderLines.values()].map(ol => ({
        productId: ol.product.id,
        productName: ol.product.name,
        quantity: ol.quantity
      })))
    }
    return Axios.post(urls.orders, orderData).then(response => response.data.id);
  }
}

```

■ **Wskazówka** Przeglądarka WWW udostępnia dwa API przeznaczone do wykonywania żądań HTTP. Tradycyjny API, `XMLHttpRequest`, jest obsługiwany przez wszystkie przeglądarki WWW, choć jednocześnie praca z nim należy do dość trudnych zadań. Istnieje również nowy API o nazwie `Fetch`, który jest łatwiejszy w obsłudze, ale pozostaje nieobsługiwany przez starsze wersje przeglądarek WWW. Wprowadzie każdego z wymienionych API można używać bezpośrednio, ale pakiety takie jak `Axios` dostarczają funkcjonalność ułatwiającą pracę z API i jednocześnie zapewniającą obsługę starszych wersji przeglądarek WWW.

Pakiet `Axios` dostarcza metody `get()` i `post()` pozwalające na wykonywanie żądań HTTP. Implementacja metody `loadProducts()` wykonuje żądanie GET do usługi sieciowej w celu pobrania danych produktu. Z kolei metoda `storeOrder()` przeprowadza konwersję szczegółów zamówienia na postać, którą można łatwo przechowywać i przekazywać dane do usługi sieciowej, korzystając w tym celu z żądania POST. Odpowiedź udzielana przez usługę sieciową to przechowywany obiekt zawierający wartość `id` unikatowo identyfikującą ten obiekt.

Wykorzystanie źródła danych w aplikacji

Wprowadzenie zmiany w konfiguracji jest wymagane, aby kompilator TypeScript mógł rozwiązać zależność dla pakietu `Axios`, jak pokazałem na listingu 16.7.

Listing 16.7. Konfigurowanie kompilatora TypeScriptu w pliku `tsconfig.json` w katalogu `webapp`

```
{
  "compilerOptions": {
    "target": "es2020",
    "outDir": "./dist",
    "rootDir": "./src",
    "jsx": "react",
    "jsxFactory": "createElement",
    "moduleResolution": "node"
  }
}
```

Ta zmiana wskazuje kompilatorowi, że zależności mogą być rozwiązywane przez sprawdzenie katalogu `node_modules`. Nie ma konieczności wprowadzania jakichkolwiek zmian dla pakietu `webpack`. Na listingu 16.8 przedstawiłem uaktualnioną wersję pliku `index.ts` wykorzystującego nowe źródło danych.

Listing 16.8. Zmianianie źródła danych w kodzie pliku `index.ts` w katalogu `src`

```
//import { LocalDataSource } from "../data/localDataSource";
import { RemoteDataSource } from "../data/remoteDataSource";
import { HtmlDisplay } from "../htmlDisplay";
import "bootstrap/dist/css/bootstrap.css";

let ds = new RemoteDataSource();

function displayData(): Promise<HTMLElement> {
  let display = new HtmlDisplay();
  display.props = {
    dataSource: ds
  }
  return display.getContent();
}

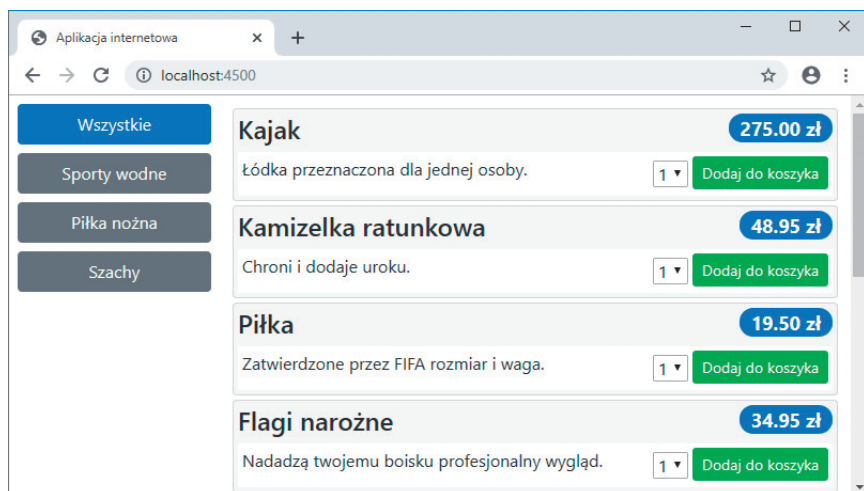
document.onreadystatechange = () => {
  if (document.readyState === "complete") {
    displayData().then(elem => {
      let rootElement = document.getElementById("app");
      rootElement.innerHTML = "";
      rootElement.appendChild(elem);
    });
  }
};
```

Narzędzia programistyczne muszą być ponownie uruchomione, aby mogła zostać wprowadzona zmiana konfiguracji na listingu 16.7. Naciśnij klawisze `Ctrl+C` w celu zatrzymania połączonych procesów usługi sieciowej i `webpacka`, a następnie z poziomu katalogu `webapp` w powłoce wydaj polecenie przedstawione na listingu 16.9.

Listing 16.9. Uruchamianie narzędzi programistycznych

```
$ npm start
```

W oknie przeglądarki WWW przejdź pod adres `http://localhost:4500`, co spowoduje pobranie danych z usługi sieciowej, jak pokazałem na rysunku 16.3.



Rysunek 16.3. Używanie zdalnych danych w przykładowej aplikacji

Używanie dekoratorów

Podobnie jak funkcjonalność JSX jest blisko związana z frameworkiem React, tak samo funkcjonalność dekoratorów jest powiązana z frameworkiem Angular (choć może być również użyteczna w aplikacjach Vue.js, o czym przekonasz się w rozdziale 21.). Obsługa dekoratorów jest propozycją do specyfikacji JavaScriptu, nie jest zbyt powszechnie stosowana poza aplikacjami tworzonymi z wykorzystaniem frameworka Angular i musi być włączona za pomocą odpowiedniej opcji kompilatora, jak pokazałem na listingu 16.10.

Listing 16.10. Włączanie obsługi dekoratorów w kodzie pliku `tsconfig.json` w katalogu `webapp`

```
{
  "compilerOptions": {
    "target": "es2020",
    "outDir": "./dist",
    "rootDir": "./src",
    "jsx": "react",
    "jsxFactory": "createElement",
    "moduleResolution": "node",
    "experimentalDecorators": true
  }
}
```

Dekorator to po prostu adnotacja, która może być zastosowana w celu modyfikacji klasy, metody, właściwości i parametrów. Aby utworzyć prosty dekorator dla projektu omawianego w rozdziale, dodaj do katalogu `src` plik o nazwie `decorators.js` zawierający kod przedstawiony na listingu 16.11.

Listing 16.11. Zawartość pliku *decorators.ts* w katalogu *src*

```
export const minimumValue = (propName: string, min: number) =>
  (constructor: any, methodName: string, descriptor: PropertyDescriptor): any => {
    const origFunction = descriptor.value;
    descriptor.value = async function wrapper(...args) {
      let results = await origFunction.apply(this, args);
      return results.map(r => ({ ...r, [propName]: r[propName] < min
        ? min : r[propName] }));
    }
  }
```

Tworzenie dekoratorów może być trudne, ponieważ opierają się one na zestawie zagnieżdżonych funkcji. Funkcja `minimumValue()` otrzymuje parametry zawierające nazwę właściwości i wartość minimalną, która ma być użyta. Wynikiem działania jest wywoływana podczas działania aplikacji funkcja z parametrami będącymi klasą, do której został zastosowany dekorator, nazwa metody i obiekt `PropertyDescriptor` opisujący metodę. Typ `PropertyDescriptor` to interfejs dostarczany przez TypeScript i opisujący kształt właściwości JavaScriptu. W przypadku metod wartość właściwości `PropertyDescriptor.value` jest używana do przechowywania funkcji i będzie zastępowana przez implementację wywołującą metodę początkową, a następnie przetwarzającą wynik w celu zdefiniowania wartości minimalnej właściwości.

W kodzie na listingu 16.12 przedstawiłem zastosowanie dekoratora `minimumValue` w metodzie zwracającej obiekty `Product`, aby wartość przypisana właściwości `price` wynosiła minimum 30.

Listing 16.12. Stosowanie dekoratora w kodzie pliku *abstractDataSource.ts* w katalogu *src/data*

```
import { Product, Order } from "../entities";
import { minimumValue } from "../decorators";

export type ProductProp = keyof Product;

export abstract class AbstractDataSource {
  private _products: Product[];
  private _categories: Set<string>;
  public order: Order;
  public loading: Promise<void>;

  constructor() {
    this._products = [];
    this._categories = new Set<string>();
    this.order = new Order();
    this.loading = this.getData();
  }

  @minimumValue("price", 30)
  async getProducts(sortProp: ProductProp = "id",
    category? : string): Promise<Product[]> {
    await this.loading;
    return this.selectProducts(this._products, sortProp, category);
  }

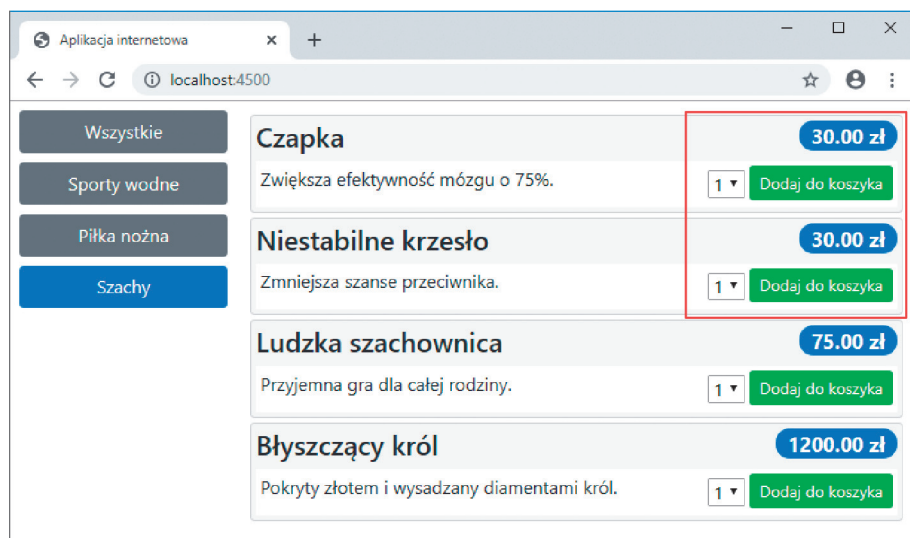
  // Pozostałe metody zostały pominięte.
}
```

Zatrzymaj działanie narzędzi programistycznych przez naciśnięcie klawiszy *Ctrl+C*, a następnie z poziomu katalogu *webapp* w powłoce wydaj polecenie przedstawione na listingu 16.13, aby ponownie uruchomić narzędzia programistyczne, ale już z nową konfiguracją.

Listing 16.13. Uruchamianie narzędzi programistycznych

```
$ npm start
```

W wyniku wprowadzonej zmiany cena minimalna produktu wynosi 30 zł, jak widać na rysunku 16.4.



Rysunek 16.4. Używanie dekoratora w celu wymuszenia ceny minimalnej

Używanie metadanych dekoratora

Funkcje dekoratorów są wywoływane w trakcie działania aplikacji, co oznacza, że są one pozbawione dostępu do informacji o typie znajdujących się w plikach kodu źródłowego TypeScriptu lub o typach ustalanych przez kompilator. Aby ułatwić proces tworzenia dekoratorów, kompilator TypeScriptu może dołączać do dekoratorów metadane dostarczające szczegółowe informacje o używanych typach. Włączenie takiej funkcjonalności wymaga wprowadzenia zmiany konfiguracyjnej w kompilatorze TypeScriptu, jak pokazałem na listingu 16.14.

Listing 16.14. Konfigurowanie kompilatora TypeScriptu w pliku *tsconfig.json* w katalogu *webapp*

```
{
  "compilerOptions": {
    "target": "es2020",
    "outDir": "./dist",
    "rootDir": "./src",
    "jsx": "react",
```

```

    "jsxFactory": "createElement",
    "moduleResolution": "node",
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true
  }
}

```

Opcja konfiguracyjna `emitDecoratorMetadata` wymaga dodania kolejnego pakietu do projektu. W nowym oknie powłoki przejdź do katalogu *webapp*, a następnie wydaj polecenie przedstawione na listingu 16.15.

Listing 16.15. Dodawanie nowego pakietu do projektu

```
$ npm install reflect-metadata@0.1.13
```

Podczas kompilacji kompilator TypeScript dołączy do generowanego kodu JavaScriptu metadane, które będą dostępne za pomocą pakietu `reflect-metadata`. W przypadku dekoratora zastosowanego dla metody na listingu 16.12 kompilator dodaje metadane wymienione w tabeli 16.2.

Tabela 16.2. Metadane dla dekoratora zastosowanego w metodzie `getProducts()` na listingu 16.12

Nazwa elementu	Opis
<code>design:type</code>	Element opisuje element, do którego został zastosowany dekorator. W przypadku dekoratora użytego na listingu 16.12 będzie to funkcja (Function)
<code>design:paramtypes</code>	Element opisuje typy parametrów funkcji, w których zostanie zastosowany dekorator. W przypadku dekoratora użytego na listingu 16.12 będzie to <code>[String, String]</code> — dwa parametry akceptujące wartości w postaci ciągu tekstowego
<code>design:returntype</code>	Element opisuje typ wyniku działania funkcji, w których zostanie zastosowany dekorator. W przypadku dekoratora użytego na listingu 16.12 będzie to obietnica (Promise)

Na listingu 16.16 zdefiniowałem nowy dekorator wykorzystujący funkcjonalność metadanych.

Listing 16.16. Definiowanie dekoratora w kodzie pliku *decorators.ts* w katalogu *src*

```

import "reflect-metadata";

export const minimumValue = (propName: string, min: number) =>
  (constructor: any, methodName: string, descriptor: PropertyDescriptor): any => {
    const origFunction = descriptor.value;
    descriptor.value = async function wrapper(...args) {
      let results = await origFunction.apply(this, args);
      return results.map(r => ({ ...r, [propName]: r[propName] < min
        ? min : r[propName] }));
    }
  }

export const addClass = (selector: string, ...classNames: string[]) =>

```

```

    (constructor: any, methodName: string, descriptor: PropertyDescriptor): any => {
      if (Reflect.getMetadata("design:returntype",
        constructor, methodName) === HTMLElement) {
        const origFunction = descriptor.value;
        descriptor.value = function wrapper(...args) {
          let content: HTMLElement = origFunction.apply(this, args);
          content.querySelectorAll(selector).forEach(elem =>
            classNames.forEach(c => elem.classList.add(c)));
          return content;
        }
      }
    }
  }
}

```

Pakiet `reflect-metadata` dodaje metody do `Reflect`, czyli funkcjonalności JavaScriptu pozwalającej na analizowanie obiektów. Zmiany wprowadzone w kodzie na listingu 16.16 używają metody `Reflect.getMetadata()` w celu pobrania elementu `design:returntype` gwarantującego, że dekorator będzie modyfikował jedynie metody zwracające obiekty `HTMLElement`. Ten dekorator akceptuje selektor CSS używany do odszukania określonych elementów generowanych przez metodę i dodania ich do jednej lub więcej klas. Na listingu 16.17 przedstawiłem zastosowanie nowego dekoratora w kodzie HTML-a wygenerowanym przez klasę `ProductList`.

Listing 16.17. Stosowanie dekoratora w kodzie pliku `productList.tsx` w katalogu `src`

```

import { createElement } from "../tools/jsxFactory";
import { Product } from "../data/entities";
import { ProductItem } from "../productItem";
import { CategoryList } from "../categoryList";
import { addClass } from "../decorators";

export class ProductList {
  props: {
    products: Product[],
    categories: string[],
    selectedCategory: string,
    addToOrderCallback?: (product: Product, quantity: number) => void,
    filterCallback?: (category: string) => void;
  }

  @addClass("select", "bg-info", "m-1")
  getContent(): HTMLElement {
    return <div className="container-fluid">
      <div className="row">
        <div className="col-3 p-2">
          <CategoryList categories={ this.props.categories }
            selectedCategory={ this.props.selectedCategory }
            callback={ this.props.filterCallback } />
        </div>
        <div className="col-9 p-2">
          {
            this.props.products.map(p =>
              <ProductItem product={ p }
                callback={ this.props.addToOrderCallback } />
            )
          }
        </div>
      </div>
    </div>
  }
}

```

```

    </div>
  </div>
</div>
}
}

```

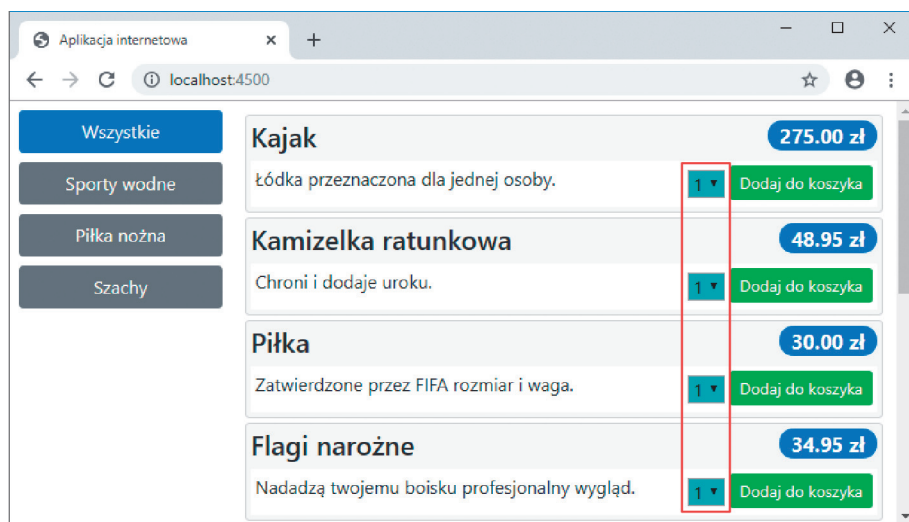
Te przykłady łączą funkcjonalności blisko związane z frameworkami React i Angular, pokazując, że oba wymienione frameworki zostały zbudowane z wykorzystaniem funkcji standardowych i mogą być zastosowane razem w tej samej aplikacji (choć takie rozwiązanie jest rzadko stosowane w rzeczywistych projektach).

Zatrzymaj działanie narzędzi programistycznych przez naciśnięcie klawiszy `Ctrl+C`, a następnie z poziomu katalogu `webapp` w powłoce wydaj polecenie przedstawione na listingu 16.18, aby ponownie uruchomić narzędzia programistyczne, ale już z nową konfiguracją.

Listing 16.18. Uruchamianie narzędzi programistycznych

```
$ npm start
```

Utworzony zostanie nowy plik paczki z metadanymi i pakietem wymaganym do jego użycia. Aplikacja dekoratora odszukuje elementy `<select>` w kodzie wygenerowanym przez klasę `ProductList`, a następnie dodaje im klasę zmieniającą kolor tła i ilość wolnego miejsca wokół elementu, jak pokazałem na rysunku 16.5.



Rysunek 16.5. Używanie dekoratora w celu modyfikacji elementów HTML-a

Dokończenie aplikacji

Większa część rozdziału 15. została poświęcona na przygotowanie narzędzi programistycznych i skonfigurowanie projektu do zapewnienia obsługi JSX, co ma ułatwić pracę z treścią HTML-a w plikach kodu źródłowego aplikacji. Skoro podstawowa struktura aplikacji jest już gotowa,

dodawanie nowych funkcjonalności jest względnie prostym zadaniem. W tej części rozdziału nie wprowadzę żadnych nowych funkcjonalności TypeScriptu, a jedynie zajmę się dokończeniem pracy nad aplikacją.

Dodawanie klasy Header

Aby wyświetlić nagłówek dostarczający użytkownikowi podsumowanie dotyczące wybranych produktów, należy dodać do katalogu *src* plik o nazwie *header.tsx* z kodem przedstawionym na listingu 16.19.

Listing 16.19. Zawartość pliku *header.tsx* w katalogu *src*

```
import { createElement } from "../tools/jsxFactory";
import { Order } from "../data/entities";

export class Header {

  props: {
    order: Order,
    submitCallback: () => void
  }

  getContent(): HTMLElement {
    let count = this.props.order.productCount;
    return <div className="p-1 bg-secondary text-white text-right">
      { count === 0 ? "(brak produktów)"
        : `Liczba produktów: ${ count }, ${ this.props.order.total.toFixed(2)} zł` }
      <button className="btn btn-sm btn-primary m-1"
        onclick={ this.props.submitCallback }>
        Złóż zamówienie
      </button>
    </div>
  }
}
```

Poprzez właściwości ta klasa otrzymuje obiekt *Order* i funkcję wywołania zwrótnego. Wynikiem działania kodu jest wyświetlenie krótkiego podsumowania obiektu *Order* z przyciskiem, którego kliknięcie rozpoczyna wykonywanie funkcji wywołania zwrótnego.

Dodawanie klasy obsługującej szczegóły zamówienia

W celu wyświetlenia szczegółów zamówienia dodaj do katalogu *src* plik o nazwie *orderDetails.tsx* zawierający kod przedstawiony na listingu 16.20.

Listing 16.20. Zawartość pliku *orderDetails.tsx* w katalogu *src*

```
import { createElement } from "../tools/jsxFactory";
import { Product, Order } from "../data/entities";

export class OrderDetails {
```

```

    props: {
      order: Order
      cancelCallback: () => void,
      submitCallback: () => void
    }

    getContent(): HTMLElement {
      return <div>
        <h3 className="text-center bg-primary text-white p-2">
          Informacje o zamówieniu
        </h3>
        <div className="p-3">
          <table className="table table-sm table-striped">
            <thead>
              <tr>
                <th>Ilość</th><th>Produkt</th>
                <th className="text-right">Cena</th>
                <th className="text-right">Wartość</th>
              </tr>
            </thead>
            <tbody>
              { this.props.order.orderLines.map(line =>
                <tr>
                  <td>{ line.quantity }</td>
                  <td>{ line.product.name }</td>
                  <td className="text-right">
                    { line.product.price.toFixed(2) } zł
                  </td>
                  <td className="text-right">
                    { line.total.toFixed(2) } zł
                  </td>
                </tr>
              )}
            </tbody>
            <tfoot>
              <tr>
                <th className="text-right" colSpan="3">Razem:</th>
                <th className="text-right">
                  { this.props.order.total.toFixed(2) } zł
                </th>
              </tr>
            </tfoot>
          </table>
        </div>
        <div className="text-center">
          <button className="btn btn-secondary m-1"
            onclick={ this.props.cancelCallback }>
            Wróć
          </button>
          <button className="btn btn-primary m-1"
            onclick={ this.props.submitCallback }>
            Złóż zamówienie
          </button>
        </div>
      </div>
    }
  }
}

```

```

    </div>
  }
}

```

Działanie klasy `OrderDetails` polega na wyświetleniu tabeli zawierającej szczegóły zamówienia oraz przyciski pozwalające na powrót do listy produktów i na złożenie zamówienia.

Dodawanie klasy obsługującej potwierdzenie zamówienia

W celu wyświetlenia komunikatu po złożeniu zamówienia dodaj do katalogu `src` plik o nazwie `summary.tsx` zawierający kod przedstawiony na listingu 16.21.

Listing 16.21. Zawartość pliku `summary.tsx` w katalogu `src`

```

import { createElement } from "../tools/jsxFactory";

export class Summary {

  props: {
    orderId: number,
    callback: () => void
  }

  getContent(): HTMLElement {
    return <div className="m-2 text-center">
      <h2>Dziękujemy!</h2>
      <p>Dziękujemy za złożenie zamówienia.</p>
      <p>Numer zamówienia #{ this.props.orderId }</p>
      <p>Zamówione produkty zostaną wkrótce wysłane.</p>
      <button className="btn btn-primary" onclick={ this.props.callback }>
        OK
      </button>
    </div>
  }
}

```

Działanie tej klasy polega na wyświetleniu krótkiego komunikatu zawierającego identyfikator zamówienia przypisany przez usługę sieciową oraz przycisku uruchamiającego funkcję wywołania zwrotnego przekazaną za pomocą właściwości.

Zakończenie pracy nad aplikacją

Ostatnim krokiem jest dodanie kodu pozwalającego na połączenie utworzonych wcześniej klas oraz dostarczenie im za pomocą właściwości wymaganych danych i funkcji wywołania zwrotnego, a także na wyświetlenie wygenerowanej treści HTML-a, jak pokazałem na listingu 16.22.

Listing 16.22. Dokończenie aplikacji w kodzie pliku `htmlDisplay.tsx` w katalogu `src`

```

import { createElement } from "../tools/jsxFactory";
import { Product, Order } from "../data/entities";
import { AbstractDataSource } from "../data/abstractDataSource";
import { ProductList } from "../productList";

```

```

import { Header } from "../header";
import { OrderDetails } from "../orderDetails";
import { Summary } from "../summary";

enum DisplayMode {
    List, Details, Complete
}

export class HtmlDisplay {
    private containerElem: HTMLElement;
    private selectedCategory: string;
    private mode: DisplayMode = DisplayMode.List;
    private orderId: number;

    constructor() {
        this.containerElem = document.createElement("div");
    }

    props: {
        dataSource: AbstractDataSource;
    }

    async getContent(): Promise<HTMLElement> {
        await this.updateContent();
        return this.containerElem;
    }

    async updateContent() {
        let products = await this.props.dataSource
            .getProducts("id", this.selectedCategory);
        let categories = await this.props.dataSource.getCategories();
        this.containerElem.innerHTML = "";
        let contentElem: HTMLElement;
        switch (this.mode) {
            case DisplayMode.List:
                contentElem = this.getListContent(products, categories);
                break;
            case DisplayMode.Details:
                contentElem = <OrderDetails order={ this.props.dataSource.order }
                    cancelCallback={ this.showList }
                    submitCallback={ this.submitOrder } />
                break;
            case DisplayMode.Complete:
                contentElem = <Summary orderId={ this.orderId }
                    callback= { this.showList } />
                break;
        }
        this.containerElem.appendChild(contentElem);
    }

    getListContent(products: Product[], categories: string[]): HTMLElement {
        return <div>
            <Header order={ this.props.dataSource.order }
                submitCallback={ this.showDetails } />

```

```

    <ProductList products={ products } categories={ categories }
      selectedCategory={ this.selectedCategory }
      addToOrderCallback={ this.addToOrder }
      filterCallback={ this.selectCategory } />
  </div>
}

addToOrder = (product: Product, quantity: number) => {
  this.props.dataSource.order.addProduct(product, quantity);
  this.updateContent();
}

selectCategory = (selected: string) => {
  this.selectedCategory = selected === "Wszystkie" ? undefined : selected;
  this.updateContent();
}

showDetails = () => {
  this.mode = DisplayMode.Details;
  this.updateContent();
}

showList = () => {
  this.mode = DisplayMode.List;
  this.updateContent();
}

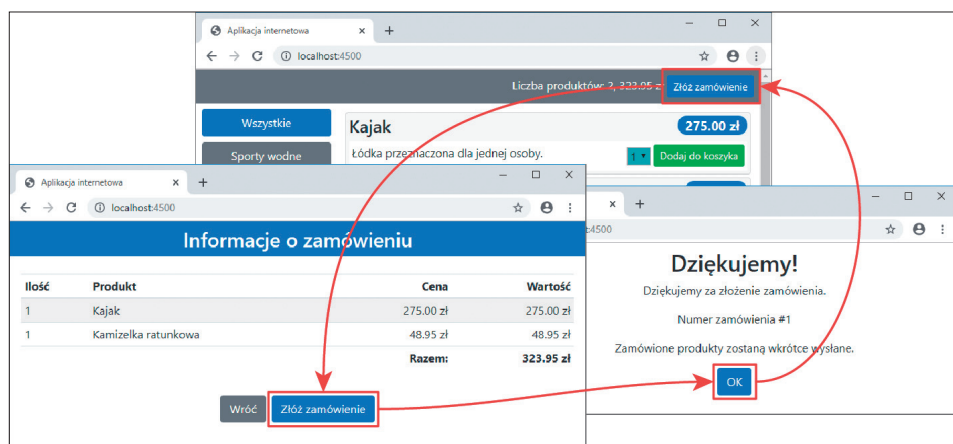
submitOrder = () => {
  this.props.dataSource.storeOrder().then(id => {
    this.orderId = id;
    this.props.dataSource.order = new Order();
    this.mode = DisplayMode.Complete;
    this.updateContent();
  });
}
}

```

Usprawnienia wprowadzone w klasie `HtmlDisplay` mają na celu ustalenie, które klasy JSX są używane do wyświetlania treści użytkownikowi. Znaczenie kluczowe ma tutaj właściwość `mode`, wykorzystująca wartości wyliczenia `DisplayMode` do pobierania treści oraz metody `showDetails()`, `showList()` i `submitOrder()` do zmiany wartości `mode` i uaktualniania wyświetlanej treści.

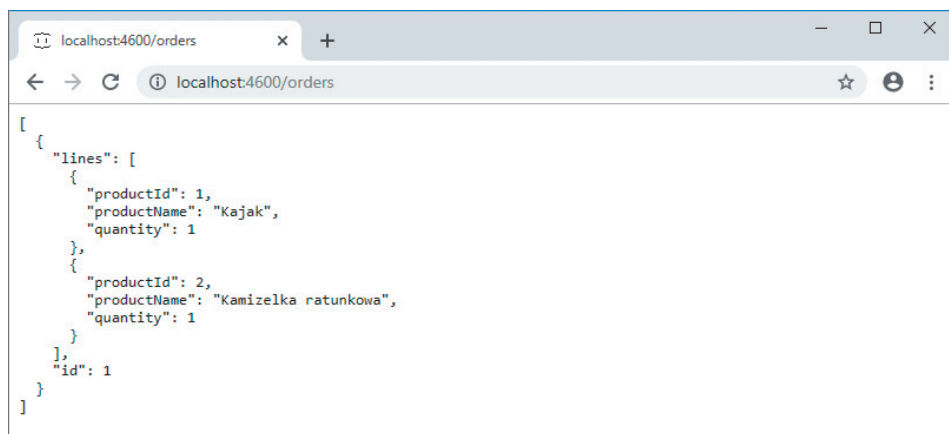
W aplikacji internetowej często znajduje się jedna klasa, która staje się dość skomplikowana, nawet w przypadku tak prostej aplikacji jak budowana w tym przykładzie. Wykorzystanie dowolnego z frameworków omówionych w dalszej części książki może pomóc, choć sprowadza się do wyrażenia złożoności w inny sposób — najczęściej w postaci skomplikowanego zestawu mapowań między adresami URL obsługiwanyymi przez aplikację a klasami treści odpowiadającymi tym adresom.

Po zapisaniu wszystkich zmian przeglądarka WWW wczyta nowy plik paczki, a użytkownik będzie miał możliwość wyboru produktów, przeglądania wybranych, a także złożenia zamówienia i przekazania go do serwera, jak pokazałem na rysunku 16.6.



Rysunek 16.6. Używanie przykładowej aplikacji

Po złożeniu zamówienia dane przekazane do serwera można przejrzeć po przejściu pod adres <http://localhost:4600/orders>, jak pokazałem na rysunku 16.7.



Rysunek 16.7. Przeglądanie złożonych zamówień

■ **Uwaga** Zamówienia nie są trwale przechowywane i zostaną utracone po zatrzymaniu lub ponownym uruchomieniu aplikacji. Dodaniem obsługi trwałego magazynu danych zajmę się w dalszej części rozdziału.

Wdrażanie aplikacji

Serwer Webpack Development Server i dostarczany przez niego zestaw narzędzi nie mogą być stosowane w środowisku produkcyjnym. Dlatego też trzeba wykonać jeszcze nieco pracy, aby przygotować aplikację do wdrożenia, co dokładnie przedstawię w tym podrozdziale.

Dodawanie pakietu produkcyjnego serwera HTTP

Jak wcześniej wspomniałem, serwer Webpack Development Server nie powinien być używany w środowisku produkcyjnym, ponieważ dostarczane przez niego funkcjonalności koncentrują się na dynamicznym tworzeniu pliku paczki po wykryciu każdej zmiany w pliku kodu źródłowego. W przypadku środowiska produkcyjnego wymagany jest zwykle serwer HTTP, który będzie dostarczał pliki HTML-a, CSS i JavaScriptu przeglądarce WWW. Dobrym wyborem dla prostych projektów jest dostępny jako *open source* serwer Express, który ma postać pakietu JavaScriptu wykonywanego przez środowisko uruchomieniowe Node.js. Zatrzymaj działanie narzędzi programistycznych przez naciśnięcie klawiszy *Ctrl+C*, a następnie z poziomu katalogu *webapp* w powłoce wydaj polecenie przedstawione na listingu 16.23, aby zainstalować niezbędny pakiet.

Listing 16.23. Dodawanie pakietu zawierającego serwer Express dla wdrożonej aplikacji

```
$ npm install --save-dev express@4.17.1
```

■ **Uwaga** Pakiet *express* może być już zainstalowany, ponieważ jest używany również przez inne narzędzia. Mimo to dobrą praktyką jest jego dodanie, ponieważ oznacza to umieszczenie odpowiednich zależności w pliku *project.json*.

Tworzenie pliku dla trwałego magazynu danych

Pakiet *json-server* będzie trwale przechowywał dane, gdy zostanie skonfigurowany do używania pliku JSON zamiast pliku JavaScriptu pozwalającego na ponowne przekazywanie danych podczas pracy nad aplikacją. Do katalogu *webapp* dodaj plik o nazwie *data.json* zawierający kod przedstawiony na listingu 16.24.

Listing 16.24. Zawartość pliku *data.json* w katalogu *webapp*

```
{
  "products": [
    { "id": 1, "name": "Kajak", "category": "Sporty wodne",
      "description": "Łódka przeznaczona dla jednej osoby.", "price": 275 },
    { "id": 2, "name": "Kamizelka ratunkowa", "category": "Sporty wodne",
      "description": "Chroni i dodaje uroku.", "price": 48.95 },
    { "id": 3, "name": "Piłka", "category": "Piłka nożna",
      "description": "Zatwierdzone przez FIFA rozmiar i waga.", "price": 19.50 },
    { "id": 4, "name": "Flagi naróżne", "category": "Piłka nożna",
      "description": "Nadadzą twojemu boisku profesjonalny wygląd.",
      "price": 34.95 },
    { "id": 5, "name": "Stadion", "category": "Piłka nożna",
      "description": "Składany stadion na 35 000 osób.", "price": 79500 },
    { "id": 6, "name": "Czapka", "category": "Szachy",
      "description": "Zwiększa efektywność mózgu o 75%.", "price": 16 },
    { "id": 7, "name": "Niestabilne krzesło", "category": "Szachy",
      "description": "Zmniejsza szanse przeciwnika.",
      "price": 29.95 },
    { "id": 8, "name": "Ludzka szachownica", "category": "Szachy",
```

```

        "description": "Przyjemna gra dla całej rodziny.", "price": 75 },
    { "id": 9, "name": "Błyszczący król", "category": "Szachy",
      "description": "Pokryty złotem i wysadzany diamentami król.", "price": 1200 }
  ],
  "orders": []
}

```

W tym pliku znajdują się dokładnie te same informacje o produktach, które wcześniej (listing 16.2) umieściłem w pliku JavaScriptu. Jednak tym razem plik ma format JSON, co oznacza, że dane zamówienia nie zostaną utracone po zatrzymaniu lub po ponownym uruchomieniu aplikacji.

Utworzenie serwera

Aby utworzyć serwer dostarczający przeglądarce WWW aplikację i jej dane, dodaj do katalogu *webapp* plik o nazwie *server.js* i umieść w nim kod przedstawiony na listingu 16.25.

Listing 16.25. Zawartość pliku *server.js* w katalogu *webapp*

```

const express = require("express");
const jsonServer = require("json-server");

const app = express();
app.use("/", express.static("dist"));
app.use("/", express.static("assets"));
const router = jsonServer.router("data.json");
app.use(jsonServer.bodyParser)
app.use("/api", (req, resp, next) => router(req, resp, next));

const port = process.argv[3] || 4000;
app.listen(port, () => console.log(`Serwer nasłuchuje na porcie numer ${port}`));

```

Polecenia zdefiniowane w pliku *server.js* konfiguruja pakiety *express* i *json-server* w taki sposób, aby zawartość katalogów *dist* i *assets* była używana w celu dostarczania plików statycznych, a adresy URL poprzedzone prefiksem */api* były obsługiwane przez usługę sieciową.

■ **Wskazówka** Istnieje możliwość utworzenia kodu serwera w języku TypeScript, skompilowania go i wygenerowania kodu JavaScriptu, który następnie będzie uruchamiany w środowisku produkcyjnym. Takie rozwiązanie jest dobre, jeśli masz szczególnie skomplikowany kod serwera. Przekonałem się, że bezpośrednia praca z JavaScriptem jest łatwiejsza w przypadku prostszych projektów wykorzystujących funkcjonalność dostarczaną przez różne pakiety.

Używanie względnych adresów URL do obsługi żądań danych

Usługa sieciowa dostarczająca dane aplikacji będzie działała równolegle z serwerem Webpack Development Server. W środowisku produkcyjnym oba typy żądań HTTP są nasłuchiwane na pojedynczym porcie. W omawianym przykładzie wymagana jest zmiana adresów URL używanych przez klasę *RemoteDataSource*, jak pokazałem na listingu 16.26.

Listing 16.26. Używanie względnych adresów URL w kodzie pliku *remoteDataSource.ts* w katalogu *src/data*

```
import { AbstractDataSource } from "../abstractDataSource";
import { Product, Order } from "../entities";
import Axios from "axios";

// const protocol = document.location.protocol;
// const hostname = document.location.hostname;
// const port = 4600;

const urls = {
  // products: `${protocol}//${hostname}:${port}/products`,
  // orders: `${protocol}//${hostname}:${port}/orders`
  products: "/api/products",
  orders: "/api/orders"
};

export class RemoteDataSource extends AbstractDataSource {

  loadProducts(): Promise<Product[]> {
    return Axios.get(urls.products).then(response => response.data);
  }
  storeOrder(): Promise<number> {
    let orderData = {
      lines: [...this.order.orderLines.values()].map(ol => ({
        productId: ol.product.id,
        productName: ol.product.name,
        quantity: ol.quantity
      })))
    }
    return Axios.post(urls.orders, orderData).then(response => response.data.id);
  }
}
```

Podane adresy URL są względne dla używanych do obsługi żądań dokumentu HTML-a. Oznacza to stosowanie się do konwencji, zgodnie z którą żądania danych są poprzedzone prefiksem */api*.

Kompilacja aplikacji

Z poziomu katalogu *webapp* w powłoce wydaj polecenie przedstawione na listingu 16.27, aby w ten sposób utworzyć plik paczki, który będzie mógł zostać wykorzystany w środowisku produkcyjnym.

Listing 16.27. Tworzenie pliku paczki dla środowiska produkcyjnego

```
$ npx webpack --mode "production"
```

Gdy wartością argumentu *mode* jest *production*, webpack tworzy paczkę, której zawartość jest zminimalizowana, czyli zoptymalizowana pod kątem wielkości kodu, a nie jego czytelności.

Kompilacja może chwilę potrwać i spowoduje wygenerowanie danych wyjściowych podobnych do tutaj przedstawionych, które zawierają informacje o plikach użytych podczas przygotowywania paczki.

```
asset bundle.js 1.91 MiB [emitted] [minimized] [big] (name: main) 1 related asset
orphan modules 15.7 KiB [orphan] 13 modules
runtime modules 878 bytes 4 modules
modules by path ./node_modules/axios/ 41.3 KiB
  modules by path ./node_modules/axios/lib/helpers/*.js 9.02 KiB 10 modules
  modules by path ./node_modules/axios/lib/core/*.js 12.1 KiB 9 modules
  modules by path ./node_modules/axios/lib/*.js 12.7 KiB 3 modules
  modules by path ./node_modules/axios/lib/cancel/*.js 1.69 KiB 3 modules
  2 modules
modules by path ./node_modules/css-loader/dist/runtime/*.js 3.78 KiB
  ./node_modules/css-loader/dist/runtime/cssWithMappingToString.js 2.21 KiB [built] [code
  generated]
  ./node_modules/css-loader/dist/runtime/api.js 1.57 KiB [built] [code generated]
./src/index.ts + 13 modules 16.4 KiB [built] [code generated]
./node_modules/style-loader/dist/runtime/injectStylesIntoStyleTag.js 6.67 KiB [built] [code
  generated]
./node_modules/css-loader/dist/cjs.js!./node_modules/bootstrap/dist/css/bootstrap.css 707
  KiB [built] [code generated]
./node_modules/reflect-metadata/Reflect.js 50 KiB [built] [code generated]
WARNING in asset size limit: The following asset(s) exceed the recommended size limit
(244 KiB).
This can impact web performance.
Assets:
  bundle.js (1.91 MiB)
WARNING in entrypoint size limit: The following entrypoint(s) combined asset size exceeds
the recommended limit (244 KiB). This can impact web performance.
Entrypoints:
  main (1.91 MiB)
    bundle.js
WARNING in webpack performance recommendations:
You can limit the size of your bundles by using import() or require.ensure to lazy load some
parts of your application.
For more info visit https://webpack.js.org/guides/code-splitting/
webpack 5.17.0 compiled with 3 warnings in 5129 ms
```

Pliki TypeScriptu zostały skompilowane na postać kodu JavaScriptu, jakby znajdowały się w środowisku produkcyjnym. Plik paczki został umieszczony w katalogu *dist*.

Testowanie gotowej aplikacji

Aby upewnić się o prawidłowej kompilacji aplikacji i zastosowaniu wszystkich wprowadzonych zmian konfiguracyjnych, z poziomu katalogu *webapp* w powłoce wydaj polecenie przedstawione na listingu 16.28.

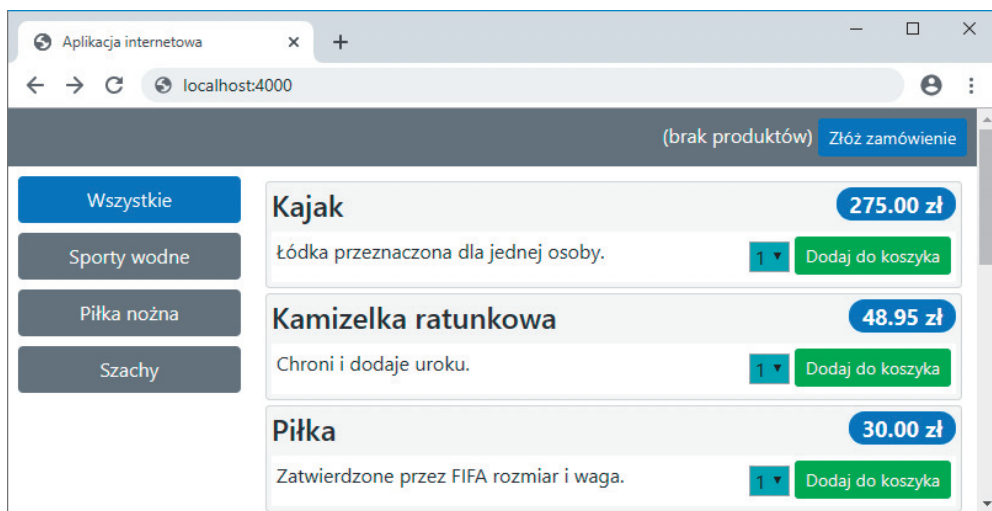
Listing 16.28. Uruchamianie serwera produkcyjnego

```
$ node server.js
```

Kod przedstawiony na listingu 16.25 zostanie wykonany i spowoduje wygenerowanie następujących danych wyjściowych:

```
Serwer nasłuchuje na porcie numer 4000
```

Otwórz okno przeglądarki WWW i przejdź pod adres `http://localhost:4000`, a zobaczysz działającą aplikację, jak pokazałem na rysunku 16.8.



Rysunek 16.8. Uruchomiona aplikacja dla środowiska produkcyjnego

Umieszczanie aplikacji w kontenerze

Ten rozdział zamierzam zakończyć utworzeniem dla przykładowej aplikacji kontenera, który następnie będzie można wdrożyć w środowisku produkcyjnym. W chwili powstawania książki Docker był najpopularniejszym rozwiązaniem w zakresie tworzenia kontenerów. Kontener to po prostu okrojona wersja systemu Linux zapewniająca wystarczającą funkcjonalność do uruchomienia aplikacji. Większość platform przetwarzania w chmurze i silników hostingowych zapewnia obsługę Dockera, a związane z nim narzędzia działają w większości popularnych systemów operacyjnych.

Instalowanie Dockera

Pierwszym krokiem jest pobranie i zainstalowanie narzędzi Dockera w komputerze używanym do utworzenia aplikacji. Te narzędzia znajdziesz na stronie <https://www.docker.com/products>. Dostępne są wersje dla systemów macOS, Windows i Linux, a także specjalizowane wersje przeznaczone do działania na platformach chmur Amazona i Microsoftu. Na potrzeby materiału przedstawionego w rozdziale i omawianej tutaj przykładowej aplikacji w zupełności wystarczy bezpłatne wydanie Community.

■ **Ostrzeżenie** Wadą używania Dockera jest to, że firma, która opracowała to oprogramowanie, zyskała reputację wdrażającej poważne zmiany prowadzące do niezgodności nowszych wersji z poprzednimi. Może to oznaczać, że przykłady przedstawione w rozdziale nie będą w nowszych wydaniach Dockera działać zgodnie z oczekiwaniami. Jeżeli napotkasz problemy, pomocy szukaj w repozytorium przygotowanym dla książki (<https://github.com/Apress/essential-typescript>) lub skontaktuj się ze mną bezpośrednio, pisząc na adres adam@adam-freeman.com.

Przygotowanie aplikacji

Pracę należy zacząć od utworzenia pliku konfiguracyjnego dla menedżera pakietów Node.js, który będzie użyty do pobrania pakietów dodatkowych wymaganych przez aplikację uruchamianą w kontenerze. W katalogu *webapp* utwórz więc plik o nazwie *deploy-package.json* i umieść w nim kod przedstawiony na listingu 16.29.

Listing 16.29. Zawartość pliku *deploy-package.json* w katalogu *webapp*

```
{
  "name": "webapp",
  "description": "Aplikacja internetowa TypeScriptu",
  "repository": "https://github.com/Apress/essential-typescript",
  "license": "0BSD",
  "devDependencies": {
    "express": "4.17.4",
    "json-server": "0.16.3"
  }
}
```

Sekcja *devDependencies* wymienia pakiety niezbędne do uruchomienia aplikacji w kontenerze. Wszystkie pakiety, dla których istnieją polecenia *import* w plikach kodu źródłowego aplikacji, zostały umieszczone w pliku paczki wygenerowanym przez narzędzie *webpack*. Pozostałe opcje opisują aplikację i zostały umieszczone w kodzie w celu uniknięcia komunikatów ostrzeżeń generowanych podczas tworzenia kontenera.

Tworzenie kontenera Dockera

Aby zdefiniować kontener, dodaj do katalogu *webapp* plik o nazwie *Dockerfile* (bez żadnego rozszerzenia) i umieść w nim kod przedstawiony na listingu 16.30.

Listing 16.30. Zawartość pliku *Dockerfile* w katalogu *webapp*

```
FROM node:14.15.4

RUN mkdir -p /usr/src/webapp

COPY dist /usr/src/webapp/dist
COPY assets /usr/src/webapp/assets

COPY data.json /usr/src/webapp/
COPY server.js /usr/src/webapp/
```

```
COPY deploy-package.json /usr/src/webapp/package.json
```

```
WORKDIR /usr/src/webapp
```

```
RUN echo 'package-lock=false' >> .npmrc
```

```
RUN npm install
```

```
EXPOSE 4000
```

```
CMD ["node", "server.js"]
```

Zawartość pliku *Dockerfile* wykorzystuje obraz bazowy skonfigurowany z Node.js i zawierający skopiowane pliki niezbędne do uruchomienia aplikacji, m.in. plik paczki z aplikacją oraz plik używany do zainstalowania pakietów Node.js wymaganych do uruchomienia aplikacji w środowisku produkcyjnym.

Aby przyspieszyć proces tworzenia kontenera, do katalogu *webapp* warto również dodać plik *.dockerignore* o zawartości przedstawionej na listingu 16.31. W omawianym przykładzie ten plik nakazuje Dockerowi zignorowanie katalogu *node_modules*, który nie jest wymagany w kontenerze, a jego przetworzenie będzie wymagało ogromnej ilości czasu.

Listing 16.31. Zawartość pliku *.dockerignore* w katalogu *webapp*

```
node_modules
```

Z poziomu katalogu *webapp* w powłoce wydaj polecenie przedstawione na listingu 16.32, które rozpocznie proces tworzenia kontenera zawierającego przykładową aplikację i wszystkie wymagane pakiety.

Listing 16.32. Tworzenie obrazu Dockera

```
$ docker build . -t webapp -f Dockerfile
```

Obraz jest szablonem dla kontenera. Docker przetworzy polecenia zdefiniowane w pliku *Dockerfile*, wskazane pakiety Node.js zostaną pobrane i zainstalowane, a pliki konfiguracyjne i kodu źródłowego będą skopiowane do obrazu.

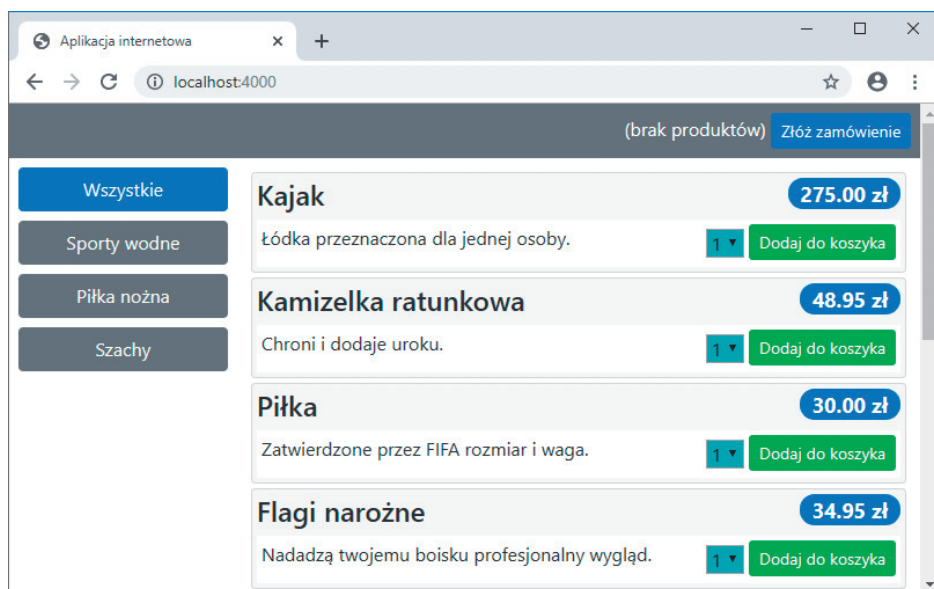
Uruchamianie aplikacji

Gdy obraz Dockera jest gotowy, można uruchomić nowy kontener za pomocą polecenia wymienionego na listingu 16.33.

Listing 16.33. Uruchamianie kontenera Dockera

```
$ docker run -p 4000:4000 webapp
```

Aby przetestować aplikację, przejdź w przeglądarce WWW pod adres *http://localhost:4000*, co spowoduje wyświetlenie odpowiedzi udzielonej przez serwer WWW uruchomiony w kontenerze, jak pokazałem na rysunku 16.9.



Rysunek 16.9. Przykładowa aplikacja uruchomiona w kontenerze

Jeżeli chcesz zatrzymać działanie kontenera, musisz zacząć od wydania polecenia przedstawionego na listingu 16.34.

Listing 16.34. Wyświetlenie listy kontenerów Dockera

```
$ docker ps
```

Dane wyjściowe tego polecenia to lista uruchomionych kontenerów (w celu zachowania zwięzłości i czytelności pominąłem część danych wyjściowych):

CONTAINER ID	IMAGE	COMMAND	CREATED
4b9b82772197	webapp	"docker-entrypoint.s..."	33 seconds ago

Wykorzystując wartość wyświetloną w kolumnie *CONTAINER ID*, wydaj polecenie przedstawione na listingu 16.35.

Listing 16.35. Zatrzymywanie kontenera Dockera

```
$ docker stop 4b9b82772197
```

W tym momencie aplikacja jest gotowa do wdrożenia na dowolnej platformie obsługującej kontenery Dockera.

Podsumowanie

W tym rozdziale dokończyłem pracę nad utworzeniem samodzielnej aplikacji internetowej — dodane zostało źródło danych wykorzystujące usługę sieciową, a także klasy JSX wyświetlające różną treść użytkownikowi. Na końcu zobaczyłeś, jak odbywa się przygotowanie aplikacji do wdrożenia i jak utworzyć obraz kontenera Dockera. W następnym rozdziale zajmę się tworzeniem aplikacji internetowej z wykorzystaniem frameworka Angular.

ROZDZIAŁ 17.



Tworzenie aplikacji internetowej Angulara — część I

W tym rozdziale rozpocznę proces tworzenia aplikacji internetowej Angulara, która będzie miała ten sam zestaw funkcji co przykład zaprezentowany w rozdziałach 15. i 16. W przeciwieństwie do innych frameworków, w których używanie języka TypeScript jest opcjonalne, Angular stawia TypeScript na pierwszym miejscu podczas tworzenia aplikacji internetowej, a także wykorzystuje oferowane przez ten język funkcje, zwłaszcza dekoratory. W tabeli 17.1 wymienię opcje kompilatora TypeScriptu użyte w rozdziale.

Tabela 17.1. Opcje kompilatora TypeScriptu użyte w rozdziale

Opcja	Opis
baseUrl	Ta opcja określa katalog główny używany podczas rozwiązywania zależności modułów
declaration	Ta opcja generuje pliki deklaracji typu dostarczające informacje typu dla kodu JavaScriptu
downlevelIteration	Ta opcja włącza obsługę iteratorów, gdy kod jest przeznaczony dla starszych wersji języka JavaScript
experimentalDecorators	Ta opcja włącza obsługę dekoratorów
importHelpers	Ta opcja określa, czy kod pomocniczy zostanie dodany do JavaScriptu w celu zmniejszenia ogólnej ilości wygenerowanego kodu
lib	Ta opcja pozwala na wybór używanych przez kompilator plików deklaracji typu
module	Ta opcja określa format używany dla modułów
moduleResolution	Ta opcja określa styl używany podczas rozwiązywania zależności modułów
outDir	Ta opcja określa katalog, w którym zostaną umieszczone wygenerowane pliki JavaScriptu
sourceMap	Ta opcja określa, czy kompilator ma generować pliki map źródełowych stosowane podczas debugowania
target	Ta opcja określa docelową wersję języka JavaScript, dla której kompilator wygeneruje kod

Przygotowanie projektu

Projekt Angulara można najłatwiej utworzyć za pomocą pakietu `angular-cli`. Z poziomu powłoki wydaj polecenie przedstawione na listingu 17.1, aby w ten sposób zainstalować niezbędny pakiet.

Listing 17.1. Instalowanie pakietu pozwalającego na utworzenie projektu Angulara

```
$ npm install --global @angular/cli@11.1.1
```

■ **Wskazówka** Przykładowy projekt dla tego rozdziału, a także projekty dla pozostałych rozdziałów książki zostały umieszczone w archiwum dostępnym pod adresem <https://ftp.helion.pl/przyklady/tys4o2.zip>.

Nazwy pakietów Angulara są poprzedzone prefiksem `@`. Po zainstalowaniu pakietu przejdź do katalogu, w którym chcesz utworzyć nowy projekt Angulara, a następnie wydaj polecenie przedstawione na listingu 17.2.

Listing 17.2. Tworzenie nowego projektu

```
$ ng new angularapp
```

Zestaw narzędzi programistycznych Angulara jest używany za pomocą polecenia `ng`, np. `ng new` powoduje utworzenie nowego projektu. W trakcie procesu tworzenia projektu trzeba będzie odpowiedzieć na kilka pytań związanych ze sposobem skonfigurowania projektu. Przygotowując przykładowy projekt dla tego rozdziału, skorzystaj z odpowiedzi przedstawionych w tabeli 17.2.

Tabela 17.2. Pytania i odpowiedzi dotyczące konfiguracji projektu

Pytanie	Odpowiedź
Czy chcesz wymusić ściśle sprawdzanie typów i bardziej rygorystyczną obsługę paczki w przestrzeni roboczej?	Nie
Czy chcesz dodać routing Angulara?	Tak
Jakiego formatu arkuszy stylów chcesz używać?	CSS

Utworzenie projektu może zabrać kilka minut, ponieważ konieczne jest pobranie ogromnej liczby pakietów JavaScriptu.

Konfigurowanie usługi sieciowej

Gdy proces tworzenia projektu się zakończy, wydaj polecenia przedstawione na listingu 17.3, aby przejść do katalogu zawierającego projekt i dodać pakiety dostarczające usługę sieciową oraz zapewniające możliwość uruchamiania wielu pakietów za pomocą pojedynczego polecenia.

Listing 17.3. Dodawanie pakietów do projektu

```
$ cd angularapp
$ npm install --save-dev json-server@0.16.3
$ npm install --save-dev npm-run-all@4.1.5
```

W celu przygotowania danych dla usługi sieciowej dodaj do katalogu *angularapp* plik o nazwie *data.js* z zawartością przedstawioną na listingu 17.4.

Listing 17.4. Zawartość pliku *data.js* w katalogu *angularapp*

```
module.exports = function () {
  return {
    products: [
      { id: 1, name: "Kajak", category: "Sporty wodne",
        description: "Łódka przeznaczona dla jednej osoby.", price: 275 },
      { id: 2, name: "Kamizelka ratunkowa", category: "Sporty wodne",
        description: "Chroni i dodaje uroku.", price: 48.95 },
      { id: 3, name: "Piłka", category: "Piłka nożna",
        description: "Zatwierdzone przez FIFA rozmiar i waga.", price: 19.50 },
      { id: 4, name: "Flagi narożne", category: "Piłka nożna",
        description: "Nadadzą twojemu boisku profesjonalny wygląd.",
        price: 34.95 },
      { id: 5, name: "Stadion", category: "Piłka nożna",
        description: "Składany stadion na 35 000 osób.", price: 79500 },
      { id: 6, name: "Czapka", category: "Szachy",
        description: "Zwiększa efektywność mózgu o 75%.", price: 16 },
      { id: 7, name: "Niestabilne krzesło", category: "Szachy",
        description: "Zmniejsza szanse przeciwnika.",
        price: 29.95 },
      { id: 8, name: "Ludzka szachownica", category: "Szachy",
        description: "Przyjemna gra dla całej rodziny.", price: 75 },
      { id: 9, name: "Błyszczący król", category: "Szachy",
        description: "Pokryty złotem i wysadzany diamentami król.", price: 1200 }
    ],
    orders: []
  }
}
```

Uaktualnij sekcję *scripts* pliku *package.json* w celu skonfigurowania narzędzi programistycznych w taki sposób, aby zestaw narzędzi Angulara i usługa sieciowa były uruchamiane w tym samym momencie. Spójrz na kod zamieszczony na listingu 17.5.

Listing 17.5. Konfigurowanie narzędzi programistycznych w kodzie pliku *package.json* w katalogu *angularapp*

```
...
"scripts": {
  "ng": "ng",
  "json": "json-server data.js -p 4600",
  "serve": "ng serve",
}
```

```

"start": "npm-run-all -p serve json",
"build": "ng build",
"test": "ng test",
"lint": "ng lint",
"e2e": "ng e2e"
},
...

```

Taka konfiguracja pozwala na uruchamianie za pomocą pojedynczego polecenia usługi sieciowej dostarczającej dane i narzędzia programistyczne Angulara.

Konfigurowanie pakietu Bootstrap CSS

Z poziomu katalogu *angularapp* w powłoce wydaj polecenie przedstawione na listingu 17.6, aby do projektu dodać framework Bootstrap CSS.

Listing 17.6. Dodawanie do projektu pakietu Bootstrap CSS

```
$ npm install bootstrap@4.6.0
```

Narzędzia programistyczne Angulara wymagają zmiany konfiguracyjnej pozwalającej na wykorzystanie w aplikacji stylów Bootstrap CSS. Otwórz plik *angular.json* w katalogu *angularapp*, a następnie w sekcji *build/styles* wprowadź zmianę przedstawioną na listingu 17.7.

Listing 17.7. Dodawanie arkusza stylów do pliku *angular.json* w katalogu *angularapp*

```

...
"build": {
  "builder": "@angular-devkit/build-angular:browser",
  "options": {
    "outputPath": "dist/angularapp",
    "index": "src/index.html",
    "main": "src/main.ts",
    "polyfills": "src/polyfills.ts",
    "tsConfig": "src/tsconfig.app.json",
    "assets": [
      "src/favicon.ico",
      "src/assets"
    ],
    "styles": [
      "src/styles.css",
      "node_modules/bootstrap/dist/css/bootstrap.min.css"
    ],
    "scripts": [],
    "es5BrowserSupport": true
  },
  ...

```

-
- **Ostrzeżenie** W pliku *angular.json* znajdują się dwa ustawienia `styles` — upewnij się o zmianie tego w sekcji `build`, a nie w sekcji `test`. Jeżeli po uruchomieniu przykładowej aplikacji zauważysz, że nie korzysta ona ze stylów CSS, prawdopodobnie wprowadziłeś zmianę w niewłaściwej sekcji `styles`.
-

Uruchomienie przykładowej aplikacji

Z poziomu katalogu *angularapp* w powłoce wydaj polecenie przedstawione na listingu 17.8 i tym samym uruchom przykładową aplikację.

Listing 17.8. Uruchamianie narzędzi programistycznych

\$ npm start

Uruchomienie narzędzi programistycznych Angulara zajmie chwilę, a później zostanie przeprowadzona początkowa kompilacja projektu, w trakcie której zostaną wygenerowane następujące dane wyjściowe:

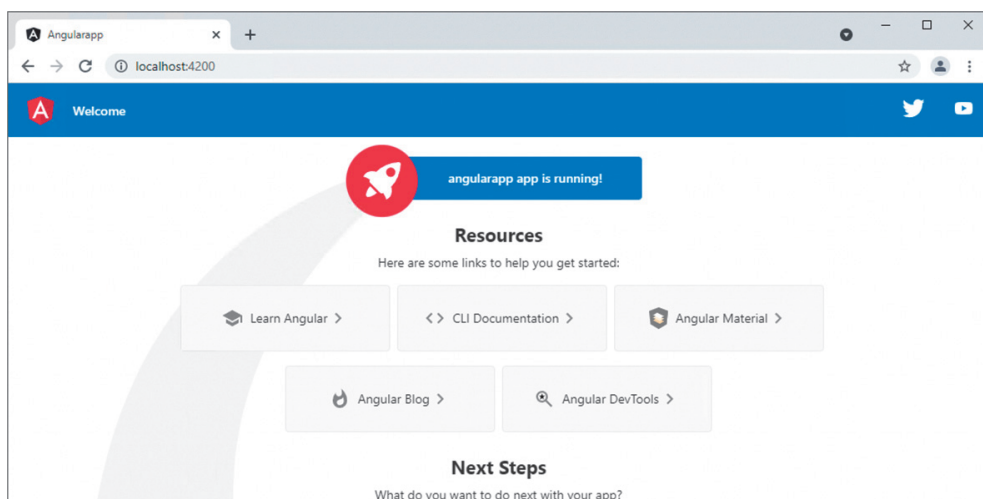
```
...
Compiling @angular/core : es2015 as esm2015
Compiling @angular/common : es2015 as esm2015
Compiling @angular/platform-browser : es2015 as esm2015
Compiling @angular/router : es2015 as esm2015
Compiling @angular/platform-browser-dynamic : es2015 as esm2015
√ Browser application bundle generation complete.

Initial Chunk Files | Names          | Size
vendor.js           | vendor         | 2.68 MB
styles.css, styles.js | styles        | 489.97 kB
polyfills.js        | polyfills      | 472.88 kB
main.js             | main           | 58.50 kB
runtime.js          | runtime        | 6.15 kB
                    | Initial Total  | 3.68 MB

Build at: 2021-01-25T07:18:49.961Z - Hash: 063fb4c85c8d3ffee713 - Time: 19481ms

** Angular Live Development Server is listening on localhost:4200, open your browser on
http://localhost:4200/ **
√ Compiled successfully.
√ Browser application bundle generation complete.
Initial Chunk Files | Names | Size
styles.css, styles.js | styles | 489.97 kB
4 unchanged chunks
Build at: 2021-01-25T07:18:52.776Z - Hash: 49799edf3d51e390dbad - Time: 2350ms
√ Compiled successfully.
...
```

Po zakończeniu początkowej kompilacji otwórz nowe okno przeglądarki WWW i przejdź pod adres *http://localhost:4200*, aby zobaczyć miejsce zarezerwowane na treść przygotowane przez polecenie na listingu 17.2, jak pokazałem na rysunku 17.1.



Rysunek 17.1. Wynik uruchomienia przykładowej aplikacji

Rola TypeScriptu w programowaniu z użyciem frameworka Angular

Framework Angular wykorzystuje przedstawione w rozdziale 15. dekoratory TypeScriptu do opisywania różnych elementów konstrukcyjnych niezbędnych podczas tworzenia aplikacji internetowej. Spójrz na zawartość pliku `app.module.ts` w katalogu `src/app`, a zobaczysz jeden z modułów używanych przez Angulara:

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';

@NgModule({
  declarations: [AppComponent],
  imports: [BrowserModule, AppRoutingModule],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Dekoratory mają ważne znaczenie podczas programowania z użyciem frameworka Angular — aby ułatwić programiście pracę podczas definiowania bądź konfigurowania aplikacji, są stosowane dla klas zawierających niewiele elementów składowych lub nawet ich pozbawionych. Na przykład dekorator `NgModule` jest stosowany do opisanie grupy powiązanych ze sobą funkcjonalności w aplikacji Angulara (moduły Angulara istnieją obok tradycyjnych modułów JavaScriptu i dlatego też plik kodu źródłowego zawiera zarówno polecenia `import`, jak i dekorator `NgModule`).

Kolejny przykład użycia dekoratorów można zobaczyć w pliku *app.component.ts* w katalogu *src/app*.

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'angularapp';
}
```

Jest to dekorator `Component` opisujący klasę, która będzie generowała treść HTML-a, podobnie jak w przypadku klas JSX utworzonych w aplikacji internetowej opracowywanej w rozdziałach 15. i 16.

Rola TypeScriptu w łańcuchu narzędzi Angulara

Łańcuch narzędzi dla frameworka Angular jest podobny do użytego w rozdziałach 15. i 16.: opiera się na pakietach `webpack` i serwerze `Webpack Development Server` z ustawieniami niezbędnymi dla Angulara. Ślady obecności pakietu `webpack` można dostrzec w komunikatach generowanych przez narzędzia programistyczne Angulara, choć szczegóły — i plik konfiguracyjny — nie są bezpośrednio udostępniane. Możesz zobaczyć i zmodyfikować konfigurację kompilatora TypeScriptu, ponieważ projekt został utworzony z plikiem *tsconfig.json*, którego zawartość przedstawia się następująco:

```
{
  "compileOnSave": false,
  "compilerOptions": {
    "baseUrl": "./",
    "outDir": "./dist/out-tsc",
    "sourceMap": true,
    "declaration": false,
    "downlevelIteration": true,
    "experimentalDecorators": true,
    "moduleResolution": "node",
    "importHelpers": true,
    "target": "es2015",
    "module": "es2020",
    "lib": ["es2018", "dom"]
  },
  "angularCompilerOptions": {
    "enableI18nLegacyMessageIdFormat": false
  }
}
```

Zgodnie z tą konfiguracją skompilowane pliki JavaScriptu będą umieszczane w katalogu *dist/out-src*, mimo to wymieniony katalog nie znajduje się w projekcie, ponieważ narzędzie `webpack` automatycznie utworzy odpowiedni plik paczki.

Najważniejsze ustawienie konfiguracyjne to `experimentalDecorators` włączające obsługę dekoratorów w plikach JavaScriptu wygenerowanych przez kompilator, o czym wspomniałem już

w rozdziale 16. Ta funkcjonalność — bardziej niż jakakolwiek inna oferowana przez TypeScript — ma istotne znaczenie podczas programowania z wykorzystaniem frameworka Angular.

■ **Ostrzeżenie** Należy zachować ostrożność podczas wprowadzania zmian w pliku *tsconfig.json*, ponieważ mogą one uszkodzić łańcuch narzędzi Angulara. Większość zmian w projekcie Angulara jest wprowadzana za pomocą pliku *angular.json*.

Tworzenie modelu danych

Pracę nad modelem danych rozpocznij od utworzenia katalogu *src/app/data*, a następnie dodaj do niego plik o nazwie *entities.ts* z kodem przedstawionym na listingu 17.9.

Listing 17.9. Zawartość pliku *entities.ts* w katalogu *src/app/data*

```
export type Product = {
  id: number,
  name: string,
  description: string,
  category: string,
  price: number
};

export class OrderLine {
  constructor(public product: Product, public quantity: number) {
    // Polecenia nie są wymagane.
  }

  get total(): number {
    return this.product.price * this.quantity;
  }
}

export class Order {
  private lines = new Map<number, OrderLine>();

  constructor(initialLines?: OrderLine[]) {
    if (initialLines) {
      initialLines.forEach(ol => this.lines.set(ol.product.id, ol));
    }
  }

  public addProduct(prod: Product, quantity: number) {
    if (this.lines.has(prod.id)) {
      if (quantity === 0) {
        this.removeProduct(prod.id);
      } else {
        this.lines.get(prod.id)!.quantity += quantity;
      }
    } else {
      this.lines.set(prod.id, new OrderLine(prod, quantity));
    }
  }
}
```

```

    }
  }

  public removeProduct(id: number) {
    this.lines.delete(id);
  }

  get orderLines(): OrderLine[] {
    return [...this.lines.values()];
  }

  get productCount(): number {
    return [...this.lines.values()]
      .reduce((total, ol) => total += ol.quantity, 0);
  }

  get total(): number {
    return [...this.lines.values()].reduce((total, ol) => total += ol.total, 0);
  }
}

```

Ten sam kod źródłowy został użyty w rozdziale 15. i nie trzeba wprowadzać w nim żadnych zmian, ponieważ Angular wykorzystuje zwykle klasy TypeScriptu dla encji modelu danych.

Tworzenie źródła danych

W celu utworzenia źródła danych należy dodać do katalogu *src/app/data* plik o nazwie *dataSource.ts* i zawartości przedstawionej na listingu 17.10.

Listing 17.10. Zawartość pliku *dataSource.ts* w katalogu *src/app/data*

```

import { Observable } from "rxjs";
import { Injectable } from '@angular/core';
import { Product, Order } from "../entities";

export type ProductProp = keyof Product;

export abstract class DataSourceImpl {
  abstract loadProducts(): Observable<Product[]>;
  abstract storeOrder(order: Order): Observable<number>;
}

@Injectable()
export class DataSource {
  private _products: Product[];
  private _categories: Set<string>;
  public order: Order;

  constructor(private impl: DataSourceImpl) {
    this._products = [];
    this._categories = new Set<string>();
    this.order = new Order();
    this.getData();
  }
}

```



```

getProducts(sortProp: ProductProp = "id", category?: string): Product[] {
    return this.selectProducts(this._products, sortProp, category);
}

protected getData(): void {
    this._products = [];
    this._categories.clear();
    this.impl.loadProducts().subscribe(rawData => {
        rawData.forEach(p => {
            this._products.push(p);
            this._categories.add(p.category);
        });
    });
}

protected selectProducts(prods: Product[], sortProp: ProductProp,
    category?: string): Product[] {
    return prods.filter(p => category === undefined || p.category === category)
        .sort((p1, p2) => p1[sortProp] < p2[sortProp]
            ? -1 : p1[sortProp] > p2[sortProp] ? 1: 0);
}

getCategories(): string[] {
    return [...this._categories.values()];
}

storeOrder(): Observable<number> {
    return this.impl.storeOrder(this.order);
}
}

```

Usługi to jedna z kluczowych funkcjonalności podczas tworzenia aplikacji z użyciem frameworka Angular. Usługi pozwalają klasom na deklarowanie zależności w konstruktorach, które następnie są rozwiązywane w trakcie działania aplikacji — ta technika jest znana jako *wstrzykiwanie zależności*. Klasa `DataSource` deklaruje w konstruktorze zależność od obiektu `DataSourceImpl`:

```

...
constructor(private impl: DataSourceImpl) {
...

```

Gdy potrzebny jest nowy obiekt `DataSource`, Angular będzie analizować konstruktor, utworzy obiekt `DataSourceImpl` i wykorzysta go do wywołania konstruktora tworzącego nowy obiekt — ten proces jest określany mianem *wstrzykiwania*. Dekorator `Injectable` wskazuje Angularowi, że inne klasy mogą deklarować zależność od klasy `DataSource`. Klasa `DataSourceImpl` jest abstrakcyjna, a klasa `DataSource` nie zawiera żadnych informacji o tym, która z konkretnych implementacji klasy zostanie użyta do rozwiązania zależności konstruktora. Wybór implementacji klas odbywa się w konfiguracji aplikacji, jak pokazałem na listingu 17.12 w dalszej części rozdziału.

Jedną z najważniejszych zalet użycia frameworka podczas tworzenia aplikacji internetowej jest automatyczna obsługa uaktualnień. Angular używa biblioteki `Reactive Extensions`, nazywanej `RxJS`, do zarządzania uaktualnieniami, co pozwala na automatyczną obsługę zmian w danych. Klasa `RxJS Observable` jest używana do opisywania sekwencji wartości generowanych na przestrzeni czasu, łącznie z czynnościami asynchronicznymi, takimi jak żądanie danych z usługi

sieciowej. Metoda `loadProducts()` zdefiniowana przez klasę `DataSourceImpl` zwraca obiekt `Observable<Product[]>`, np.:

```
...
abstract loadProducts(): Observable<Product[]>;
...
```

Argument typu generycznego TypeScriptu jest używany do określenia, że wynikiem działania metody `loadProducts()` jest obiekt `Observable`, który wygeneruje sekwencję obiektów tablicy `Product`. Wartości generowane przez obiekt `Observable` są otrzymywane za pomocą metody `subscribe()`, jak pokazałem w kolejnym fragmencie kodu:

```
...
this.impl.loadProducts().subscribe(rawData => {
  rawData.forEach(p => {
    this._products.push(p);
    this._categories.add(p.category);
  });
});
...
```

W omawianym przykładzie używam klasy `Observable` jako bezpośredniego zamiennika dla standardowej klasy JavaScriptu `Promise`. Klasa `Observable` zapewnia funkcje zaawansowane, przeznaczone do pracy ze skompilowanymi sekwencjami, a jej dużą zaletą jest to, że Angular będzie uaktualniać treść wyświetlaną użytkownikowi po wygenerowaniu treści przez `Observable`. Oznacza to możliwość zdefiniowania pozostałej części klasy `DataSource` bez konieczności zajmowania się obsługą zadań asynchronicznych.

Tworzenie implementacji klasy źródła danych

Aby rozbudować klasę `DataSourceImpl` o możliwość pracy z usługą sieciową, należy dodać do katalogu `src/app/data` plik o nazwie `remoteDataSource.ts` z kodem przedstawionym na listingu 17.11.

Listing 17.11. Zawartość pliku `remoteDataSource.ts` w katalogu `src/app/data`

```
import { Injectable } from "@angular/core";
import { HttpClient } from "@angular/common/http";
import { Observable } from "rxjs";
import { map } from "rxjs/operators";
import { DataSourceImpl } from "../dataSource";
import { Product, Order } from "../entities";

const protocol = "http";
const hostname = "localhost";
const port = 4600;

const urls = {
  products: `${protocol}://${hostname}:${port}/products`,
  orders: `${protocol}://${hostname}:${port}/orders`
};
```

```

@Injectable()
export class RemoteDataSource extends DataSourceImpl {

  constructor(private http: HttpClient) {
    super();
  }

  loadProducts(): Observable<Product[]> {
    return this.http.get<Product[]>(urls.products);
  }

  storeOrder(order: Order): Observable<number> {
    let orderData = {
      lines: [...order.orderLines.values()].map(ol => ({
        productId: ol.product.id,
        productName: ol.product.name,
        quantity: ol.quantity
      })))
    return this.http.post<{ id: number}>(urls.orders, orderData)
      .pipe<number>(map(val => val.id));
  }
}

```

Konstruktor `RemoteDataSource` deklaruje zależność od egzemplarza klasy `HttpClient`, która jest wbudowaną klasą Angulara przeznaczoną do wykonywania żądań HTTP. Klasa `HttpClient` definiuje metody `get()` i `post()` używane do wykonywania żądań HTTP GET i POST. Oczekiwany typ danych jest podawany jako argument typu:

```

...
loadProducts(): Observable<Product[]> {
  return this.http.get<Product[]>(urls.products);
}
...

```

Typ argumentu jest używany dla wyniku działania metody `get()` — jest to obiekt typu `Observable`, który wygeneruje sekwencję podanego typu (w omawianym przykładzie `Product[]`).

■ **Wskazówka** Argumenty typu generycznego dla metod `HttpClient` to standardowe argumenty TypeScriptu. W tle nie mamy do czynienia z żadną magią frameworka Angular, a programista jest odpowiedzialny za określenie typu, który będzie odpowiadał danym otrzymanym z serwera.

Biblioteka RxJS zawiera funkcjonalność, która może być używana do przeprowadzania operacji na wartościach wygenerowanych przez obiekt `Observable`. Część z nich została użyta na listingu 17.11:

```

...
return this.http.post<{ id: number}>(urls.orders, orderData)
  .pipe<number>(map(val => val.id));
...

```

Metoda `pipe()` została wykorzystana z funkcją `map()` do utworzenia obiektu `Observable` generującego wartości na podstawie danych pochodzących z innego obiektu `Observable`. Pozwala to na otrzymanie wyniku żądania HTTP POST i wyodrębnienie po prostu właściwości `id` z wyniku.

■ **Uwaga** W aplikacji internetowej wykorzystującej jedynie TypeScript utworzyłem klasę abstrakcyjną źródła danych, a następnie podklasy dostarczające dane lokalne lub pochodzące z usługi sieciowej — były one wczytywane za pomocą metody wywoływanej w konstruktorze klasy abstrakcyjnej. Takie podejście nie sprawdza się zbyt dobrze w przypadku frameworka Angular, ponieważ klasa `HttpClient` nie jest przypisana właściwości egzemplarza aż do chwili wywołania konstruktora klasy abstrakcyjnej za pomocą słowa kluczowego `super`. Oznacza to, że podklasa będzie poproszona o dostarczenie danych, jeszcze zanim zostanie prawidłowo przygotowana. Aby uniknąć takiego problemu, zdecydowałem się na umieszczenie w klasie abstrakcyjnej tej części źródła danych, która zajmuje się obsługą danych.

Konfigurowanie źródła danych

Ostatnimi krokami podczas tworzenia źródła danych są przygotowanie modułu Angulara, dzięki któremu źródło danych stanie się dostępne do użycia w pozostałej części aplikacji, oraz wybór implementacji klasy abstrakcyjnej `DataSourceImpl`. Do katalogu `src/app/data` należy dodać plik `data.module.ts` z kodem przedstawionym na listingu 17.12.

Listing 17.12. Zawartość pliku `data.module.ts` w katalogu `src/app/data`

```
import { NgModule } from '@angular/core';
import { HttpClientModule } from '@angular/common/http';
import { DataSource, DataSourceImpl } from './dataSource';
import { RemoteDataSource } from './remoteDataSource';

@NgModule({
  imports: [HttpClientModule],
  providers: [DataSource, { provide: DataSourceImpl, useClass: RemoteDataSource }]
})
export class DataModelModule { }
```

Klasa `DataModelModule` została zdefiniowana, więc można zastosować dekorator `NgModule`. Właściwość `imports` dekoratora definiuje zależności wymagane do spełnienia przez klasy modelu danych, natomiast właściwość `providers` definiuje klasy w module Angulara, które mogą być wstrzykiwane do konstruktorów innych klas aplikacji. W przypadku tego modułu właściwość `imports` wskazuje frameworkowi Angular, że wymagany jest moduł zawierający klasę `HttpClient`, natomiast właściwość `providers` wskazuje Angularowi, że klasa `DataSource` może być używana w przypadku mechanizmu wstrzykiwania zależności, a zależności klasy `DataSourceImpl` powinny być rozwiązane za pomocą klasy `RemoteDataSource`.

Wyświetlenie filtrowanej listy produktów

Za generowanie treści HTML-a odpowiada w Angularze kod umieszczony w dwóch plikach: klasa TypeScriptu z dekoratorem `Component` i szablon HTML-a z dyrektywami bezpośrednio związanymi z generowaniem treści dynamicznej. Po uruchomieniu aplikacji następuje kompilacja szablonu HTML-a, a dyrektywy są wykonywane z wykorzystaniem metod i właściwości dostarczonych przez klasę TypeScriptu.

Klasy, do których jest stosowany dekorator `Component`, są, całkiem logicznie, określane mianem *komponentów*. Zgodnie z konwencją podczas programowania z użyciem frameworka Angular, rola klasy jest określana w nazwie pliku. Dlatego też w celu utworzenia komponentu odpowiedzialnego za wyświetlenie użytkownikowi szczegółów o pojedynczym produkcie należy dodać do katalogu *src/app* plik o nazwie *productItem.component.ts* zawierający kod przedstawiony na listingu 17.13.

Listing 17.13. Zawartość pliku *productItem.component.ts* w katalogu *src/app*

```
import { Component, Input, Output, EventEmitter } from "@angular/core";
import { Product } from '../data/entities';

export type productSelection = {
  product: Product,
  quantity: number
}

@Component({
  selector: "product-item",
  templateUrl: "../productItem.component.html"
})
export class ProductItem {
  quantity: number = 1;

  @Input()
  product: Product;

  @Output()
  addToCart = new EventEmitter<productSelection>();

  handleAddToCart() {
    this.addToCart.emit({ product: this.product,
      quantity: Number(this.quantity)});
  }
}
```

Dekorator `Component` odpowiada za konfigurację komponentu. Właściwość `selector` określa selektor CSS, który będzie przez Angulara używany do zastosowania komponentu do kodu HTML-a aplikacji, natomiast właściwość `templateUrl` określa szablon HTML-a komponentu. W przypadku klasy `ProductItem` właściwość `selector` nakazuje frameworkowi Angular zastosowanie komponentu po napotkaniu elementu `product-item`, a szablon HTML-a komponentu znajduje się w pliku *productItem.component.html* w tym samym katalogu, w którym umieszczono plik TypeScriptu.

Angular używa dekoratora `Input` do wskazania właściwości pozwalających komponentom na otrzymywanie wartości za pomocą atrybutów elementów HTML-a. Dekorator `Output` jest używany do wskazania przepływu danych z komponentu poprzez niestandardowe zdarzenie. Klasa `ProductItem` otrzymuje obiekt `Product`, którego szczegóły są wyświetlane użytkownikowi, i gdy użytkownik kliknie przycisk myszą, wywołuje niestandardowe zdarzenie, dostępne poprzez właściwość `addToCart`.

Aby utworzyć szablon komponentu, należy dodać do katalogu `src/app` plik o nazwie `productItem.component.html` z kodem przedstawionym na listingu 17.14.

Listing 17.14. Zawartość pliku `productItem.component.html` w katalogu `src/app`

```
<div class="card m-1 p-1 bg-light">
  <h4>
    {{ product.name }}
    <span class="badge badge-pill badge-primary float-right">
      {{ product.price.toFixed(2) }} zł
    </span>
  </h4>
  <div class="card-text bg-white p-1">
    {{ product.description }}
    <button class="btn btn-success btn-sm float-right"
      (click)="handleAddToCart()">
      Dodaj do koszyka
    </button>
    <select class="form-control-inline float-right m-1" [(ngModel)]="quantity">
      <option>1</option>
      <option>2</option>
      <option>3</option>
    </select>
  </div>
</div>
```

Szablon Angulara używa podwójnych nawiasów klamrowych do wyświetlania wyniku działania wyrażenia JavaScriptu, jak pokazałem w kolejnym fragmencie kodu:

```
...
<span class="badge badge-pill badge-primary float-right">
  {{ product.price.toFixed(2) }} zł
</span>
...
```

Wyrażenie jest wykonywane w kontekście komponentu, więc w tym przykładzie następuje odczytanie wartości właściwości `product.price`, wywołanie metody `toFixed()` i wstawienie wygenerowanego wyniku do elementu ``.

Obsługa zdarzeń jest oznaczana przez ujęcie w nawias nazwy zdarzenia, np.:

```
...
<button class="btn btn-success btn-sm float-right" (click)="handleAddToCart()">
...
```

Ten fragment kodu informuje Angulara, że jeśli element `<button>` wyemituje zdarzenie `click`, wówczas powinna być wywołana metoda `handleAddToCart()` komponentu. Elementy formularza mają zapewnioną specjalną obsługę przez framework Angular, o czym możesz się przekonać na przykładzie elementu `<select>`.

```
...
<select class="form-control-inline float-right m-1" [(ngModel)]=>"quantity">
...
```

Dyrektywa `ngModel` jest zastosowana bezpośrednio w nawiasach kwadratowym i okrągłym oraz tworzy dwukierunkowe wiązanie między elementem `<select>` a właściwością `quantity` komponentu. Zmiana właściwości `quantity` będzie odzwierciedlona przez element `<select>`, natomiast wartość wybrana w elemencie `<select>` zostanie przypisana właściwości `quantity`.

Wyświetlanie przycisków kategorii

W celu utworzenia komponentu wyświetlającego listę przycisków kategorii należy dodać do katalogu `src/app` plik o nazwie `categoryList.component.ts` z kodem przedstawionym na listingu 17.15.

Listing 17.15. Zawartość pliku `categoryList.component.ts` w katalogu `src/app`

```
import { Component, Input, Output, EventEmitter } from "@angular/core";

@Component({
  selector: "category-list",
  templateUrl: "./categoryList.component.html"
})
export class CategoryList {

  @Input()
  selected: string

  @Input()
  categories: string[];

  @Output()
  selectCategory = new EventEmitter<string>();

  getBtnClass(category: string): string {
    return "btn btn-block " +
      (category === this.selected ? "btn-primary" : "btn-secondary");
  }
}
```

Komponent `CategoryList` ma właściwości `Input` otrzymujące aktualnie wybraną kategorię oraz listę kategorii do wyświetlenia. Dekorator `Output` jest zastosowany do właściwości `selectCategory` i definiuje niestandardowe zdarzenie wywoływane po dokonaniu wyboru przez użytkownika. Metoda `getBtnClass()` to metoda pomocnicza — jej działanie polega na zwrocie listy klas Bootstrap, które powinny być przypisane elementowi `<button>`, i pomaga ona uniknąć umieszczenia skomplikowanych wyrażeń w szablonie komponentu. W celu utworzenia szablonu dla omawianego komponentu należy dodać do katalogu `src/app` plik o nazwie `categoryList.component.html` z kodem przedstawionym na listingu 17.16.

Listing 17.16. Zawartość pliku `categoryList.component.html` w katalogu `src/app`

```
<button *ngFor="let cat of categories" [class]="getBtnClass(cat)"
  (click)="selectCategory.emit(cat)">
  {{ cat }}
</button>
```

Ten szablon używa dyrektywy `ngFor` do wygenerowania elementu `<button>` dla poszczególnych wartości zwracanych przez właściwości `categories`. Gwiazdka poprzedzająca dyrektywę `ngFor` wskazuje na zwięzłą składnię pozwalającą na bezpośrednie zastosowanie dyrektywy do wygenerowanego elementu.

Szablony Angulara wykorzystują nawias kwadratowy do utworzenia jednokierunkowego wiązania między atrybutem a wartością danych, np.:

```
...
<button *ngFor="let cat of categories" [class]="getBtnClass(cat)"
      (click)="selectCategory.emit(cat)">
...

```

Nawias kwadratowy pozwala na przypisanie wartości atrybutu `class` za pomocą wyrażenia JavaScriptu będącego wynikiem wywołania metody `getBtnClass()` komponentu.

Utworzenie nagłówka

W celu utworzenia komponentu wyświetlającego nagłówki zawierający podsumowanie produktów wybranych przez użytkownika i zapewniający możliwość przejścia do podsumowania zamówienia należy dodać do katalogu `src/app` plik o nazwie `header.component.ts` z kodem przedstawionym na listingu 17.17.

Listing 17.17. Zawartość pliku `header.component.ts` w katalogu `src/app`

```
import { Component, Input, Output, EventEmitter } from "@angular/core";
import { Order } from '../data/entities';

@Component({
  selector: "header",
  templateUrl: "../header.component.html"
})
export class Header {

  @Input()
  order: Order;

  @Output()
  submit = new EventEmitter<void>();

  get headerText(): string {
    let count = this.order.productCount;
    return count === 0 ? "(brak produktów)"
      : `Liczba produktów: ${count}, ${this.order.total.toFixed(2)} zł`
  }
}
```

Natomiast aby utworzyć szablon komponentu podsumowania, należy dodać do katalogu `src/app` plik o nazwie `header.component.html` z kodem przedstawionym na listingu 17.18.

Listing 17.18. Zawartość pliku *header.component.html* w katalogu *src/app*

```
<div class="p-1 bg-secondary text-white text-right">
  {{ headerText }}
  <button class="btn btn-sm btn-primary m-1" (click)="submit.emit()">
    Złóż zamówienie
  </button>
</div>
```

Połączenie komponentów produktu, kategorii i nagłówka

Zdefiniowanie komponentu przedstawiającego komponenty *ProductItem*, *CategoryList* i *Header* wymaga dodania do katalogu *src/app* pliku o nazwie *productList.component.ts* zawierającego kod przedstawiony na listingu 17.19.

Listing 17.19. Zawartość pliku *productList.component.ts* w katalogu *src/app*

```
import { Component } from "@angular/core";
import { DataSource } from '../data/dataSource';
import { Product } from '../data/entities';

@Component({
  selector: "product-list",
  templateUrl: "../productList.component.html"
})
export class ProductList {
  selectedCategory = "Wszystkie";

  constructor(public dataSource: DataSource) {}

  get products(): Product[] {
    return this.dataSource.getProducts("id",
      this.selectedCategory === "Wszystkie" ? undefined : this.selectedCategory);
  }

  get categories(): string[] {
    return ["Wszystkie", ...this.dataSource.getCategories()];
  }

  handleCategorySelect(category: string) {
    this.selectedCategory = category;
  }

  handleAdd(data: {product: Product, quantity: number}) {
    this.dataSource.order.addProduct(data.product, data.quantity);
  }

  handleSubmit() {
    console.log("WYŚLIJ");
  }
}
```

Klasa `ProductList` deklaruje zależność od klasy `DataSource` oraz definiuje metody `products()` i `categories()` zwracające dane z `DataSource`. Są trzy metody odpowiadające na działania użytkownika: `handleCategorySelect()` wywoływana po kliknięciu przez użytkownika przycisku wyboru kategorii, `handleAdd()` wywoływana po dodaniu przez użytkownika produktu do zamówienia i `handleSubmit()` wywoływana, gdy użytkownik chce przejść do podsumowania zamówienia. Metoda `handleSubmit()` na razie jedynie wyświetla komunikat w konsoli, a jej pełną implementacją zajmę się w rozdziale 18.

Aby utworzyć szablon dla omawianego komponentu, należy dodać do katalogu `src/app` plik o nazwie `productList.component.html` i zawartości przedstawionej na listingu 17.20.

Listing 17.20. Zawartość pliku `productList.component.html` w katalogu `src/app`

```
<header [order]="dataSource.order" (submit)="handleSubmit()"></header>
<div class="container-fluid">
  <div class="row">
    <div class="col-3 p-2">
      <category-list [selected]="selectedCategory" [categories]="categories"
        (selectCategory)="handleCategorySelect($event)"></category-list>
    </div>
    <div class="col-9 p-2">
      <product-item *ngFor="let p of products" [product]="p"
        (addToCart)="handleAdd($event)"></product-item>
    </div>
  </div>
</div>
```

Ten szablon pokazuje możliwość połączenia komponentów w celu wyświetlenia treści użytkownikowi. Niestandardowe elementy HTML-a o znacznikach odpowiadających właściwościom selector w dekoratorach `Component` zostały zastosowane dla klas zdefiniowanych na listingach we wcześniejszej części rozdziału, np.:

```
...
<header [order]="dataSource.order" (submit)="handleSubmit()"></header>
...
```

Znacznik `<header>` odpowiada ustawieniu właściwości `selector` dekoratora `Component` zastosowanego w klasie `Header`, której kod przedstawiłem na listingu 17.17. Atrybut `order` jest używany w celu dostarczenia wartości dla właściwości `Input` zdefiniowanej przez klasę `Header` i pozwala egzemplarzowi `ProductList` na dostarczenie obiektu `Header` z wymaganymi danymi. Atrybut `submit` odpowiada właściwości `Output` zdefiniowanej przez klasę `Header` i pozwala egzemplarzowi `ProductList` na otrzymywanie powiadomień. Szablon `ProductList` wykorzystuje elementy `header`, `category-list` i `product-item` do wyświetlenia komponentów odpowiednio `Header`, `CategoryList` i `ProductItem`.

Konfigurowanie aplikacji

Moduł aplikacji jest wykorzystywany do zarejestrowania zarówno komponentów używanych przez aplikację, jak i wszystkich dodatkowych modułów, które zostały zdefiniowane — np. moduł utworzony dla modelu danych we wcześniejszej części rozdziału. Na listingu 17.21 przedstawiłem zmiany konieczne do wprowadzenia w module aplikacji, który został zdefiniowany w pliku `app.module.ts`.

Listing 17.21. Konfigurowanie modułu w pliku *app.module.ts* w katalogu *src/app*

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { FormsModule } from '@angular/forms';
import { DataModelModule } from './data/data.module';
import { ProductItem } from './productItem.component';
import { CategoryList } from './categoryList.component';
import { Header } from './header.component';
import { ProductList } from './productList.component';

@NgModule({
  declarations: [AppComponent, ProductItem, CategoryList, Header, ProductList],
  imports: [BrowserModule, AppRoutingModule, FormsModule, DataModelModule],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Właściwość `declarations` dekoratora `NgModule` została użyta do zadeklarowania komponentów wymaganych przez aplikację i jest stosowana w celu dodania klas zdefiniowanych we wcześniejszej części rozdziału. Z kolei właściwość `imports` została użyta do wymienienia listy pozostałych modułów niezbędnych aplikacji i jest uaktualniona, aby zawierała również moduł modelu danych zdefiniowany na listingu 17.12.

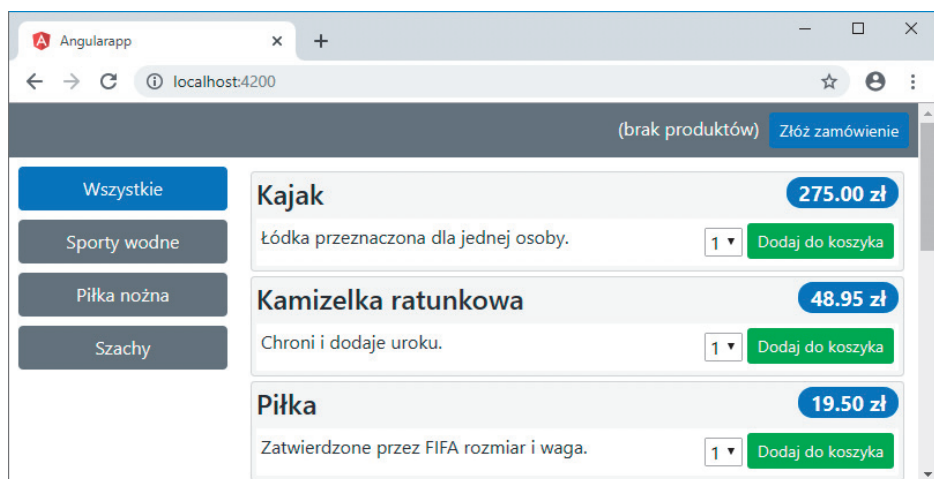
Wyświetlenie nowych komponentów użytkownikowi wymaga zastąpienia zawartości pliku *app.component.html* pojedynczym elementem przedstawionym na listingu 17.22.

Listing 17.22. Nowa zawartość pliku *app.component.html* w katalogu *src/app*

```
<product-list></product-list>
```

Po uruchomieniu aplikacji Angular napotka element `product-list` i porówna go z właściwością `selector` dekoratora `Component` skonfigurowanego za pomocą modułu Angulara. Znacznik `<product-list>` odpowiada właściwości `selector` dekoratora `Component` zastosowanego dla klasy `ProductList` zdefiniowanej na listingu 17.19. Framework Angular tworzy nowy obiekt `ProductList`, generuje zawartość jego szablonu, a następnie wstawia te dane do elementu `<product-list>` zdefiniowanego na listingu 17.22. Kod HTML-a wygenerowany przez komponent `ProductList` zostaje przeanalizowany, a elementy `header`, `category-list` i `product-item` będą odkryte, co prowadzi do utworzenia egzemplarzy odpowiadających im komponentów i wstawienia treści do poszczególnych elementów. Ten proces jest powtarzany aż do chwili przetworzenia wszystkich elementów odpowiadających komponentom oraz wyświetlenia treści użytkownikowi, jak pokazałem na rysunku 17.2.

Użytkownik może filtrować listę produktów i dodawać produkty do zamówienia. Kliknięcie przycisku *Złóż zamówienie* powoduje jedynie wyświetlenie komunikatu w konsoli JavaScriptu przeglądarki WWW. Obsługa pozostałej funkcjonalności aplikacji zostanie dodana w następnym rozdziale.



Rysunek 17.2. Przykładowa aplikacja Angulara wyświetlająca treść użytkownikowi

Podsumowanie

W tym rozdziale wyjaśniłem rolę języka TypeScript podczas tworzenia aplikacji internetowych z wykorzystaniem frameworka Angular. Dowiedziałeś się, że dekoratory TypeScriptu są używane do opisywania różnych elementów konstrukcyjnych stosowanych podczas tworzenia aplikacji Angulara. Omówiłem również sposób kompilacji szablonów HTML-a aplikacji Angulara podczas jej uruchamiania przez przeglądarkę WWW, co oznacza, że funkcjonalności oferowane przez język TypeScript zostały usunięte i nie mogą być używane w szablonach. W następnym rozdziale zajmę się dokończeniem pracy nad aplikacją i przygotowaniem jej do wdrożenia.



Tworzenie aplikacji internetowej Angulara — część II

W rozdziale będę kontynuował pracę nad aplikacją internetową Angulara, której tworzenie rozpocząłem w poprzednim rozdziale. Zajmę się dodaniem pozostałych funkcjonalności i przygotowaniem aplikacji do jej wdrożenia w kontenerze. W tabeli 18.1 wymienilem opcje kompilatora TypeScriptu użyte w rozdziale.

Tabela 18.1. Opcje kompilatora TypeScriptu użyte w rozdziale

Opcja	Opis
baseUrl	Ta opcja określa katalog główny używany podczas rozwiązywania zależności modułów
declaration	Ta opcja generuje pliki deklaracji typu dostarczające informacje typu dla kodu JavaScriptu
downlevelIteration	Ta opcja włącza obsługę iteratorów, gdy kod jest przeznaczony dla starszych wersji języka JavaScript
emitDecoratorMetadata	Ta opcja powoduje dołączenie metadanych dekoratora w kodzie JavaScriptu wyemitowanym przez kompilator
experimentalDecorators	Ta opcja włącza obsługę dekoratorów
importHelpers	Ta opcja określa, czy kod pomocniczy zostanie dodany do JavaScriptu w celu zmniejszenia ogólnej ilości wygenerowanego kodu
lib	Ta opcja pozwala na wybór używanych przez kompilator plików deklaracji typu
module	Ta opcja określa format używany dla modułów
moduleResolution	Ta opcja określa styl używany podczas rozwiązywania zależności modułów
outDir	Ta opcja określa katalog, w którym zostaną umieszczone wygenerowane pliki JavaScriptu

Tabela 18.1. Opcje kompilatora TypeScriptu użyte w rozdziale (ciąg dalszy)

Opcja	Opis
sourceMap	Ta opcja określa, czy kompilator ma generować pliki map źródłowych stosowane podczas debugowania
target	Ta opcja określa docelową wersję języka JavaScript, dla której kompilator wygeneruje kod
typeRoots	Ta opcja określa katalog główny, który kompilator będzie sprawdzał w poszukiwaniu plików deklaracji

Przygotowanie projektu

W rozdziale będę kontynuował pracę nad projektem utworzonym w poprzednim rozdziale. Na potrzeby omawianego tutaj materiału nie trzeba wprowadzać żadnych zmian w aplikacji. Otwórz nowe okno powłoki, przejdź do katalogu *angularapp*, a następnie wydaj polecenie przedstawione na listingu 18.1, aby w ten sposób uruchomić usługę sieciową i narzędzia programistyczne Angulara.

Listing 18.1. Uruchamianie narzędzi programistycznych

```
$ npm start
```

- **Wskazówka** Przykładowy projekt dla tego rozdziału, a także projekty dla pozostałych rozdziałów książki zostały umieszczone w archiwum dostępnym pod adresem <https://ftp.helion.pl/przyklady/tys4o2.zip>.

Po zakończeniu kompilacji początkowej otwórz nowe okno przeglądarki WWW i przejdź pod adres <http://localhost:4200>, a zobaczysz uruchomioną aplikację, jak pokazałem na rysunku 18.1.

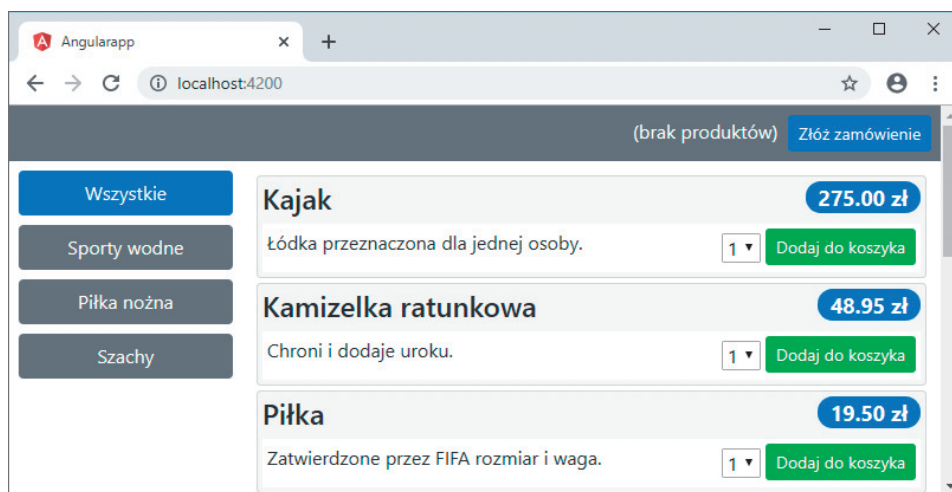
Dokończenie pracy nad funkcjonalnością aplikacji

Potrzebny jest komponent wyświetlający szczegółowe informacje o zamówieniu. Do katalogu *src/app* dodaj plik o nazwie *orderDetails.component.ts* zawierający kod przedstawiony na listingu 18.2.

Listing 18.2. Zawartość pliku *orderDetails.component.ts* w katalogu *src/app*

```
import { Component } from "@angular/core";
import { Router } from "@angular/router";
import { Order } from "../data/entities";
import { DataSource } from '../data/dataSource';
```

```
@Component({
  selector: "order-details",
```



Rysunek 18.1. Przykładowa aplikacja uruchomiona w przeglądarce WWW

```

    templateUrl: "./orderDetails.component.html"
  })
  export class OrderDetails {

    constructor(private dataSource: DataSource, private router: Router) {}

    get order() : Order {
      return this.dataSource.order;
    }

    submit() {
      this.dataSource.storeOrder().subscribe(id =>
        this.router.navigateByUrl(`/summary/${id}`));
    }
  }

```

Komponent `OrderDetails` otrzymuje za pomocą swojego konstruktora obiekt `DataSource` i dostarcza szablonowi właściwość `order`. Z kolei szablon wykorzystuje system routingu URL frameworka Angular do wyboru na podstawie bieżącego adresu URL komponentów wyświetlanych użytkownikowi. W tabeli 18.2 wymienilem obsługiwane przez przykładową aplikację adresy URL i ich przeznaczenie.

Obiekt `Router` otrzymany przez konstruktor `OrderDetails` pozwala komponentowi na używanie funkcjonalności routingu URL do poruszania się po nowych adresach URL i jest stosowany w metodzie `submit()`.

```

...
submit() {
  this.dataSource.storeOrder().subscribe(id =>
    this.router.navigateByUrl(`/summary/${id}`));
}
...

```

Tabela 18.2. Adresy URL obsługiwane przez aplikację

Adres URL	Opis
/products	Ten adres URL powoduje wyświetlenie komponentu ProductList zdefiniowanego w rozdziale 17.
/order	Ten adres URL powoduje wyświetlenie komponentu OrderDetails zdefiniowanego na listingu 18.2
/summary	Ten adres URL powoduje wyświetlenie podsumowania zamówienia po jego przekazaniu do serwera. Adres URL będzie zawierał numer zamówienia, więc zamówienie o identyfikatorze 5 zostanie wyświetlone po użyciu adresu URL /summary/5
/	Domyślny adres URL powoduje przekierowanie pod adres /products, aby został wyświetlony komponent ProductList

Ta metoda wykorzystuje obiekt DataSource w celu przekazania zamówienia użytkownika do serwera, czeka na udzielenie odpowiedzi, a następnie używa metody `navigateByUrl()` obiektu Router do przejścia pod adres URL, który spowoduje wyświetlenie użytkownikowi informacji podsumowujących złożone zamówienie.

Aby utworzyć szablon dla komponentu OrderDetails, dodaj do katalogu `src/app` plik o nazwie `orderDetails.component.html` z kodem przedstawionym na listingu 18.3.

Listing 18.3. Zawartość pliku `orderDetails.component.html` w katalogu `src/app`

```
<h3 class="text-center bg-primary text-white p-2">Informacje o zamówieniu</h3>
<div class="p-3">
  <table class="table table-sm table-striped">
    <thead>
      <tr>
        <th>Ilość</th><th>Produkt</th>
        <th class="text-right">Cena</th>
        <th class="text-right">Wartość</th>
      </tr>
    </thead>
    <tbody>
      <tr *ngFor="let line of order.orderLines">
        <td>{{ line.quantity }}</td>
        <td>{{ line.product.name }}</td>
        <td class="text-right">{{ line.product.price.toFixed(2) }} zł</td>
        <td class="text-right">{{ line.total.toFixed(2) }} zł</td>
      </tr>
    </tbody>
    <tfoot>
      <tr>
        <th class="text-right" colSpan="3">Razem:</th>
        <th class="text-right">
          {{ order.total.toFixed(2) }} zł
        </th>
      </tr>
    </tfoot>
  </table>
</div>
```



```
<div class="text-center">
  <button class="btn btn-secondary m-1" routerLink="/products">Wróć</button>
  <button class="btn btn-primary m-1" (click)="submit()">Złóż zamówienie</button>
</div>
```

Ten komponent wyświetla szczegółowe informacje o produktach wybranych przez użytkownika oraz przyciski pozwalające na wywołanie metody `submit()` i przejście do listy produktów, co powoduje wyświetlenie komponentu `ProductList`. Nawigacja została skonfigurowana przez zastosowanie dyrektywy `routerLink` dla elementu `<button>` oraz podanie adresu URL, pod który ma przejść przeglądarka WWW po kliknięciu elementu.

```
...
<button class="btn btn-secondary m-1" routerLink="/products">Wróć</button>
...
```

Dyrektywa `routerLink` jest częścią funkcjonalności routingu Angulara i pozwala na nawigację bez konieczności stosowania obiektu `Router` w klasie komponentu.

Dodawanie komponentu obsługującego podsumowanie zamówienia

W celu utworzenia komponentu wyświetlanego po przejściu pod adresu URL `/summary` należy dodać do katalogu `src/app` plik o nazwie `summary.component.ts` z kodem przedstawionym na listingu 18.4.

Listing 18.4. Zawartość pliku `summary.component.ts` w katalogu `src/app`

```
import { Component } from "@angular/core";
import { Router, ActivatedRoute } from "@angular/router";

@Component({
  selector: "summary",
  templateUrl: "../summary.component.html"
})
export class Summary {

  constructor(private activatedRoute: ActivatedRoute) {}

  get id(): string {
    return this.activatedRoute.snapshot.params["id"];
  }
}
```

Komponent `Summary` deklaruje zależność od obiektu `ActivatedRoute`, która przez framework Angular zostanie rozwiązana za pomocą mechanizmu wstrzykiwania zależności. Klasa `ActivatedRoute` jest odpowiedzialna za opisanie bieżącej trasy, czyli za opisanie aktualnie aktywnej trasy za pomocą właściwości `snapshot` wymienionej klasy. Komponent `Summary` odczytuje wartość parametru `id` zawierającego identyfikator złożonego zamówienia. Na przykład adres URL w postaci `/summary/5` oznacza, że wartością `id` jest 5. Aby zdefiniować szablon dla tego komponentu, należy dodać do katalogu `src/app` plik o nazwie `summary.component.html` zawierający kod przedstawiony na listingu 18.5.

Listing 18.5. Zawartość pliku *summary.component.html* w katalogu *src/app*

```
<div class="m-2 text-center">
  <h2>Dziękujemy!</h2>
  <p>Dziękujemy za złożenie zamówienia.</p>
  <p>Numer zamówienia #{{ id }}</p>
  <p>Zamówione produkty zostaną wkrótce wysłane.</p>
  <button class="btn btn-primary" routerLink="/products">OK</button>
</div>
```

Szablon wyświetla wartość właściwości `id` pobraną z aktywnej trasy oraz element `<button>`, którego kliknięcie powoduje przejście pod adres URL `/products`.

Tworzenie konfiguracji routingu

Aby opisać adresy URL obsługiwane przez aplikację i komponenty wyświetlające treści po przejściu pod te adresy, konieczne jest wprowadzenie w pliku *app.module.ts* zmian przedstawionych na listingu 18.6 i tym samym przygotowanie konfiguracji routingu.

Listing 18.6. Konfigurowanie aplikacji w pliku *app.module.ts* w katalogu *src/app*

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { FormsModule } from '@angular/forms';
import { DataModelModule } from './data/data.module';
import { ProductItem } from './productItem.component';
import { CategoryList } from './categoryList.component';
import { Header } from './header.component';
import { ProductList } from './productList.component';
import { RouterModule } from '@angular/router';
import { OrderDetails } from './orderDetails.component';
import { Summary } from './summary.component';

const routes = RouterModule.forRoot([
  { path: "products", component: ProductList },
  { path: "order", component: OrderDetails},
  { path: "summary/:id", component: Summary},
  { path: "", redirectTo: "/products", pathMatch: "full"}
]);

@NgModule({
  declarations: [AppComponent, ProductItem, CategoryList, Header, ProductList,
    OrderDetails, Summary],
  imports: [BrowserModule, AppRoutingModule, FormsModule, DataModelModule, routes],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Metoda `RouterModule.forRoot()` jest używana do opisywania adresów URL i komponentów wyświetlanych po przejściu pod te adresy. Pozwala również na przekierowanie do domyślnego adresu URL, czyli `/products`. Aby wskazać frameworkowi Angular to, gdzie mają być wyświetlane komponenty wymienione w konfiguracji routingu, zawartość pliku `app.component.html` należy zastąpić przedstawioną na listingu 18.7.

Listing 18.7. Nowa zawartość pliku `app.component.html` w katalogu `src/app`

```
<router-outlet></router-outlet>
```

Ostatnia zmiana polega na modyfikacji komponentu `ProductList` — jego metoda `submit()` używa teraz funkcjonalności routingu Angulara w celu przejścia pod adres URL `/order`, jak pokazałem na listingu 18.8.

Listing 18.8. Przejście pod adres URL zdefiniowane w pliku `productList.component.ts` w katalogu `src/app`

```
import { Component } from "@angular/core";
import { DataSource } from '../data/dataSource';
import { Product } from '../data/entities';
import { Router } from "@angular/router";

@Component({
  selector: "product-list",
  templateUrl: "../productList.component.html"
})
export class ProductList {
  selectedCategory = "Wszystkie";

  constructor(public dataSource: DataSource, private router: Router) {}

  get products(): Product[] {
    return this.dataSource.getProducts("id",
      this.selectedCategory === "Wszystkie" ? undefined : this.selectedCategory);
  }

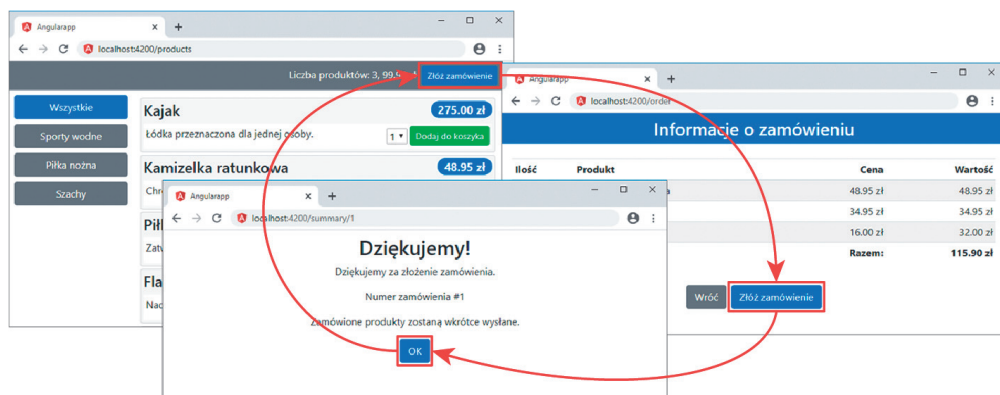
  get categories(): string[] {
    return ["Wszystkie", ...this.dataSource.getCategories()];
  }

  handleCategorySelect(category: string) {
    this.selectedCategory = category;
  }

  handleAdd(data: {product: Product, quantity: number}) {
    this.dataSource.order.addProduct(data.product, data.quantity);
  }

  handleSubmit() {
    this.router.navigateByUrl("/order");
  }
}
```

Zapisz wprowadzone zmiany i poczekaj chwilę, aż narzędzia programistyczne Angulara ponownie skompilują aplikację. W oknie przeglądarki WWW odśwież uruchomioną aplikację, która w tym momencie jest już ukończona. Masz możliwość wybierania produktów oraz wyświetlenia podsumowania zamówienia i przekazania go do serwera, jak pokazałem na rysunku 18.2.



Rysunek 18.2. Dodawanie komponentów do przykładowej aplikacji

- **Wskazówka** Jeżeli kliknięcie przycisku *Złóż zamówienie* spowoduje jedynie zmianę adresu URL, oznacza to, że prawdopodobnie nie zastąpiłeś zawartości pliku *app.component.html* zawartością przedstawioną na listingu 18.7.

Wdrażanie aplikacji

Narzędzia programistyczne Angulara opierają swoje działanie na serwerze Webpack Development Server, który nie jest odpowiedni do stosowania w środowisku produkcyjnym, ponieważ dodaje funkcje takie jak automatyczne odświeżanie strony po wygenerowaniu paczki JavaScriptu. W tym podrozdziale przedstawię proces przygotowania aplikacji Angulara do wdrożenia — jest on podobny do zastosowanego podczas wdrażania aplikacji internetowej utworzonej wyłącznie za pomocą TypeScriptu.

Dodawanie pakietu produkcyjnego serwera HTTP

W przypadku środowiska produkcyjnego wymagany jest zwykle serwer HTTP, który będzie dostarczał pliki HTML-a, CSS i JavaScriptu przeglądarce WWW. W omawianym przykładzie zdecydowałem się na użycie serwera Express — znajduje się on w tym samym pakiecie, z którego korzystam we wszystkich przykładach w tej części książki. Ten serwer jest dobrym wyborem dla wielu aplikacji internetowych. Zatrzymaj działanie narzędzi programistycznych Angulara przez naciśnięcie klawiszy *Ctrl+C*, a następnie z poziomu katalogu *angularapp* aplikacji w powłoce wydaj polecenie przedstawione na listingu 18.9, aby zainstalować niezbędny pakiet.

Listing 18.9. Dodawanie pakietów pozwalających na wdrożenie aplikacji Angulara

```
$ npm install --save-dev express@4.17.1
$ npm install --save-dev connect-history-api-fallback@1.6.0
```

Drugie polecenie powoduje zainstalowanie pakietu `connect-history-api-fallback`, który okazuje się użyteczny podczas wdrażania aplikacji wykorzystujących routing URL. Działanie pakietu polega na mapowaniu obsługiwanych przez aplikację żądań prowadzących do pliku `index.html` i zagwarantowaniu, że odświeżenie strony w przeglądarce WWW nie spowoduje wyświetlenia użytkownikowi błędu informującego o nieznalezieniu zasobu.

Tworzenie pliku dla trwałego magazynu danych

W celu utworzenia trwałego magazynu danych dla usługi sieciowej dodaj do katalogu `angularapp` plik o nazwie `data.json` zawierający kod przedstawiony na listingu 18.10.

Listing 18.10. Zawartość pliku `data.json` w katalogu `angularapp`

```
{
  "products": [
    { "id": 1, "name": "Kajak", "category": "Sporty wodne",
      "description": "Łódka przeznaczona dla jednej osoby.", "price": 275 },
    { "id": 2, "name": "Kamizelka ratunkowa", "category": "Sporty wodne",
      "description": "Chroni i dodaje uroku.", "price": 48.95 },
    { "id": 3, "name": "Piłka", "category": "Piłka nożna",
      "description": "Zatwierdzone przez FIFA rozmiar i waga.", "price": 19.50 },
    { "id": 4, "name": "Flagi naróżne", "category": "Piłka nożna",
      "description": "Nadadzą twojemu boisku profesjonalny wygląd.",
        "price": 34.95 },
    { "id": 5, "name": "Stadion", "category": "Piłka nożna",
      "description": "Składany stadion na 35 000 osób.", "price": 79500 },
    { "id": 6, "name": "Czapka", "category": "Szachy",
      "description": "Zwiększa efektywność mózgu o 75%.", "price": 16 },
    { "id": 7, "name": "Niestabilne krzesło", "category": "Szachy",
      "description": "Zmniejsza szanse przeciwnika.",
        "price": 29.95 },
    { "id": 8, "name": "Ludzka szachownica", "category": "Szachy",
      "description": "Przyjemna gra dla całej rodziny.", "price": 75 },
    { "id": 9, "name": "Błyszczący król", "category": "Szachy",
      "description": "Pokryty złotem i wysadzany diamentami król.", "price": 1200 }
  ],
  "orders": []
}
```

Tworzenie serwera

Aby utworzyć serwer dostarczający przeglądarce WWW aplikację i jej dane, dodaj do katalogu `angularapp` plik o nazwie `server.js` i umieść w nim kod przedstawiony na listingu 18.11.

Listing 18.11. Zawartość pliku *server.js* w katalogu *angularapp*

```
const express = require("express");
const jsonServer = require("json-server");
const history = require("connect-history-api-fallback");

const app = express();
app.use(history());
app.use("/", express.static("dist/angularapp"));

const router = jsonServer.router("data.json");
app.use(jsonServer.bodyParser)
app.use("/api", (req, resp, next) => router(req, resp, next));

const port = process.argv[3] || 4001;
app.listen(port, () => console.log(`Serwer nasłuchuje na porcie numer ${port}`));
```

Polecenia zdefiniowane w pliku *server.js* konfiguruja pakiety *express* i *json-server* w taki sposób, aby udostępniać zawartość katalogu *dist/angularapp*, w którym Angular umieszcza pliki JavaScriptu i HTML-a oraz nakazuje przeglądarce WWW ich wczytanie. Adresy URL poprzedzone prefiksem */api* będą obsługiwane przez usługę sieciową.

Używanie względnych adresów URL do obsługi żądań danych

Usługa sieciowa dostarczająca dane aplikacji będzie działała równolegle z serwerem frameworka Angular. Aby przygotować aplikację na wykonywanie żądań za pomocą pojedynczego portu, wymagana jest zmiana kodu klasy *RemoteDataSource*, jak pokazałem na listingu 18.12.

Listing 18.12. Używanie względnych adresów URL w kodzie pliku *remoteDataSource.ts* w katalogu *src/app/data*

```
import { Injectable } from "@angular/core";
import { HttpClient } from "@angular/common/http";
import { Observable } from "rxjs";
import { map } from "rxjs/operators";
import { DataSourceImpl } from "../dataSource";
import { Product, Order } from "../entities";

// const protocol = document.location.protocol;
// const hostname = document.location.hostname;
// const port = 4600;

const urls = {
  // products: `${protocol}/${hostname}:${port}/products`,
  // orders: `${protocol}/${hostname}:${port}/orders`
  products: "/api/products",
  orders: "/api/orders"
};

@Injectable()
export class RemoteDataSource extends DataSourceImpl {
  constructor(private http: HttpClient) {
    super();
  }
```

```

    }

    loadProducts(): Observable<Product[]> {
        return this.http.get<Product[]>(urls.products);
    }

    storeOrder(order: Order): Observable<number> {
        let orderData = {
            lines: [...order.orderLines.values()].map(ol => ({
                productId: ol.product.id,
                productName: ol.product.name,
                quantity: ol.quantity
            }))
        }
        return this.http.post<{ id: number}>(urls.orders, orderData)
            .pipe<number>(map(val => val.id));
    }
}

```

Podane adresy URL są względne dla używanych do obsługi żądań dokumentu HTML-a. Oznacza to stosowanie się do konwencji, zgodnie z którą żądania danych są poprzedzone prefiksem */api*.

Kompilowanie aplikacji

W celu skompilowania aplikacji należy z poziomu katalogu *angularapp* w powłoce wydać polecenie przedstawione na listingu 18.13, które spowoduje utworzenie produkcyjnej wersji aplikacji.

Listing 18.13. Tworzenie produkcyjnej paczki aplikacji

```
$ ng build --prod
```

W wyniku procesu kompilacji nastąpi utworzenie w katalogu *dist* zestawu zoptymalizowanych plików. Proces kompilacji może zająć dłuższą chwilę i spowoduje wygenerowanie następujących danych wyjściowych, które informują o utworzonych plikach:

```

√ Browser application bundle generation complete.
√ Copying assets complete.
√ Index html generation complete.
Initial Chunk Files | Names | Size
main.fae8db30eaa4f8e5a238.js | main | 255.33 kB
styles.a5f71e09a5471b3525f6.css | styles | 141.60 kB
polyfills.6abdde2583a2e01a2350.js | polyfills | 35.73 kB
runtime.7b63b9fd40098a2e8207.js | runtime | 1.45 kB
 | Initial Total | 434.11 kB
Build at: 09:44:57.144Z - Hash: 1ad09b3df3412b22d555 - Time: 24746ms

```

Testowanie gotowej aplikacji

Aby upewnić się o prawidłowym skompilowaniu aplikacji i zastosowaniu wszystkich wprowadzonych zmian konfiguracyjnych, z poziomu katalogu *angularapp* w powłoce wydaj polecenie przedstawione na listingu 18.14.

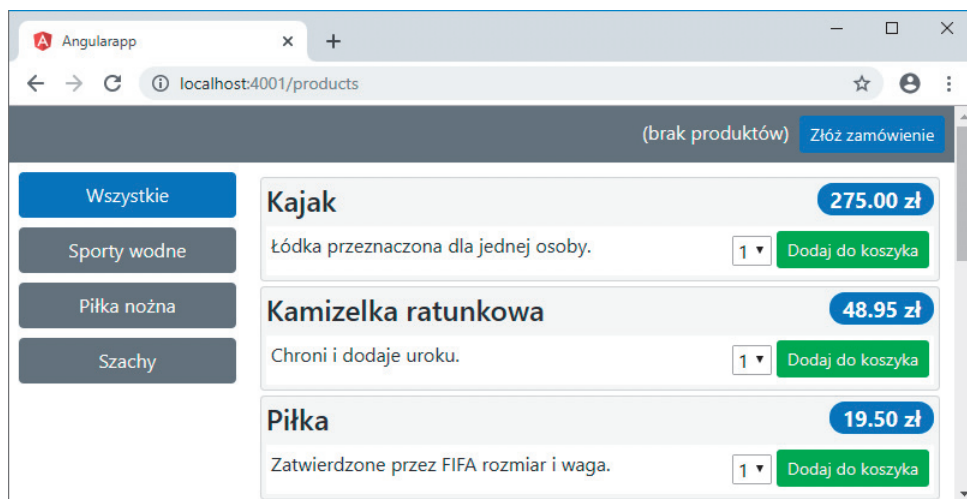
Listing 18.14. Uruchamianie serwera produkcyjnego

```
$ node server.js
```

Kod przedstawiony na listingu 18.14 zostanie wykonany i spowoduje wygenerowanie następujących danych wyjściowych:

Serwer nasłuchuje na porcie numer 4001

Otwórz okno przeglądarki WWW i przejdź pod adres *http://localhost:4001*, a zobaczysz działającą aplikację, jak pokazałem na rysunku 18.3.



Rysunek 18.3. Uruchomiona aplikacja dla środowiska produkcyjnego

Umieszczanie aplikacji w kontenerze

Ten rozdział zamierzam zakończyć utworzeniem dla przykładowej aplikacji kontenera, który następnie będzie można wdrożyć w środowisku produkcyjnym. Jeżeli w rozdziale 15. nie zainstalowałeś Dockera, będziesz to musiał zrobić teraz, jeśli chcesz wykonywać przykłady przedstawione w pozostałej części rozdziału.

Przygotowanie aplikacji

Pracę należy zacząć od utworzenia pliku konfiguracyjnego dla menedżera pakietów Node.js, który będzie użyty do pobrania pakietów dodatkowych wymaganych przez aplikację uruchamianą w kontenerze. W katalogu *angularapp* utwórz więc plik o nazwie *deploy-package.json* i umieść w nim kod przedstawiony na listingu 18.15.

Listing 18.15. Zawartość pliku *deploy-package.json* w katalogu *angularapp*

```
{
  "name": "angularapp",
  "description": "Aplikacja internetowa Angulara",
  "repository": "https://github.com/Apress/essential-typescript",
  "license": "0BSD",
  "devDependencies": {
    "express": "4.17.1",
    "json-server": "0.16.3",
    "connect-history-api-fallback": "1.6.0"
  }
}
```

Sekcja *devDependencies* wymienia pakiety niezbędne do uruchomienia aplikacji w kontenerze. Wszystkie pakiety, dla których istnieją polecenia *import* w plikach kodu źródłowego aplikacji, zostały umieszczone w pliku paczki wygenerowanym przez pakiet *webpack*. Pozostałe opcje opisują aplikację i zostały umieszczone w kodzie w celu uniknięcia komunikatów ostrzeżeń generowanych podczas tworzenia kontenera.

Tworzenie kontenera Dockera

Aby zdefiniować kontener, należy dodać do katalogu *angularapp* plik o nazwie *Dockerfile* (bez żadnego rozszerzenia) i umieścić w nim kod przedstawiony na listingu 18.16.

Listing 18.16. Zawartość pliku *Dockerfile* w katalogu *angularapp*

```
FROM node:14.15.4

RUN mkdir -p /usr/src/angularapp

COPY dist /usr/src/angularapp/dist/
COPY data.json /usr/src/angularapp/
COPY server.js /usr/src/angularapp/
COPY deploy-package.json /usr/src/angularapp/package.json

WORKDIR /usr/src/angularapp

RUN echo 'package-lock=false' >> .npmrc
RUN npm install

EXPOSE 4001

CMD ["node", "server.js"]
```

Zawartość pliku *Dockerfile* wykorzystuje obraz bazowy skonfigurowany z Node.js i zawierający skopiowane pliki niezbędne do uruchomienia aplikacji, m.in. plik paczki z aplikacją oraz plik używany do zainstalowania pakietów Node.js wymaganych do uruchomienia aplikacji w środowisku produkcyjnym.

Aby przyspieszyć proces tworzenia kontenera, do katalogu *angularapp* warto również dodać plik *.dockerignore* o zawartości przedstawionej na listingu 18.17. W omawianym przykładzie ten plik nakazuje Dockerowi zignorowanie katalogu *node_modules*, który nie jest wymagany w kontenerze, a jego przetworzenie będzie wymagało ogromnej ilości czasu.

Listing 18.17. Zawartość pliku *.dockerignore* w katalogu *angularapp*

```
node_modules
```

Z poziomu katalogu *angularapp* w powłoce wydaj polecenie przedstawione na listingu 18.18, które rozpocznie proces tworzenia kontenera zawierającego przykładową aplikację i wszystkie wymagane pakiety.

Listing 18.18. Tworzenie obrazu Dockera

```
$ docker build . -t angularapp -f Dockerfile
```

Obraz jest szablonem dla kontenera. Docker przetworzy polecenia zdefiniowane w pliku *Dockerfile*, wskazane pakiety Node.js zostaną pobrane i zainstalowane, a pliki konfiguracyjne i kodu źródłowego będą skopiowane do obrazu.

Uruchamianie aplikacji

Gdy obraz Dockera jest gotowy, można uruchomić nowy kontener za pomocą polecenia wymienionego na listingu 18.19.

Listing 18.19. Uruchamianie kontenera Dockera

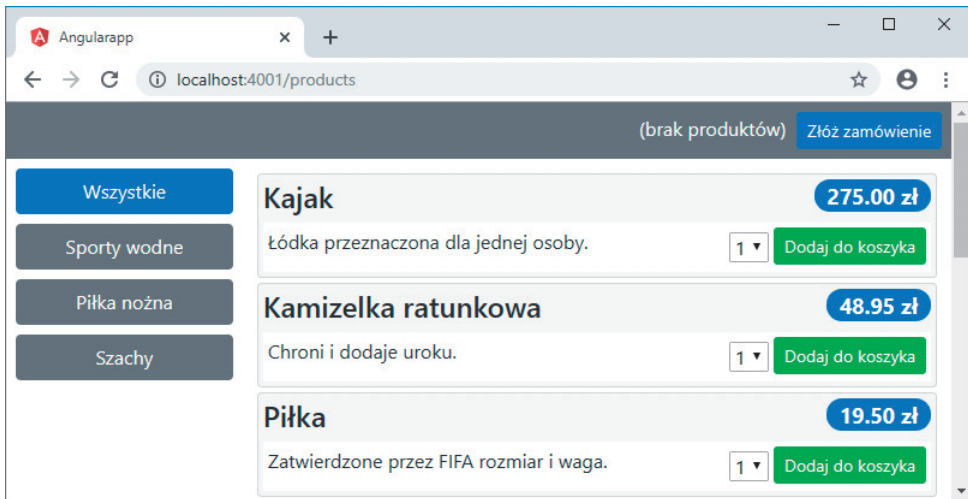
```
$ docker run -p 4001:4001 angularapp
```

Aby przetestować aplikację, należy przejść w przeglądarce WWW pod adres *http://localhost:4001*, co spowoduje wyświetlenie odpowiedzi udzielonej przez serwer WWW uruchomiony w kontenerze, jak pokazałem na rysunku 18.4.

Jeżeli chcesz zatrzymać działanie kontenera, musisz zacząć od wydania polecenia przedstawionego na listingu 18.20.

Listing 18.20. Wyświetlenie listy kontenerów Dockera

```
$ docker ps
```



Rysunek 18.4. Przykładowa aplikacja uruchomiona w kontenerze

Dane wyjściowe tego polecenia to lista uruchomionych kontenerów (w celu zachowania zwięzłości i czytelności pominąłem część danych wyjściowych):

CONTAINER ID	IMAGE	COMMAND	CREATED
48dbd2431700	angularapp	"docker-entrypoint.s..."	41 seconds ago

Wykorzystując wartość wyświetloną w kolumnie *CONTAINER ID*, wydaj polecenie przedstawione na listingu 18.21.

Listing 18.21. Zatrzymanie kontenera Dockera

```
$ docker stop 48dbd2431700
```

W tym momencie aplikacja jest gotowa do wdrożenia na dowolnej platformie obsługującej kontenery Dockera.

Podsumowanie

W tym rozdziale dokończyłem pracę nad przykładową aplikacją Angulara przez dodanie komponentów i wykorzystanie funkcjonalności routingu URL do określenia treści wyświetlanej użytkownikowi. Zobaczyłeś, jak wygląda proces kompilowania aplikacji dla środowiska produkcyjnego i jej umieszczenia w kontenerze, co pozwala na łatwe wdrożenie. W następnym rozdziale zajmę się tworzeniem aplikacji internetowej za pomocą frameworka React.

ROZDZIAŁ 19.



Tworzenie aplikacji internetowej React — część I

W tym rozdziale rozpocznę proces tworzenia aplikacji React, która będzie miała taką samą funkcjonalność jak dwie utworzone wcześniej. Wprawdzie język TypeScript jest opcjonalny w trakcie programowania React, ale ma zapewnioną dobrą obsługę, a połączenie obu wymienionych technologii podczas tworzenia aplikacji sprawdza się doskonale. W tabeli 19.1 wymienię opcje kompilatora TypeScriptu użyte w rozdziale.

Tabela 19.1. Opcje kompilatora TypeScriptu użyte w rozdziale

Opcja	Opis
<code>allowJs</code>	Ta opcja powoduje dodanie wszystkich plików JavaScriptu w procesie kompilacji
<code>allowSyntheticDefaultImports</code>	Ta opcja powoduje importowanie modułów, które nie deklarują wyraźnie eksportowanych funkcji. Jest używana w celu zwiększenia zgodności kodu źródłowego
<code>esModuleInterop</code>	Ta opcja dodaje kod pomocniczy pozwalający na importowanie funkcjonalności z modułów niedefiniujących domyślnie eksportowanych funkcji i jest używana z opcją <code>allowSyntheticDefaultImports</code>
<code>forceConsistentCasingInFileNames</code>	Ta opcja gwarantuje, że nazwy w poleceniach <code>import</code> będą dopasowane do wielkości znaków użytej w zaimportowanym pliku
<code>isolatedModules</code>	Ta opcja powoduje traktowanie każdego pliku jako oddzielnego modułu, co z kolei zwiększa zgodność z narzędziem Babel
<code>lib</code>	Ta opcja pozwala na wybór używanych przez kompilator plików deklaracji typu
<code>module</code>	Ta opcja określa format używany dla modułów

Tabela 19.1. Opcje kompilatora TypeScriptu użyte w rozdziale (ciąg dalszy)

Opcja	Opis
<code>moduleResolution</code>	Ta opcja określa styl używany podczas rozwiązywania zależności modułów
<code>noEmit</code>	Ta opcja uniemożliwia kompilatorowi emisję kodu JavaScriptu, a wynik jest jedynie sprawdzany pod kątem błędów
<code>resolveJsonModule</code>	Ta opcja pozwala na importowanie plików JSON, jakby były modułami
<code>skipLibCheck</code>	Ta opcja przyspiesza kompilację przez pominięcie standardowo stosowanej operacji sprawdzania plików deklaracji
<code>strict</code>	Ta opcja włącza ściślejsze sprawdzanie kodu TypeScriptu
<code>target</code>	Ta opcja określa docelową wersję języka JavaScript, dla której kompilator wygeneruje kod

Przygotowanie projektu

Projekt React najłatwiej jest utworzyć za pomocą pakietu `create-react-app`. W powłoce przejdź do katalogu, w którym chcesz utworzyć aplikację React, a następnie wydaj polecenie przedstawione na listingu 19.1, aby w ten sposób zainstalować niezbędny pakiet.

Listing 19.1. Instalowanie pakietu przeznaczonego do tworzenia projektów React

```
$ npm install --global create-react-app@4.0.1
```

- **Wskazówka** Przykładowy projekt dla tego rozdziału, a także projekty dla pozostałych rozdziałów książki zostały umieszczone w archiwum dostępnym pod adresem <https://ftp.helion.pl/przyklady/tys4o2.zip>.

Po zainstalowaniu pakietu wydaj polecenie przedstawione na listingu 19.2 i tym samym utwórz nowy projekt o nazwie `reactapp`.

Listing 19.2. Tworzenie projektu React

```
$ npx create-react-app reactapp --template typescript --use-npm
```

Argument `--template typescript` nakazuje pakietowi `create-react-app` utworzenie skonfigurowanego do użycia z TypeScriptem projektu React, który będzie zawierał zainstalowany i skonfigurowany kompilator TypeScriptu, a także pliki deklaracji opisujące API React i powiązane z nim narzędzia. Z kolei argument `--use-npm` nakazuje zainstalowanie pakietów za pomocą menedżera pakietów npm, z którego korzystam w książce.

■ **Wskazówka** Na stronie <https://create-react-app.dev/docs/adding-typescript/> znajdziesz więcej informacji na temat dodawania obsługi TypeScriptu do istniejącego projektu React.

Konfigurowanie usługi sieciowej

Po utworzeniu projektu wydaj polecenia przedstawione na listingu 19.3, aby przejść do katalogu projektu i dodać pakiety zapewniające obsługę usługi sieciowej oraz pozwalające na uruchamianie wielu pakietów za pomocą jednego polecenia.

Listing 19.3. Dodawanie niezbędnych pakietów do projektu

```
$ cd reactapp
$ npm install --save-dev json-server@0.16.3
$ npm install --save-dev npm-run-all@4.1.5
```

W celu przygotowania danych dla usługi sieciowej dodaj do katalogu *reactapp* plik o nazwie *data.js* zawierający kod przedstawiony na listingu 19.4.

Listing 19.4. Zawartość pliku data.js File w katalogu reactapp

```
module.exports = function () {
  return {
    products: [
      { id: 1, name: "Kajak", category: "Sporty wodne",
        description: "Łódka przeznaczona dla jednej osoby.", price: 275 },
      { id: 2, name: "Kamizelka ratunkowa", category: "Sporty wodne",
        description: "Chroni i dodaje uroku.", price: 48.95 },
      { id: 3, name: "Piłka", category: "Piłka nożna",
        description: "Zatwierdzone przez FIFA rozmiar i waga.", price: 19.50 },
      { id: 4, name: "Flagi narożne", category: "Piłka nożna",
        description: "Nadadzą twojemu boisku profesjonalny wygląd.",
          price: 34.95 },
      { id: 5, name: "Stadion", category: "Piłka nożna",
        description: "Składany stadion na 35 000 osób.", price: 79500 },
      { id: 6, name: "Czapka", category: "Szachy",
        description: "Zwiększa efektywność mózgu o 75%.", price: 16 },
      { id: 7, name: "Niestabilne krzesło", category: "Szachy",
        description: "Zmniejsza szanse przeciwnika.",
          price: 29.95 },
      { id: 8, name: "Ludzka szachownica", category: "Szachy",
        description: "Przyjemna gra dla całej rodziny.", price: 75 },
      { id: 9, name: "Błyszczący król", category: "Szachy",
        description: "Pokryty złotem i wysadzany diamentami król.", price: 1200 }
    ],
    orders: []
  }
}
```

Uaktualnij sekcję *scripts* pliku *package.json* i tym samym skonfiguruj narzędzia programistyczne w taki sposób, aby zbiór narzędzi React i usługa sieciowa były uruchamiane jednocześnie, jak pokazałem na listingu 19.5.

Listing 19.5. Konfigurowanie narzędzi w pliku *package.json* w katalogu *reactapp*

```
...
"scripts": {
  "json": "json-server data.js -p 4600",
  "serve": "react-scripts start",
  "start": "npm-run-all -p serve json",
  "build": "react-scripts build",
  "test": "react-scripts test",
  "eject": "react-scripts eject"
},
...
```

Instalowanie pakietu Bootstrap CSS

Polecenie przedstawione na listingu 19.6 wydane z poziomu katalogu *reactapp* w powłoce powoduje dodanie do projektu obsługi frameworka Bootstrap CSS.

Listing 19.6. Dodawanie pakietu Bootstrap CSS

```
$ npm install bootstrap@4.6.1
```

Aby mieć pewność o dołączeniu arkusza stylów Bootstrap CSS w aplikacji, dodaj do pliku *index.tsx* w katalogu *src* polecenie `import` przedstawione na listingu 19.7.

Listing 19.7. Deklarowanie zależności w pliku *index.tsx* w katalogu *src*

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import reportWebVitals from './serviceWorker';
import 'bootstrap/dist/css/bootstrap.css';

ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
);

reportWebVitals();
```

Uruchamianie przykładowej aplikacji

Polecenie wymienione na listingu 19.8 wydane z poziomu katalogu *reactapp* w powłoce spowoduje uruchomienie przykładowej aplikacji.

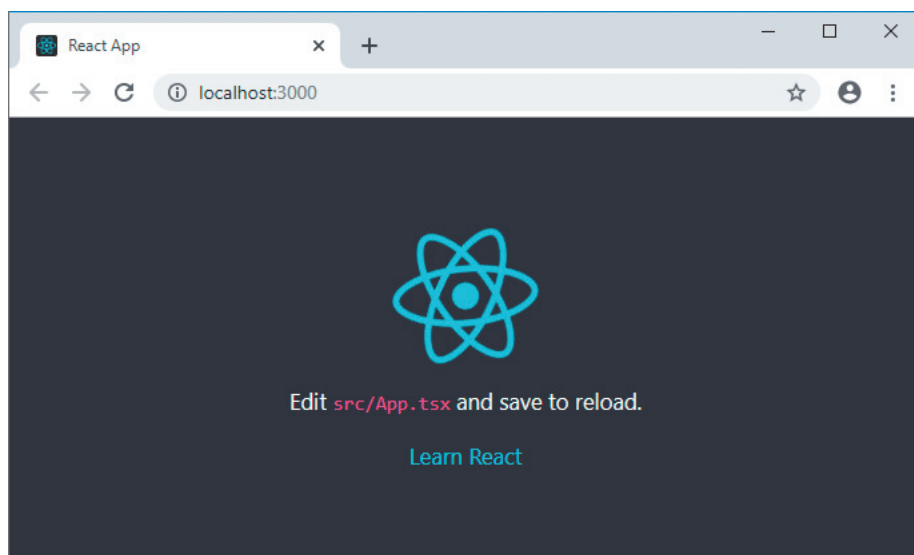
Listing 19.8. Uruchamianie narzędzi programistycznych

```
$ npm start
```

Nastąpi uruchomienie usługi sieciowej i narzędzi kompilacji React, co spowoduje wygenerowanie następujących danych wyjściowych:

```
Compiled successfully!
You can now view reactapp in the browser.
  Local:            http://localhost:3000/
  On Your Network:  http://172.22.208.1:3000/
Note that the development build is not optimized.
To create a production build, use npm run build.
```

Po przejściu w przeglądarce WWW pod adres *http://localhost:3000* zobaczysz miejsce zarezerwowane dla aplikacji, które zostało zdefiniowane podczas procesu tworzenia projektu, jak pokazałem na rysunku 19.1.



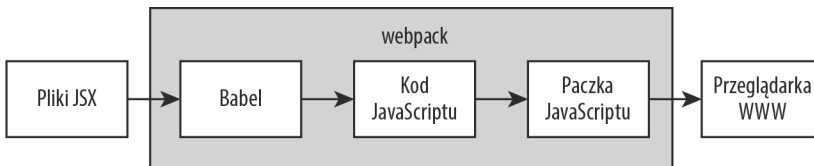
Rysunek 19.1. Przykładowa aplikacja uruchomiona w przeglądarce WWW

TypeScript i programowanie z użyciem frameworka React

Język TypeScript jest opcjonalny podczas programowania z użyciem Reacta, co znajduje odzwierciedlenie w sposobie konfiguracji narzędzi programistycznych i kompilatora TypeScriptu. W tle pakiet webpack i serwer Webpack Development Server są używane do tworzenia pliku paczki JavaScriptu i dostarczenia go przeglądarce WWW.

Podczas programowania z użyciem frameworka React wykorzystywany jest format JSX, który poznałeś w rozdziale 15., pozwalający na połączenie w jednym pliku kodu JavaScriptu i HTML-a. Narzędzia programistyczne React mają możliwość konwersji plików JSX na postać czystego kodu JavaScriptu, co odbywa się za pomocą pakietu Babel. Babel to po prostu kompilator

JavaScriptu pozwalający na przeprowadzanie konwersji kodu utworzonego w najnowszych wydaniach JavaScriptu na postać kodu działającego w starszych przeglądarkach WWW, czyli podobnie jak w przypadku funkcjonalności wyboru wersji JavaScriptu w kompilatorze TypeScript. Możliwości kompilatora Babel mogą być rozszerzane za pomocą wtyczek — obecnie pozwala on na konwersję wielu formatów, w tym także JSX, na postać kodu JavaScriptu. Na rysunku 19.2 pokazałem podstawowy zestaw narzędzi programistycznych React stosowany podczas pracy nad zwykłymi projektami JavaScriptu.



Rysunek 19.2. Zestaw narzędzi programistycznych React podczas pracy z projektami JavaScriptu

Wtyczka dla kompilatora Babel odpowiedzialna za obsługę JSX pełni taką samą funkcję jak klasa fabryki JSX utworzona w rozdziale 15. — zastępuje fragmenty kodu HTML-a poleceniami JavaScriptu, choć wykorzystuje przy tym znacznie bardziej zaawansowany i efektywny API React. Wynikiem konwersji jest czysty kod JavaScriptu umieszczany w pojedynczym pliku, który następnie jest przekazywany przeglądarce WWW w celu jego wykonania. Plik paczki zawiera również kod JavaScriptu przeznaczony do rozpakowania dowolnego zasobu CSS lub obrazka wymaganej przez aplikację.

Sposób, w jaki zestaw narzędzi React współdziała z językiem TypeScript, jest nietypowy. To, co się tak naprawdę dzieje, można sprawdzić poprzez analizę pliku konfiguracyjnego kompilatora TypeScript, który został dodany do projektu:

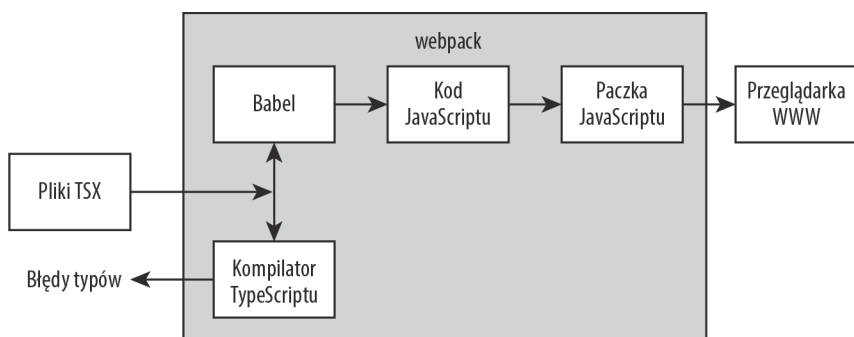
```

{
  "compilerOptions": {
    "target": "es5",
    "lib": ["dom", "dom.iterable", "esnext"],
    "allowJs": true,
    "skipLibCheck": true,
    "esModuleInterop": true,
    "allowSyntheticDefaultImports": true,
    "strict": true,
    "forceConsistentCasingInFileNames": true,
    "noFallthroughCasesInSwitch": true,
    "module": "esnext",
    "moduleResolution": "node",
    "resolveJsonModule": true,
    "isolatedModules": true,
    "noEmit": true,
    "jsx": "react-jsx"
  },
  "include": ["src"]
}
  
```

Ustawienie, na które warto zwrócić uwagę, to `noEmit`. Gdy opcja `noEmit` ma przypisaną wartość `true`, kompilator TypeScriptu nie będzie generował plików JavaScriptu. Powodem

zastosowania tak nietypowego ustawienia kompilatora TypeScriptu jest to, że za konwersję kodu TypeScriptu na język JavaScript odpowiada pakiet Babel, a nie kompilator TypeScriptu. Zestaw narzędzi programistycznych React zawiera wtyczkę Babel pozwalającą skonwertować kod TypeScriptu na czysty kod JavaScriptu.

Wprawdzie Babel ma możliwość konwersji kodu TypeScriptu na język JavaScript, ale nie potrafi obsługiwać funkcji TypeScriptu i nie wie, w jaki sposób przeprowadzać sprawdzanie typu. To zadanie pozostawiono kompilatorowi TypeScriptu, więc odpowiedzialność za obsługę kodu TypeScriptu jest podzielona — kompilator TypeScriptu odpowiada za wyszukiwanie błędów, a Babel za utworzenie kodu JavaScriptu, który później będzie wykonywany przez przeglądarkę WWW, jak pokazałem na rysunku 19.3.



Rysunek 19.3. Zestaw narzędzi programistycznych React i obsługa kodu TypeScriptu

W tym kontekście ustawienie `noEmit` ma sens, ponieważ w celu sprawdzenia poprawności typów kompilator TypeScriptu nie musi się zajmować obsługą treści HTML-a lub tworzeniem plików JavaScriptu.

Ograniczeniem w przedstawionym rozwiązaniu jest to, że Babel nie potrafi poradzić sobie z każdą funkcją TypeScriptu, choć tak naprawdę ograniczeń jest zaskakująco niewiele. W chwili powstawania książki wyliczenia nie były w pełni obsługiwane i nie można było stosować przestrzeni nazw (po wprowadzeniu modułów JavaScriptu przestrzenie nazw zostały uznane za przestarzałe i nie są omawiane w książce).

■ **Uwaga** Podczas uruchamiania narzędzi programistycznych możesz otrzymać ostrzeżenie informujące o niedopasowaniu między wersjami TypeScriptu. To ostrzeżenie odzwierciedla możliwe różnice między funkcjami sprawdzania typu zaimplementowanymi w najnowszych wersjach kompilatora TypeScriptu i sposobem, w jaki kod TypeScriptu jest przez pakiet Babel konwertowany na kod JavaScriptu. W przypadku prostych projektów, takich jak omawiany w rozdziale, istnieje niewielkie niebezpieczeństwo, że to niedopasowanie doprowadzi do poważnych problemów. Mimo wszystko powinieneś rozważyć używanie tylko tych wersji TypeScriptu, które są wyraźnie obsługiwane przez pakiet `create-react-app`.

Podobnie jak w innych rozdziałach tej części książki, zamierzam skorzystać z operatora rozwinięcia, który wymaga wprowadzenia zmiany w konfiguracji kompilatora TypeScriptu, jak pokazałem na listingu 19.9.

Listing 19.9. Zmiana konfiguracji kompilatora w pliku *tsconfig.json* w katalogu *reactapp*

```

{
  "compilerOptions": {
    "target": "es6",
    "lib": ["dom", "dom.iterable", "esnext"],
    "allowJs": true,
    "skipLibCheck": true,
    "esModuleInterop": true,
    "allowSyntheticDefaultImports": true,
    "strict": true,
    "forceConsistentCasingInFileNames": true,
    "noFallthroughCasesInSwitch": true,
    "module": "esnext",
    "moduleResolution": "node",
    "resolveJsonModule": true,
    "isolatedModules": true,
    "noEmit": true,
    "jsx": "react-jsx"
  },
  "include": ["src"]
}

```

Konwersja przeprowadzana przez Babel potrafi poradzić sobie z operatorem rozwinięcia bez konieczności wprowadzania zmian w konfiguracji, a ustawienie `target` przedstawione na listingu 19.9 jedynie uniemożliwia kompilatorowi TypeScriptu wygenerowanie błędów.

Definiowanie typów encji

Framework React koncentruje się na przedstawianiu treści użytkownikowi, a pozostałe zadania, takie jak zarządzanie danymi aplikacji i wykonywanie żądań HTTP, są pozostawione innym pakietom. Niezbędne pakiety wymagane do implementacji funkcjonalności aplikacji zostaną dodane do projektu na dalszym etapie prac. Teraz powinniśmy skupić się na funkcjach dostarczanych przez React, natomiast tymi brakującymi zajmiesz się później. Na początek należy zdefiniować encje używane przez aplikację. Utwórz katalog *src/data* i dodaj do niego plik o nazwie *entities.ts* zawierający kod przedstawiony na listingu 19.10.

Listing 19.10. Zawartość pliku *entities.ts* w katalogu *src/data*

```

export type Product = {
  id: number,
  name: string,
  description: string,
  category: string,
  price: number
};

export class OrderLine {
  constructor(public product: Product, public quantity: number) {
    // Polecenia nie są wymagane.
  }
}

```

```

    get total(): number {
        return this.product.price * this.quantity;
    }
}

export class Order {
    private lines = new Map<number, OrderLine>();

    constructor(initialLines?: OrderLine[]) {
        if (initialLines) {
            initialLines.forEach(ol => this.lines.set(ol.product.id, ol));
        }
    }

    public addProduct(prod: Product, quantity: number) {
        if (this.lines.has(prod.id)) {
            if (quantity === 0) {
                this.removeProduct(prod.id);
            } else {
                this.lines.get(prod.id)!.quantity += quantity;
            }
        } else {
            this.lines.set(prod.id, new OrderLine(prod, quantity));
        }
    }

    public removeProduct(id: number) {
        this.lines.delete(id);
    }

    get orderLines(): OrderLine[] {
        return [...this.lines.values()];
    }

    get productCount(): number {
        return [...this.lines.values()]
            .reduce((total, ol) => total += ol.quantity, 0);
    }

    get total(): number {
        return [...this.lines.values()].reduce((total, ol) => total += ol.total, 0);
    }
}

```

To dokładnie ten sam zbiór typów danych, które są używane w innych aplikacjach internetowych utworzonych w tej części książki. Niezależnie od wybranego frameworka ten sam zestaw funkcji może być wykorzystany do opisanego typów danych.

Wyświetlanie filtrowanej listy produktów

Framework React używa formatu JSX, aby umożliwić zdefiniowanie elementów HTML-a z kodem JavaScriptu. To podejście jest podobne do zastosowanego podczas tworzenia aplikacji internetowej niewykorzystującej żadnego frameworka. W trakcie kompilacji elementy HTML-a

zostają skonwertowane na polecenia JavaScriptu używające API React do efektywnego wyświetlania treści użytkownikowi. Takie rozwiązanie okazuje się znacznie bardziej eleganckie niż zastosowane w rozdziale 15.

Elementem konstrukcyjnym o znaczeniu kluczowym w aplikacji React jest komponent, który staje się odpowiedzialny za generowanie treści HTML-a. Komponenty są konfigurowane za pomocą właściwości, mogą reagować na działania użytkownika przez obsługę zdarzeń generowanych w elementach HTML-a wyświetlanych przez te komponenty oraz definiować lokalne dane stanu.

W celu wyświetlenia informacji o pojedynczym produkcie dodaj do katalogu *src* plik o nazwie *productItem.tsx* z kodem przedstawionym na listingu 19.11 tworzącym prosty komponent React.

Listing 19.11. Zawartość pliku *productItem.tsx* w katalogu *src*

```
import React, { Component, ChangeEvent } from "react";
import { Product } from "../data/entities";

interface Props {
  product: Product,
  callback: (product: Product, quantity: number) => void
}

interface State {
  quantity: number
}

export class ProductItem extends Component<Props, State> {
  constructor(props: Props) {
    super(props);
    this.state = {
      quantity: 1
    }
  }

  render() {
    return <div className="card m-1 p-1 bg-light">
      <h4>
        { this.props.product.name }
        <span className="badge badge-pill badge-primary float-right">
          { this.props.product.price.toFixed(2) } zł
        </span>
      </h4>
      <div className="card-text bg-white p-1">
        { this.props.product.description }
        <button className="btn btn-success btn-sm float-right"
          onClick={ this.handleAddToCart } >
          Dodaj do koszyka
        </button>
        <select className="form-control-inline float-right m-1"
          onChange={ this.handleQuantityChange }>
          <option>1</option>
          <option>2</option>
```

```

        <option>3</option>
      </select>
    </div>
  </div>
}

handleQuantityChange = (ev: ChangeEvent<HTMLSelectElement>): void =>
  this.setState({ quantity: Number(ev.target.value) });

handleAddToCart = (): void =>
  this.props.callback(this.props.product, this.state.quantity);
}

```

Używanie języka TypeScript wymaga pewnych zmian w sposobie definiowania komponentów frameworka React, aby typy danych stosowane do opisywania właściwości i danych stanu definiowały i wykorzystywały argumenty typów generycznych klasy `Component`. Komponent `ProductItem` otrzymuje właściwości dostarczające obiekt `Product` i funkcję wywołania zwrótnego wykonywaną po kliknięciu przycisku *Dodaj do koszyka*. Komponent `ProductItem` ma jedną właściwość danych stanu, `quantity`, używaną do reakcji na wybór przez użytkownika wartości w elemencie `<select>`. Właściwości i dane stanu są opisywane za pomocą interfejsów `Props` i `State`, które są wykorzystywane jako parametry typów generycznych do konfiguracji klasy bazowej dla komponentów, np. jak pokazałem w kolejnym fragmencie kodu:

```

...
export class ProductItem extends Component<Props, State> {
...

```

Argumenty typów generycznych pozwalają kompilatorowi TypeScriptu na sprawdzanie komponentu podczas jego stosowania, aby mogły być używane jedynie właściwości zdefiniowane przez interfejs `Props` w celu zagwarantowania, że uaktualnienia są stosowane tylko poprzez właściwości zdefiniowane przez interfejs `State`.

Pliki deklaracji dla komponentu frameworka React obejmują typy dla zdarzeń generowanych przez elementy HTML-a z wykorzystaniem metody `render()`. W przypadku zdarzenia `change()` wywoływanego przez element `<select>` funkcja obsługi zdarzeń będzie otrzymywała obiekt `ChangeEvent<HTMLSelectElement>`. Zmiana właściwości komponentu musi się odbywać poprzez metodę `setState()`, dzięki której framework React jest informowany o wprowadzeniu zmiany.

```

...
handleQuantityChange = (ev: ChangeEvent<HTMLSelectElement>): void =>
  this.setState({ quantity: Number(ev.target.value) });
...

```

Kompilator TypeScriptu gwarantuje obsługę odpowiedniego typu zdarzenia, a także to, że uaktualnienia za pomocą metody `setState()` mają odpowiedni typ i dotyczą jedynie właściwości zdefiniowanych przez typ `State`.

Używanie zaczepów i komponentów funkcyjnych

Komponent przedstawiony na listingu 19.11 został zdefiniowany za pomocą klasy, choć framework React obsługuje również komponenty definiowane za pomocą funkcji. Gdy korzystasz z TypeScriptu, komponenty funkcyjne otrzymują adnotację `FunctionComponent<t>`, w której typ generyczny `T` opisuje właściwości otrzymywanego komponentu. Na listingu 19.12 znajduje się zmodyfikowana wersja komponentu `ProductItem` wyrażona w postaci funkcji, a nie klasy.

Listing 19.12. Definiowanie komponentu funkcyjnego w kodzie pliku `productItem.tsx` w katalogu `src`

```
import React, { FunctionComponent, useState } from "react";
import { Product } from "../data/entities";

interface Props {
  product: Product,
  callback: (product: Product, quantity: number) => void
}

// interface State {
//   quantity: number
// }

export const ProductItem: FunctionComponent<Props> = (props) => {

  const [quantity, setQuantity] = useState<number>(1);

  return <div className="card m-1 p-1 bg-light">
    <h4>
      { props.product.name }
      <span className="badge badge-pill badge-primary float-right">
        { props.product.price.toFixed(2) } zł
      </span>
    </h4>
    <div className="card-text bg-white p-1">
      { props.product.description }
      <button className="btn btn-success btn-sm float-right"
        onClick={ () => props.callback(props.product, quantity) }>
        Dodaj do koszyka
      </button>
      <select className="form-control-inline float-right m-1"
        onChange={ (ev) => setQuantity(Number(ev.target.value)) }>
        <option>1</option>
        <option>2</option>
        <option>3</option>
      </select>
    </div>
  </div>
}
```

Wynikiem działania komponentu funkcyjnego jest kod HTML-a przeznaczony do wyświetlenia użytkownikowi i zdefiniowany za pomocą tego samego zestawu elementów i wyrażeń, który komponenty oparte na klasie generują za pomocą metody `render()`.

Komponent oparty na klasie wykorzystuje właściwości i metody (dostępne za pomocą słowa kluczowego `this`) do implementacji danych stanu i obsługi cyklu życiowego, który framework React zapewnia aplikacji. Z kolei komponenty funkcyjne do otrzymania tego samego efektu stosują tzw. *zaczepy*:

```
...
const [quantity, setQuantity] = useState<number>(1);
...
```

Jest to przykład zaczepu stanu dostarczającego komponentowi funkcyjnemu właściwość danych stanu, której modyfikacja spowoduje wywołanie operacji uaktualnienia treści. Funkcja `useState()` otrzymuje argument typu generycznego i wartość początkową, a zwraca właściwość, która może być odczytana w celu pobrania wartości bieżącej, oraz funkcję pozwalającą na zmianę wartości. W omawianym przykładzie właściwość ma nazwę `quantity`, a funkcja zmieniająca jej wartość to `setQuantity()`, co oznacza zastosowanie się do powszechnie przyjętej konwencji nazw. W efekcie właściwość `quantity` może być używana w wyrażeniach do pobrania wartości danych stanu.

```
...
onClick={ () => props.callback(props.product, quantity) }>
...
```

Właściwość `quantity` jest stałą, co oznacza brak możliwości jej modyfikacji. Zamiast tego zmiana musi się odbywać za pomocą funkcji `setQuantity()`:

```
...
<select className="form-control-inline float-right m-1"
  onChange={ (ev) => setQuantity(Number(ev.target.value)) }>
...
```

Wykorzystanie oddzielnych właściwości i funkcji zapewnia, że każda zmiana danych stanu spowoduje wywołanie procesu uaktualnienia frameworka React, a kompilator TypeScriptu sprawdzi wartości przekazywane funkcji, aby mieć pewność o ich dopasowaniu do argumentu typu generycznego dostarczanego funkcji `useState()`.

■ **Wskazówka** Wybór między komponentami opartymi na funkcji lub klasie to kwestia osobistych preferencji, a oba podejścia są w pełni obsługiwane przez React. Ja stosuję klasy, ponieważ ten model programowania bardziej mi pasuje. Warto w tym miejscu dodać, że oba rozwiązania mają swoje wady i zalety i mogą być dowolnie łączone w projekcie.

Wyświetlanie listy kategorii i nagłówka

W celu zdefiniowania komponentu wyświetlającego listę kategorii dodaj do katalogu `src` plik o nazwie `categoryList.tsx` z kodem przedstawionym na listingu 19.13.

Listing 19.13. Zawartość pliku `categoryList.tsx` w katalogu `src`

```
import React, { Component } from "react";

interface Props {
```



```

    selected: string,
    categories: string[],
    selectCategory: (category: string) => void;
  }

export class CategoryList extends Component<Props> {

  render() {
    return <div>
      { ["Wszystkie", ...this.props.categories].map(c => {
        let btnClass = this.props.selected === c
          ? "btn-primary": "btn-secondary";
        return <button key={ c }
          className={ `btn btn-block ${btnClass}` }
          onClick={ () => this.props.selectCategory(c) }>
          { c }
        </button>
      }) }
    </div>
  }
}

```

Komponent `CategoryList` nie definiuje żadnych danych stanu, a jego klasa bazowa została określona za pomocą tylko jednego argumentu typu. W celu utworzenia komponentu wyświetlającego nagłówek dodaj do katalogu `src` plik o nazwie `header.tsx` z kodem przedstawionym na listingu 19.14.

Listing 19.14. Zawartość pliku `header.tsx` w katalogu `src`

```

import React, { Component } from "react";
import { Order } from "../data/entities";
interface Props {
  order: Order
}

export class Header extends Component<Props> {
  render() {
    let count = this.props.order.productCount;
    return <div className="p-1 bg-secondary text-white text-right">
      { count === 0 ? "(brak produktów)"
        : `Liczba produktów: ${ count }, ${ this.props.order.total.toFixed(2) } zł` }
      <button className="btn btn-sm btn-primary m-1">
        Złóż zamówienie
      </button>
    </div>
  }
}

```

Przygotowanie i przetestowanie komponentów

Aby utworzyć komponent wyświetlający nagłówek, listę produktów i przyciski kategorii, dodaj do katalogu `src` plik o nazwie `productList.tsx` i umieść w nim kod przedstawiony na listingu 19.15.

Listing 19.15. Zawartość pliku *productList.tsx* w katalogu *src*

```
import React, { Component } from "react";
import { Header } from "../header";
import { ProductItem } from "../productItem";
import { CategoryList } from "../categoryList";
import { Product, Order } from "../data/entities";

interface Props {
  products: Product[],
  categories: string[],
  order: Order,
  addToOrder: (product: Product, quantity: number) => void
}

interface State {
  selectedCategory: string;
}

export class ProductList extends Component<Props, State> {

  constructor(props: Props) {
    super(props);
    this.state = {
      selectedCategory: "Wszystkie"
    }
  }

  render() {
    return <div>
      <Header order={ this.props.order } />
      <div className="container-fluid">
        <div className="row">
          <div className="col-3 p-2">
            <CategoryList categories={ this.props.categories }
              selected={ this.state.selectedCategory }
              selectCategory={ this.selectCategory } />
          </div>
          <div className="col-9 p-2">
            {
              this.products.map(p =>
                <ProductItem key={ p.id } product={ p }
                  callback={ this.props.addToOrder } />
              )
            }
          </div>
        </div>
      </div>
    </div>
  }

  get products(): Product[] {
    return this.props.products.filter(p => this.state.selectedCategory === "Wszystkie"
      || p.category === this.state.selectedCategory);
  }
}
```

```

    selectCategory = (cat: string) => {
      this.setState({ selectedCategory: cat });
    }
  }
}

```

Komponenty są stosowane za pomocą niestandardowych elementów HTML-a o nazwach odpowiadających nazwom klas tych komponentów. Do konfiguracji komponentów są wykorzystywane właściwości dostarczające dane lub funkcje wywołań zwrotnych, podobnie jak to przedstawiłem w rozdziale 15. podczas tworzenia własnej implementacji JSX. Komponent `ProductList` dostarcza funkcjonalność poprzez połączenie komponentów `Header`, `CategoryList` i `ProductItem`, z których każdy został skonfigurowany za pomocą właściwości otrzymywanych przez komponent `ProductList` lub jego dane stanu.

Aby mieć pewność, że komponenty mogą wyświetlać treść użytkownikowi, zawartość pliku *App.tsx* należy zastąpić przedstawioną na listingu 19.16.

Listing 19.16. Nowa zawartość pliku *App.tsx* w katalogu *src*

```

import React, { Component } from 'react';
import { Product, Order } from './data/entities';
import { ProductList } from './productList';

let testData: Product[] = [1, 2, 3, 4, 5].map(num =>
  ({ id: num, name: `Prod${num}`, category: `Kat${num % 2}`,
    description: `Produkt ${num}`, price: 100}));

interface Props {
  // Polecenia nie są wymagane.
}

interface State {
  order: Order
}

export default class App extends Component<Props, State> {

  constructor(props: Props) {
    super(props);
    this.state = {
      order: new Order()
    }
  }

  render = () =>
    <div className="App">
      <ProductList products={testData}
        categories={this.categories}
        order={this.state.order}
        addToOrder={this.addToOrder} />
    </div>

  get categories(): string[] {
    return [...new Set(testData.map(p => p.category))]
  }
}

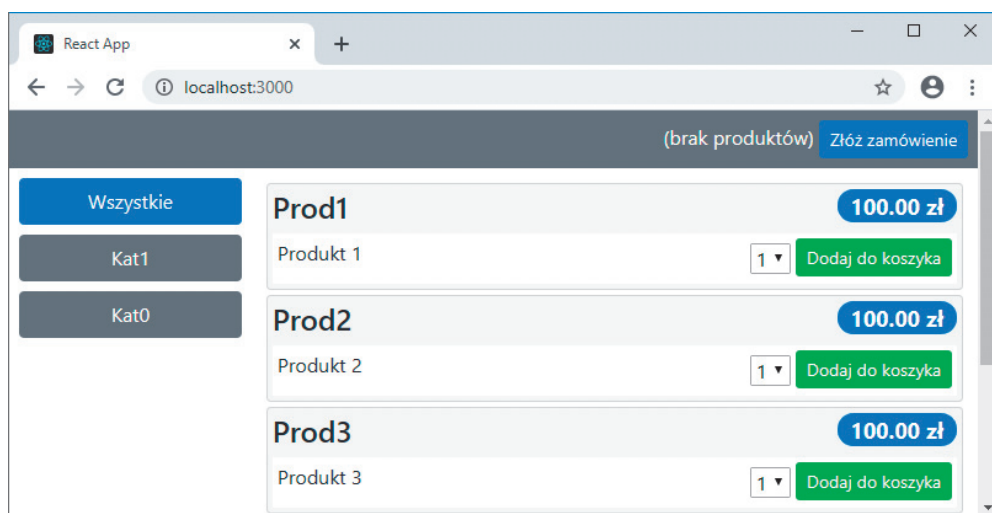
```

```

    addToOrder = (product: Product, quantity: number) => {
      this.setState(state => {
        state.order.addProduct(product, quantity);
        return state;
      })
    }
  }
}

```

Komponent App został uaktualniony do wyświetlania komponentu ProductList, który z kolei wykorzystuje dane testowe. Wprawdzie obsługa dla usługi sieciowej zostanie dodana w dalszej części rozdziału, ale zmiany wprowadzone na listingu 19.16 są wystarczające do wyświetlenia listy produktów, jak pokazałem na rysunku 19.4. (Aby zobaczyć wprowadzone zmiany, stronę w przeglądarce internetowej być może trzeba będzie odświeżyć ręcznie, ponieważ funkcja automatycznego odświeżania bywa zawodna).



Rysunek 19.4. Testowanie komponentu wyświetlającego listę produktów

Tworzenie magazynu danych

W większości projektów opartych na frameworku React danymi aplikacji zarządza magazyn danych. Dostępnych jest wiele pakietów zapewniających funkcjonalność magazynu danych, a najczęściej używanym jest Redux. Jeżeli chcesz dodać pakiety Redux do projektu, z poziomu katalogu *reactapp* w powłoce wydaj polecenia przedstawione na listingu 19.17.

Listing 19.17. Dodawanie kolejnych pakietów do przykładowego projektu

```

$ npm install redux@4.0.5
$ npm install react-redux@7.2.2
$ npm install --save-dev @types/react-redux

```

Pakiet `redux` zawiera deklaracje TypeScriptu. Mimo to konieczny jest pakiet dodatkowy, `react-redux`, pozwalający na połączenie komponentów frameworka React z magazynem danych.

Magazyn danych Redux rozdziela operacje odczytu i modyfikowania danych. Wprawdzie na początku może się to wydawać niewygodne, ale jest zgodne z podejściem stosowanym podczas programowania z użyciem frameworka React, np. w trakcie obsługi danych stanu komponentu, i bardzo szybko można się do tego przyzwyczaić. W magazynie danych Redux *akcja* to obiekt przekazywany do magazynu danych w celu zmiany znajdujących się w nim danych. Akcja ma typ i jest definiowana za pomocą funkcji *tworzenia akcji*. Aby opisać akcje obsługiwane przez magazyn danych, dodaj do katalogu `src/data` plik o nazwie `types.ts` i zawartości przedstawionej na listingu 19.18.

Listing 19.18. Zawartość pliku `types.ts` w katalogu `src/data`

```
import { Product, Order } from "../entities";
import { Action } from "redux";

export interface StoreData {
  products: Product[],
  order: Order
}

export enum ACTIONS {
  ADD_PRODUCTS, MODIFY_ORDER, RESET_ORDER
}

export interface AddProductsAction extends Action<ACTIONS.ADD_PRODUCTS> {
  payload: Product[]
}

export interface ModifyOrderAction extends Action<ACTIONS.MODIFY_ORDER> {
  payload: {
    product: Product,
    quantity: number
  }
}

export interface ResetOrderAction extends Action<ACTIONS.RESET_ORDER> {}

export type StoreAction = AddProductsAction | ModifyOrderAction | ResetOrderAction;
```

-
- **Uwaga** Istnieje wiele różnych sposobów na tworzenie i konfigurowanie magazynu danych oraz łączenie go z komponentami frameworka React. W tym rozdziale wykorzystałem najprostsze, polegające na zdefiniowaniu w oddzielnej klasie obsługi żądań HTTP współdziałających z usługą sieciową. Tutaj nie jest najważniejszy sposób obsługi danych, ale możliwość wykorzystania adnotacji TypeScriptu do opisanego kompilatorowi wybranego podejścia, aby była przeprowadzana operacja sprawdzania typu.
-

Interfejs `StoreData` opisuje dane, którymi będzie zarządzał magazyn danych — to dla niego przykładowa aplikacja definiuje właściwości `products` i `order`.

Wyliczenie `ACTIONS` definiuje zbiór wartości, z których każda odpowiada akcji obsługiwanej przez magazyn danych. Poszczególne wartości wyliczenia są używane jako argument typu dla typu `Action`, który jest interfejsem dostarczonym przez pakiet `Redux`. Interfejs `Action` jest rozszerzony w celu opisywania charakterystyki obiektu dla typów akcji, z których część ma właściwość `payload` dostarczającą dane wymagane do zastosowania tej akcji. Typ `StoreAction` to połączenie interfejsów akcji.

Następnym krokiem jest zdefiniowanie funkcji przygotowania akcji odpowiedzialnej za utworzenie obiektów akcji opisujących operacje, które modyfikują magazyn danych. Do katalogu `src/data` dodaj plik o nazwie `actionCreators.ts` z kodem przedstawionym na listingu 19.19.

Listing 19.19. Zawartość pliku `actionCreators.ts` w katalogu `src/data`

```
import { ACTIONS, AddProductsAction, ModifyOrderAction, ResetOrderAction }
  from "./types";
import { Product } from "./entities";

export const addProduct = (...products: Product[]): AddProductsAction => ({
  type: ACTIONS.ADD_PRODUCTS,
  payload: products
});

export const modifyOrder =
  (product: Product, quantity: number): ModifyOrderAction => ({
    type: ACTIONS.MODIFY_ORDER,
    payload: { product, quantity }
  });

export const resetOrder = (): ResetOrderAction => ({
  type: ACTIONS.RESET_ORDER
});
```

Funkcja zdefiniowana na listingu 19.19 działa w charakterze pomostu między komponentami aplikacji i magazynem danych oraz zapewnia możliwość tworzenia akcji przetwarzanych później przez magazyn danych w celu zastosowania zmian. Akcje są przetwarzane przez funkcje nazywane *reduktorami*, które otrzymują informacje o bieżącym stanie magazynu danych i obiekt akcji opisujący wymaganą zmianę. Aby utworzyć reduktor w przykładowej aplikacji, dodaj do katalogu `src/data` plik `reducer.ts` zawierający kod przedstawiony na listingu 19.20.

Listing 19.20. Zawartość pliku `reducer.ts` w katalogu `src/data`

```
import { ACTIONS, StoreData, StoreAction } from "./types";
import { Order } from "./entities";
import { Reducer } from "redux";

export const StoreReducer: Reducer<StoreData, StoreAction>
  = (data: StoreData | undefined, action) => {

  data = data || { products: [], order: new Order() }
  switch(action.type) {
    case ACTIONS.ADD_PRODUCTS:
      return {
        ...data,
```

```

        products: [...data.products, ...action.payload]
    };

    case ACTIONS.MODIFY_ORDER:
        data.order.addProduct(action.payload.product, action.payload.quantity);
        return { ...data };

    case ACTIONS.RESET_ORDER:
        return {
            ...data,
            order: new Order()
        }
    default:
        return data;
}
}

```

Funkcja reduktora otrzymuje dane znajdujące się obecnie w magazynie danych i akcję, a zwraca zmodyfikowane dane. Ta operacja jest opisywana przez typ `Reducer<S, A>`, gdzie `S` określa typ przedstawiający kształt magazynu danych, natomiast `A` to typ przedstawiający akcje obsługiwane przez ten magazyn. W omawianym przykładzie typem funkcji reduktora jest `Reducer<StoreData, StoreAction>`.

```

...
export const StoreReducer: Reducer<StoreData, StoreAction>
    = (data: StoreData | undefined, action): StoreData => {
...

```

Po wywołaniu funkcji identyfikuje ona akcję za pomocą właściwości `type` dziedziczonej po interfejsie `Action` oraz uaktualnia dane za pomocą właściwości `payload` dla dostarczających ją akcji. Funkcja reduktora będzie wywołana również podczas tworzenia magazynu danych, co daje możliwość zdefiniowania danych początkowych dla aplikacji.

Ostatnim krokiem jest utworzenie magazynu danych, który będzie mógł być używany przez aplikację. Do katalogu `src/data` dodaj plik o nazwie `dataStore.js` z kodem przedstawionym na listingu 19.21.

Listing 19.21. Zawartość pliku `dataStore.ts` w katalogu `src/data`

```

import { createStore, Store } from "redux";
import { StoreReducer } from "../reducer";
import { StoreData, StoreAction } from "../types";

export const dataStore: Store<StoreData, StoreAction> = createStore(StoreReducer);

```

W tym pliku została zdefiniowana metoda Redux `createStore()` przeznaczona do utworzenia obiektu magazynu danych. Ta funkcja jest wyeksportowana, więc można z niej korzystać w całej aplikacji.

Tworzenie klasy żądania HTTP

Magazyn danych Redux pozwala na stosowanie akcji obsługujących żądania HTTP, choć opiera się to na bardziej zaawansowanych funkcjach, które nie ujawniają niczego szczególnego z zakresu TypeScriptu. Aby zachować prostotę przykładu, zamierzam w oddzielnej klasie zdefiniować obsługę żądań HTTP pobierających dane produktów i zamówień składanych przez użytkownika. Framework React nie zawiera zintegrowanej obsługi HTTP, więc z poziomu katalogu *reactapp* w powłoce należy wydać polecenie przedstawione na listingu 19.22, które spowoduje dodanie do projektu pakietu *axios*.

Listing 19.22. Dodawanie kolejnego pakietu do projektu

```
$ npm install axios@0.21.1
```

Po zainstalowaniu niezbędnego pakietu dodaj do katalogu *src/data* plik o nazwie *httpHandler.ts* zawierający kod przedstawiony na listingu 19.23.

Listing 19.23. Zawartość pliku *httpHandler.ts* w katalogu *src/data*

```
import Axios from "axios";
import { Product, Order } from "../entities";

const protocol = "http";
const hostname = "localhost";
const port = 4600;

const urls = {
  products: `${protocol}://${hostname}:${port}/products`,
  orders: `${protocol}://${hostname}:${port}/orders`
};

export class HttpHandler {

  loadProducts(callback: (products: Product[]) => void): void {
    Axios.get(urls.products).then(response => callback(response.data))
  }

  storeOrder(order: Order, callback: (id: number) => void): void {
    let orderData = {
      lines: [...order.orderLines.values()].map(ol => ({
        productId: ol.product.id,
        productName: ol.product.name,
        quantity: ol.quantity
      }))
    }
    Axios.post(urls.orders, orderData)
      .then(response => callback(response.data.id));
  }
}
```


Łączenie komponentów z magazynem danych

Pakiet React-Redux jest odpowiedzialny za połączenie komponentów aplikacji React z magazynem danych Redux. Ten pakiet nie zawiera plików deklaracji TypeScriptu i dlatego też na listingu 19.17 zainstalowałem jeszcze pakiet dodatkowy. W celu połączenia komponentu `ProductList` z magazynem danych dodaj do katalogu `src/data` plik o nazwie `productListConnector.ts` zawierający kod przedstawiony na listingu 19.24.

Listing 19.24. Zawartość pliku `productListConnector.ts` w katalogu `src/data`

```
import { StoreData } from "../types";
import { modifyOrder } from "../actionCreators";
import { connect } from "react-redux";
import { ProductList } from "../productList";

const mapStateToProps = (data: StoreData) => ({
  products: data.products,
  categories: [...new Set(data.products.map(p => p.category))],
  order: data.order
});

const mapDispatchToProps = {
  addToOrder: modifyOrder
};

const connectFunction = connect(mapStateToProps, mapDispatchToProps);
export const ConnectedProductList = connectFunction(ProductList);
```

Proces łączenia przeprowadza mapowanie właściwości danych z magazynu danych i funkcji tworzenia akcji na właściwości komponentu. Wynikiem jest wygenerowanie komponentu, który jest skonfigurowany częściowo za pomocą właściwości używanych po zastosowaniu go w elemencie HTML-a, a częściowo przez magazyn danych. W omawianym przykładzie właściwości `products`, `categories` i `order` są mapowane na właściwości produktu i zamówień w magazynie danych, a właściwość `addToOrder` jest mapowana na funkcję tworzenia akcji `modifyOrder`. Otrzymany w wyniku łączenia komponent, `ConnectedProductList`, pozwala na połączenie komponentu `ProductList` z magazynem danych.

■ **Wskazówka** Zwróć uwagę na to, że podczas mapowania komponentu nie wykorzystałem adnotacji typu. Wprowadźcie typy są dostępne, ale pozostają zawile i dlatego preferuję pozostawienie kompilatorowi zadania ustalenia odpowiednich typów i poinformowania mnie o ewentualnych problemach.

Aby dokończyć proces łączenia z magazynem danych, na listingu 19.25 znajdziesz zmodyfikowaną wersję komponentu `App`, która wybiera magazyn danych, umieszcza w nim dane pochodzące z usługi sieciowej oraz usuwa niepotrzebne już dane testowe i właściwości.

Listing 19.25. Stosowanie magazynu danych w kodzie pliku `App.tsx` w katalogu `src`

```
import React, { Component } from 'react';
//import { Product, Order } from './data/entities';
//import { ProductList } from './productList';
```

```
import { createStore } from "../data/dataStore";
import { Provider } from 'react-redux';
import { HttpHandler } from "../data/httpHandler";
import { addProduct } from '../data/actionCreators';
import { ConnectedProductList } from '../data/productListConnector';

interface Props {
  // Polecenia nie są wymagane.
}

export default class App extends Component<Props> {
  private httpHandler = new HttpHandler();

  // constructor(props: Props) {
  //   super(props);
  //   this.state = {
  //     order: new Order()
  //   }
  // }

  componentDidMount = () => this.httpHandler
    .loadProducts(data => {dataStore.dispatch(addProduct(...data))});

  render = () =>
    <div className="App">
      <Provider store={ createStore }>
        <ConnectedProductList />
      </Provider>
    </div>

  submitCallback = () => {
    console.log("Złóż zamówienie");
  }
}
```

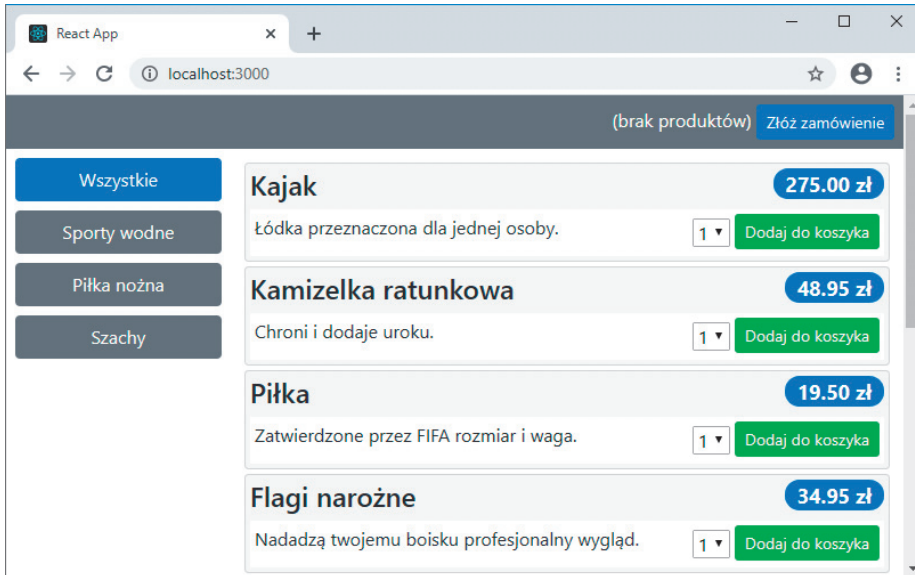
Komponent Provider skonfigurował magazyn danych w taki sposób, że jest on dostępny za pomocą komponentu ConnectedProductList. Pozwala to na wykorzystanie funkcjonalności połączenia.

```
...
<Provider store={ createStore }>
  <ConnectedProductList />
</Provider>
...
```

Magazyn danych może być używany bezpośrednio lub poprzez mapowanie właściwości. W omawianym przykładzie komponent App pobiera dane z usługi sieciowej za pomocą klasy HttpHandler oraz jawnie tworzy i przekazuje akcję odpowiedzialną za uaktualnienie danych w magazynie.

```
...
this.httpHandler.loadProducts(data => dataStore.dispatch(addProduct(...data)));
...
```

W wyniku wprowadzonych modyfikacji dane zostają pobrane z serwera i umieszczone w magazynie danych, który następnie wywołuje operację uaktualnienia powodującą, że komponenty połączone z magazynem danych wyświetlają nowe dane, jak pokazałem na rysunku 19.5.



Rysunek 19.5. Używanie magazynu danych w przykładowej aplikacji

Podsumowanie

W tym rozdziale rozpocząłem pracę nad projektem React używającym języka TypeScript. Wyjaśniłem powody stosowania nietypowej konfiguracji narzędzi programistycznych oraz jej wpływ na konfigurację kompilatora TypeScriptu. Utworzyłem również komponenty aplikacji opartej na frameworku React definiujące funkcjonalność TypeScriptu oraz połączyłem je z prostym magazynem danych Redux. W następnym rozdziale dokończę pracę nad projektem aplikacji React i pokażę, jak przygotować ją do wdrożenia.



Tworzenie aplikacji internetowej React — część II

W rozdziale dokończę proces tworzenia aplikacji React przez dodanie obsługi routingu URL oraz pozostałych komponentów. Na końcu pokażę, jak przygotować aplikację do wdrożenia w kontenerze. W tabeli 20.1 wymienię opcje kompilatora TypeScriptu użyte w rozdziale.

Tabela 20.1. Opcje kompilatora TypeScriptu użyte w rozdziale

Opcja	Opis
<code>allowJs</code>	Ta opcja powoduje dodanie wszystkich plików JavaScriptu w procesie kompilacji
<code>allowSyntheticDefaultImports</code>	Ta opcja powoduje importowanie modułów, które nie deklarują wyraźnie eksportowanych funkcji. Jest używana w celu zwiększenia zgodności kodu źródłowego
<code>esModuleInterop</code>	Ta opcja dodaje kod pomocniczy pozwalający na importowanie funkcjonalności z modułów niedefiniujących domyślnie eksportowanych funkcji i jest używana z opcją <code>allowSyntheticDefaultImports</code>
<code>forceConsistentCasingInFileNames</code>	Ta opcja gwarantuje, że nazwy w poleceniach <code>import</code> będą dopasowane do wielkości znaków użytej w zaimportowanym pliku
<code>isolatedModules</code>	Ta opcja powoduje traktowanie każdego pliku jako oddzielnego modułu, co z kolei zwiększa zgodność z narzędziem Babel
<code>lib</code>	Ta opcja pozwala na wybór używanych przez kompilator plików deklaracji typu
<code>module</code>	Ta opcja określa format używany dla modułów
<code>moduleResolution</code>	Ta opcja określa styl używany podczas rozwiązywania zależności modułów

Tabela 20.1. Opcje kompilatora TypeScriptu użyte w rozdziale (ciąg dalszy)

Opcja	Opis
<code>noEmit</code>	Ta opcja uniemożliwia kompilatorowi emisję kodu JavaScriptu, a wynik jest jedynie sprawdzany pod kątem błędów
<code>resolveJsonModule</code>	Ta opcja pozwala na importowanie plików JSON, jakby były modułami
<code>skipLibCheck</code>	Ta opcja przyspiesza kompilację przez pominięcie standardowo stosowanej operacji sprawdzania plików deklaracji
<code>strict</code>	Ta opcja włącza ściślejsze sprawdzanie kodu TypeScriptu
<code>target</code>	Ta opcja określa docelową wersję języka JavaScript, dla której kompilator wygeneruje kod

Przygotowanie projektu

W rozdziale będę kontynuował pracę nad projektem utworzonym w poprzednim rozdziale. Otwórz nowe okno powłoki, przejdź do katalogu *reactapp*, a następnie wydaj polecenie przedstawione na listingu 20.1, aby w ten sposób uruchomić usługę sieciową i narzędzia programistyczne React.

■ **Wskazówka** Przykładowy projekt dla tego rozdziału, a także projekty dla pozostałych rozdziałów książki zostały umieszczone w archiwum dostępnym pod adresem <https://ftp.helion.pl/przyklady/tys4o2.zip>.

Listing 20.1. Uruchamianie narzędzi programistycznych

```
$ npm start
```

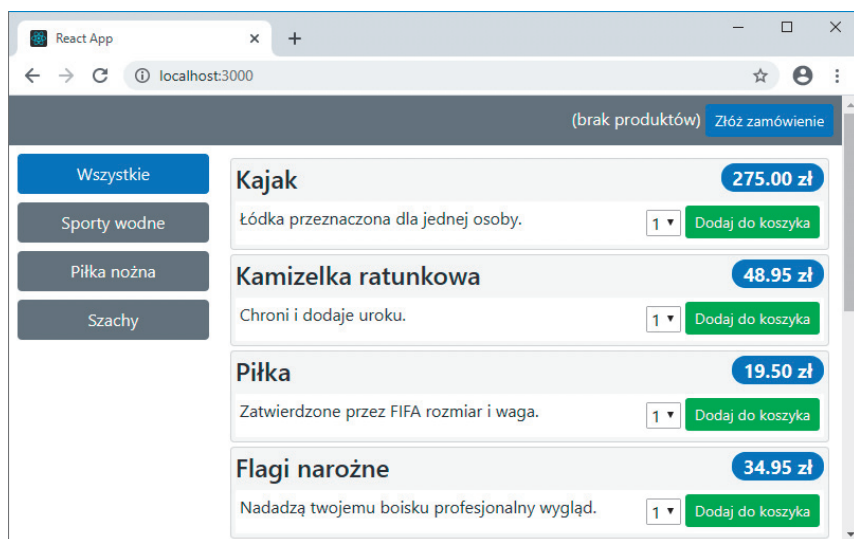
Po zakończeniu kompilacji początkowej otwórz nowe okno przeglądarki WWW i przejdź pod adres <http://localhost:3000>, a zobaczysz uruchomioną aplikację, jak pokazałem na rysunku 20.1.

Konfigurowanie routingu URL

Większość rzeczywistych projektów React wykorzystuje routing URL, czyli używa aktualnego adresu URL w przeglądarce WWW do wyboru komponentu wyświetlanego użytkownikowi. Framework React nie zawiera wbudowanej obsługi routingu; do tego celu najczęściej jest używany pakiet React Router. Z poziomu katalogu *reactapp* w powłoce wydaj polecenia przedstawione na listingu 20.2, aby zainstalować wymieniony pakiet i pliki definicji typu.

Listing 20.2. Dodawanie pakietu do projektu

```
$ npm install react-router-dom@5.2.0
$ npm install --save-dev @types/react-router-dom
```



Rysunek 20.1. Przykładowa aplikacja uruchomiona w przeglądarce WWW

Pakiet React Router zapewnia obsługę różnych systemów nawigacji, a react-router-dom oferuje funkcjonalność wymaganą przez aplikacje internetowe. W tabeli 20.2 wymienię obsługiwane przez przykładową aplikację adresy URL i ich przeznaczenie.

Tabela 20.2. Adresy URL obsługiwane przez aplikację

Adres URL	Opis
/products	Ten adres URL powoduje wyświetlenie komponentu ProductList zdefiniowanego w rozdziale 19.
/order	Ten adres URL powoduje wyświetlenie komponentu odpowiedzialnego za dostarczenie informacji o zamówieniu
/summary	Ten adres URL powoduje wyświetlenie podsumowania zamówienia po jego przekazaniu do serwera. Adres URL będzie zawierał numer zamówienia, więc zamówienie o identyfikatorze 5 zostanie wyświetlone po przejściu pod adres URL /summary/5
/	Domyślny adres URL powoduje przekierowanie pod adres /products, aby został wyświetlony komponent ProductList

Nie wszystkie komponenty wymagane przez aplikację zostały już utworzone, więc na listingu 20.3 zaprezentowałem zmiany niezbędne do skonfigurowania adresów URL /products i /, natomiast pozostałe będą dodawane w kolejnych sekcjach.

Listing 20.3. Konfigurowanie routingu URL w kodzie pliku App.tsx w katalogu src

```
import React, { Component } from 'react';
import { createStore } from './data/dataStore';
import { Provider } from 'react-redux';
import { HttpHandler } from './data/httpHandler';
import { addProduct } from './data/actionCreators';
```

```

import { ConnectedProductList } from './data/productListConnector';
import { Switch, Route, Redirect, BrowserRouter } from "react-router-dom";

interface Props {
  // Polecenia nie są wymagane.
}

export default class App extends Component<Props> {
  private httpHandler = new HttpHandler();

  componentDidMount = () => this.httpHandler
    .loadProducts(data => {dataStore.dispatch(addProduct(...data))});

  render = () =>
    <div className="App">
      <Provider store={ dataStore }>
        <BrowserRouter>
          <Switch>
            <Route path="/products" component={ ConnectedProductList } />
            <Redirect to="/products" />
          </Switch>
        </BrowserRouter>
      </Provider>
    </div>

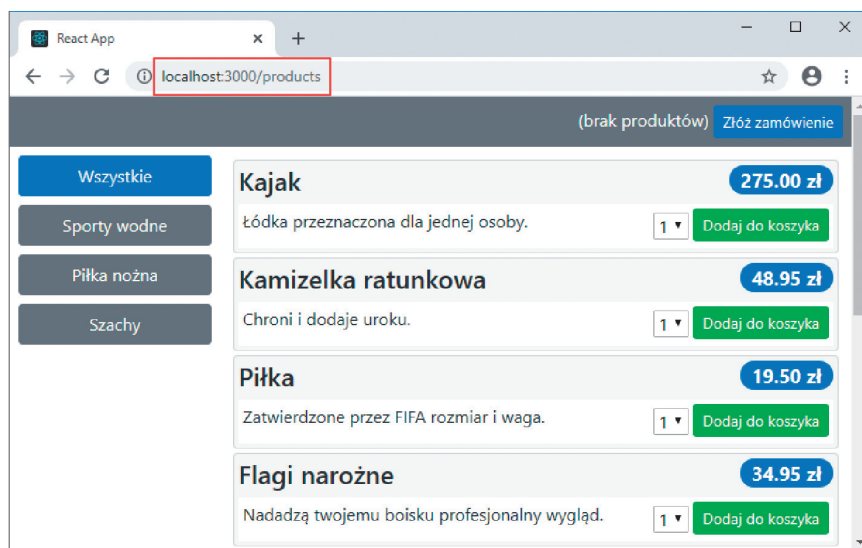
  submitCallback = () => {
    console.log("Złóż zamówienie");
  }
}

```

Podczas konfiguracji pakiet React Router wykorzystuje różne komponenty. Najpierw `BrowserRouter` pozwala na zdefiniowanie treści wybieranej przez użycie adresu URL w przeglądarce WWW. Następnie `Route` tworzy mapowanie między adresem URL a komponentem. Z kolei `Switch` jest odpowiednikiem bloku `switch` w JavaScriptcie i wybiera komponent z pierwszego komponentu Router, którego właściwość zostanie dopasowana do bieżącego adresu URL. Natomiast `Redirect` zapewnia rozwiązanie awaryjne, które spowoduje przekierowanie przeglądarki WWW pod adres URL, jeśli nie zostanie znalezione żadne dopasowanie. Po zapisaniu zmian przedstawionych na listingu 20.3 aplikacja zostanie ponownie skompilowana, a przeglądarka WWW będzie przekierowana pod adres URL `/products`, jak pokazałem na rysunku 20.2.

Dokończenie pracy nad funkcjonalnością aplikacji

Skoro aplikacja może wyświetlać komponenty na podstawie bieżącego adresu URL, do projektu można dodać pozostałe komponenty. W celu umożliwienia poruszania się po adresach URL z poziomu przycisku wyświetlanego przez komponent `Header` dodaj do pliku `header.tsx` polecenia przedstawione na listingu 20.4.



Rysunek 20.2. Dodanie obsługi routingu URL w przykładowej aplikacji

Listing 20.4. Dodawanie nawigacji do kodu pliku `header.tsx` w katalogu `src`

```
import React, { Component } from "react";
import { Order } from "../data/entities";
import { NavLink } from "react-router-dom";

interface Props {
  order: Order
}

export class Header extends Component<Props> {

  render() {
    let count = this.props.order.productCount;
    return <div className="p-1 bg-secondary text-white text-right">
      { count === 0 ? "(brak produktów)"
        : `Liczba produktów: ${ count }, ${ this.props.order.total.toFixed(2)} zł` }
      <NavLink to="/order" className="btn btn-sm btn-primary m-1">
        Złóż zamówienie
      </NavLink>
    </div>
  }
}
```

Komponent `NavLink` powoduje wygenerowanie elementu kotwicy (czyli `<a>`), po którego kliknięciu następuje przejście pod podany adres URL. Klasa Bootstrap CSS zastosowana dla `NavLink` nadaje temu elementowi wygląd przycisku.

Dodawanie komponentu obsługującego podsumowanie zamówienia

W celu wyświetlenia szczegółowych informacji o zamówieniu należy dodać do katalogu *src* plik o nazwie *orderDetails.tsx* z kodem przedstawionym na listingu 20.5.

Listing 20.5. Zawartość pliku *orderDetails.tsx* w katalogu *src*

```
import React, { Component } from "react";
import { StoreData } from "../data/types";
import { Order } from "../data/entities";
import { connect } from "react-redux";
import { NavLink } from "react-router-dom";

const mapStateToProps = (data: StoreData) => ({
  order: data.order
})
interface Props {
  order: Order,
  submitCallback: () => void
}

const connectFunction = connect(mapStateToProps);
export const OrderDetails = connectFunction(
  class extends Component<Props> {
    render() {
      return <div>
        <h3 className="text-center bg-primary text-white p-2">
          ↪ Informacje o zamówieniu</h3>
        <div className="p-3">
          <table className="table table-sm table-striped">
            <thead>
              <tr>
                <th>Ilość</th><th>Produkt</th>
                <th className="text-right">Cena</th>
                <th className="text-right">Wartość</th>
              </tr>
            </thead>
            <tbody>
              { this.props.order.orderLines.map(line =>
                <tr key={ line.product.id }>
                  <td>{ line.quantity }</td>
                  <td>{ line.product.name }</td>
                  <td className="text-right">
                    { line.product.price.toFixed(2) } zł
                  </td>
                  <td className="text-right">
                    { line.total.toFixed(2) } zł
                  </td>
                </tr>
              )}
            </tbody>
            <tfoot>
              <tr>
                <th className="text-right" colSpan={3}>Razem:</th>
```

```

        <th className="text-right">
          { this.props.order.total.toFixed(2) } zł
        </th>
      </tr>
    </tfoot>
  </table>
</div>
<div className="text-center">
  <NavLink to="/products" className="btn btn-secondary m-1">
    Wróć
  </NavLink>
  <button className="btn btn-primary m-1"
    onClick={ this.props.submitCallback }>
    Złóż zamówienie
  </button>
</div>
</div>
  });

```

W rozdziale 19. zapewniłem istniejącemu komponentowi otrzymywanie właściwości połączonych z magazynem danych. Na listingu 20.5 przedstawiłem utworzenie komponentu, który zawsze jest połączony z magazynem danych — dzięki temu unika się konieczności definiowania oddzielnego komponentu, choć jednocześnie tego komponentu nie można używać w przypadku braku magazynu danych, np. w innym projekcie. Przedstawiony komponent wykorzystuje `NavLink` w celu powrotu użytkownika do przycisku `/products` i wywołuje funkcję, gdy użytkownik jest gotowy przekazać zamówienie do usługi sieciowej.

Dodawanie komponentu potwierdzającego złożenie zamówienia

W celu wyświetlenia użytkownikowi komunikatu po złożeniu zamówienia za pomocą usługi sieciowej należy dodać do katalogu `src` plik o nazwie `summary.tsx` z kodem przedstawionym na listingu 20.6.

Listing 20.6. Zawartość pliku `summary.tsx` w katalogu `src`

```

import React, { Component } from "react";
import { match } from "react-router";
import { NavLink } from "react-router-dom";

interface Params {
  id: string;
}

interface Props {
  match: match<Params>
}

export class Summary extends Component<Props> {
  render() {
    let id = this.props.match.params.id;
    return <div className="m-2 text-center">
      <h2>Dziękujemy!</h2>
    </div>
  }
}

```

```

        <p>Dziękujemy za złożenie zamówienia.</p>
        <p>Numer zamówienia #{ id }</p>
        <p>Zamówione produkty zostaną wkrótce wysłane.</p>
        <NavLink to="/products" className="btn btn-primary">OK</NavLink>
    </div>
  }
}

```

Komponent `Summary` musi otrzymać jedynie przypisany przez usługę sieciową numer zamówienia, który jest pobierany z bieżącej trasy. Pakiet routingu dostarcza za pomocą właściwości szczegółowe informacje o trasie, stosując się do istniejącego wzorca frameworka React. Deklaracje typu dla pakietu React Router są używane w celu opisanie parametru oczekiwanego przez komponent, co pozwala kompilatorowi TypeScriptu na sprawdzanie typów.

Dokończenie konfiguracji routingu

Na listingu 20.7 widać dodanie nowych elementów `Route` pozwalających na wyświetlanie komponentów `OrderDetails` i `Summary`, które pozwalają na dokończenie konfiguracji routingu w przykładowej aplikacji.

Listing 20.7. Dodawanie pozostałych tras w kodzie pliku `App.tsx` w katalogu `src`

```

import React, { Component } from 'react';
import { createStore } from './data/dataStore';
import { Provider } from 'react-redux';
import { HttpHandler } from './data/httpHandler';
import { addProduct } from './data/actionCreators';
import { ConnectedProductList } from './data/productListConnector';
import { Switch, Route, Redirect, BrowserRouter, RouteComponentProps }
  from "react-router-dom";
import { OrderDetails } from './orderDetails';
import { Summary } from './summary';

interface Props {
  // Polecenia nie są wymagane.
}

export default class App extends Component<Props> {
  private httpHandler = new HttpHandler();

  componentDidMount = () => this.httpHandler
    .loadProducts(data => {dataStore.dispatch(addProduct(...data))});

  render = () =>
    <div className="App">
      <Provider store={dataStore}>
        <BrowserRouter>
          <Switch>
            <Route path="/products" component={ConnectedProductList} />
            <Route path="/order" render={ (props) =>
              <OrderDetails { ...props } submitCallback={ () =>
                this.submitCallback(props) } />
            } />
          </Switch>
        </BrowserRouter>
      </Provider>
    </div>

```

```

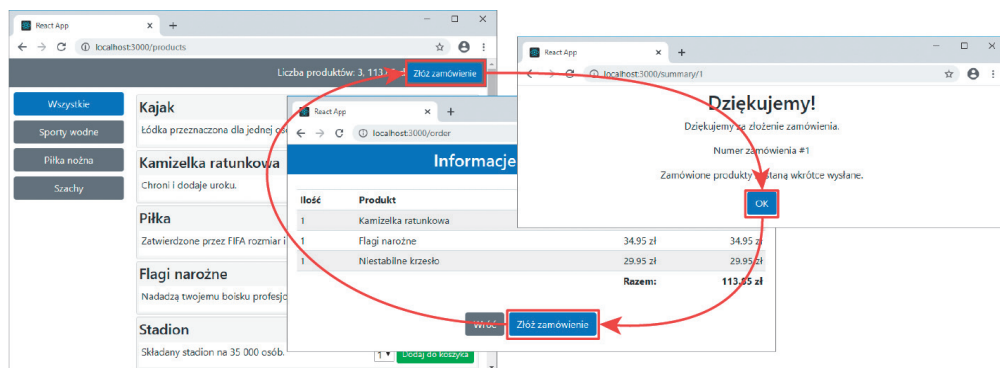
    } />
    <Route path="/summary/:id" component={ Summary } />
    <Redirect to="/products" />
  </Switch>
</BrowserRouter>
</Provider>
</div>

submitCallback = (routeProps: RouteComponentProps) => {
  this.httpHandler.storeOrder(dataStore.getState().order,
    id => routeProps.history.push( `~/summary/${id}`));
}
}

```

Komponent Route dla komponentu OrderDetails używa funkcji render() do wyboru komponentu i zapewnienia mu właściwości dostarczanych przez system routingu i funkcję wywołania zwrotnego. Metoda submitCallback() wymaga dostępu do funkcjonalności routingu dostarczanych jako właściwości komponentów, aby w ten sposób przejść pod nowy adres URL, ale są one dostępne jedynie w komponencie routera Browser. W celu obejścia tego ograniczenia dostarczyłem komponent OrderDetails z funkcją wewnętrzną przekazującą właściwości routingu metodzie submitCallback(), co pozwala na użycie metody history.push(). Komponent Route dla komponentu Summary definiuje adres URL z parametrem dostarczającym numer zamówienia wyświetlany użytkownikowi.

Po zapisaniu zmiany produkty mogą być dodawane do zamówienia, które następnie można przekazać do usługi sieciowej, jak pokazałem na rysunku 20.3.



Rysunek 20.3. Dokończenie pracy nad funkcjonalnością przykładowej aplikacji

Wdrażanie aplikacji

Narzędzia programistyczne frameworka React opierają swoje działanie na serwerze Webpack Development Server, który nie jest odpowiedni do stosowania w środowisku produkcyjnym, ponieważ dodaje funkcje takie jak automatyczne odświeżanie strony po wygenerowaniu paczki JavaScriptu. W tym podrozdziale przedstawię proces przygotowania aplikacji React do wdrożenia — jest on podobny do zastosowanego podczas wdrażania pozostałych aplikacji internetowych, także tych utworzonych za pomocą innych frameworków.

Dodawanie pakietu produkcyjnego serwera HTTP

W przypadku środowiska produkcyjnego wymagany jest zwykle serwer HTTP, który będzie dostarczał pliki HTML-a, CSS i JavaScriptu przeglądarce WWW. W omawianym przykładzie zdecydowałem się na użycie serwera Express — znajduje się on w tym samym pakiecie, z którego korzystam we wszystkich przykładach w tej części książki. Ten serwer jest dobrym wyborem dla wielu aplikacji internetowych. Zatrzymaj działanie narzędzi programistycznych React przez naciśnięcie klawiszy *Ctrl+C*, a następnie z poziomu katalogu *reactapp* aplikacji w powłoce wydaj polecenia przedstawione na listingu 20.8, aby zainstalować niezbędne pakiety.

Listing 20.8. Dodawanie pakietów pozwalających na wdrożenie aplikacji React

```
$ npm install --save-dev express@4.17.1
$ npm install --save-dev connect-history-api-fallback@1.6.0
```

Drugie polecenie powoduje zainstalowanie pakietu *connect-history-api-fallback*, który okazuje się użyteczny podczas wdrażania aplikacji wykorzystujących routing URL. Działanie pakietu polega na mapowaniu obsługiwanych przez aplikację żądań prowadzących do pliku *index.html* i zagwarantowaniu, że odświeżenie strony w przeglądarce WWW nie spowoduje wyświetlenia użytkownikowi błędu informującego o nieznalezieniu zasobu.

Tworzenie pliku dla trwałego magazynu danych

W celu utworzenia trwałego magazynu danych dla usługi sieciowej dodaj do katalogu *reactapp* plik o nazwie *data.json* zawierający kod przedstawiony na listingu 20.9.

Listing 20.9. Zawartość pliku *data.json* w katalogu *reactapp*

```
{
  "products": [
    { "id": 1, "name": "Kajak", "category": "Sporty wodne",
      "description": "Łódka przeznaczona dla jednej osoby.", "price": 275 },
    { "id": 2, "name": "Kamizelka ratunkowa", "category": "Sporty wodne",
      "description": "Chroni i dodaje uroku.", "price": 48.95 },
    { "id": 3, "name": "Piłka", "category": "Piłka nożna",
      "description": "Zatwierdzone przez FIFA rozmiar i waga.", "price": 19.50 },
    { "id": 4, "name": "Flagi narożne", "category": "Piłka nożna",
      "description": "Nadadzą twojemu boisku profesjonalny wygląd.",
      "price": 34.95 },
    { "id": 5, "name": "Stadion", "category": "Piłka nożna",
      "description": "Składany stadion na 35 000 osób.", "price": 79500 },
    { "id": 6, "name": "Czapka", "category": "Szachy",
      "description": "Zwiększa efektywność mózgu o 75%.", "price": 16 },
    { "id": 7, "name": "Niestabilne krzesło", "category": "Szachy",
      "description": "Zmniejsza szanse przeciwnika.",
      "price": 29.95 },
    { "id": 8, "name": "Ludzka szachownica", "category": "Szachy",
      "description": "Przyjemna gra dla całej rodziny.", "price": 75 },
```

```
{ "id": 9, "name": "Błyszczący król", "category": "Szachy",
  "description": "Pokryty złotem i wysadzany diamentami król.", "price": 1200 }
],
"orders": []
}
```

Tworzenie serwera

Aby utworzyć serwer dostarczający przeglądarce WWW aplikację i jej dane, dodaj do katalogu *reactapp* plik o nazwie *server.js* i umieść w nim kod przedstawiony na listingu 20.10.

Listing 20.10. Zawartość pliku *server.js* w katalogu *reactapp*

```
const express = require("express");
const jsonServer = require("json-server");
const history = require("connect-history-api-fallback");

const app = express();
app.use(history());
app.use("/", express.static("build"));

const router = jsonServer.router("data.json");
app.use(jsonServer.bodyParser)
app.use("/api", (req, resp, next) => router(req, resp, next));

const port = process.argv[3] || 4002;
app.listen(port, () => console.log(`Serwer nasłuchuje na porcie numer ${port}`));
```

Polecenia zdefiniowane w pliku *server.js* konfiguruja pakiety *express* i *json-server* w taki sposób, aby udostępniać zawartość katalogu *build*, w którym React umieszcza pliki JavaScriptu i HTML-a oraz nakazuje przeglądarce WWW ich wczytanie. Adresy URL poprzedzone prefiksem */api* będą obsługiwane przez usługę sieciową.

Używanie względnych adresów URL do obsługi żądań danych

Usługa sieciowa dostarczająca dane aplikacji będzie działała równolegle z serwerem frameworka React. Aby przygotować aplikację na wykonywanie żądań za pomocą pojedynczego portu, wymagana jest zmiana kodu klasy *HttpHandler*, jak pokazałem na listingu 20.11.

Listing 20.11. Używanie względnych adresów URL w kodzie pliku *httpHandler.ts* w katalogu *src/data*

```
import Axios from "axios";
import { Product, Order } from "../entities";

// const protocol = document.location.protocol;
// const hostname = document.location.hostname;
// const port = 4600;

const urls = {
  // products: `${protocol}/${hostname}:${port}/products`,
```

```
// orders: `${protocol}/${hostname}:${port}/orders`
  products: "/api/products",
  orders: "/api/orders"
};

export class HttpHandler {

  loadProducts(callback: (products: Product[]) => void): void {
    Axios.get(urls.products).then(response => callback(response.data))
  }

  storeOrder(order: Order, callback: (id: number) => void): void {
    let orderData = {
      lines: [...order.orderLines.values()].map(ol => ({
        productId: ol.product.id,
        productName: ol.product.name,
        quantity: ol.quantity
      })))
    }
    Axios.post(urls.orders, orderData)
      .then(response => callback(response.data.id));
  }
}
```

Podane adresy URL są względne dla używanych do obsługi żądań dokumentu HTML-a. Oznacza to stosowanie się do konwencji, zgodnie z którą żądania danych są poprzedzone prefiksem */api*.

Kompilowanie aplikacji

Z poziomu katalogu *reactapp* w powłoce należy wydać polecenie przedstawione na listingu 20.12, które spowoduje utworzenie produkcyjnej wersji aplikacji.

Listing 20.12. Tworzenie paczki produkcyjnej

```
$ npm run build
```

W wyniku procesu kompilacji nastąpi utworzenie w katalogu *build* zestawu zoptymalizowanych plików. Kompilacja może chwilę potrwać i powoduje wygenerowanie danych wyjściowych podobnych do tutaj przedstawionych, które zawierają informacje o plikach użytych podczas przygotowywania paczki.

```
Creating an optimized production build...
```

```
Compiled successfully.
```

```
File sizes after gzip:
```

```
59.35 KB  build\static\js\2.943d36b9.chunk.js
22.63 KB  build\static\css\2.658248ec.chunk.css
2.52 KB   build\static\js\main.51b0a5f5.chunk.js
1.39 KB   build\static\js\3.03f9fbbc.chunk.js
1.16 KB   build\static\js\runtime-main.e80fc4bd.js
278 B     build\static\css\main.6dea0f05.chunk.css
```

The project was built assuming it is hosted at /.
 You can control this with the homepage field in your package.json.
 The build folder is ready to be deployed.
 You may serve it with a static server:
 npm install -g serve
 serve -s build
 Find out more about deployment here:
<https://cra.link/deployment>

Testowanie gotowej aplikacji

Aby upewnić się o prawidłowej kompilacji aplikacji i zastosowaniu wszystkich wprowadzonych zmian konfiguracyjnych, z poziomu katalogu *reactapp* w powłoce wydaj polecenie przedstawione na listingu 20.13.

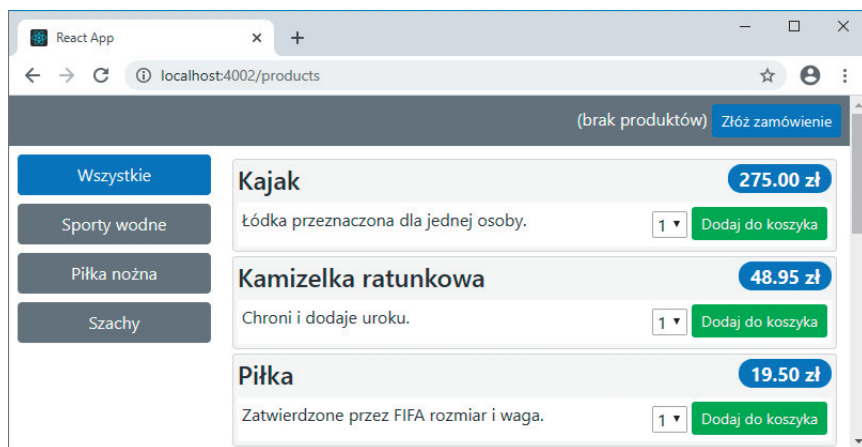
Listing 20.13. Uruchamianie serwera produkcyjnego

```
$ node server.js
```

Kod przedstawiony na listingu 20.13 zostanie wykonany i spowoduje wygenerowanie następujących danych wyjściowych:

Serwer nasłuchuje na porcie numer 4002

Otwórz okno przeglądarki WWW i przejdź pod adres *http://localhost:4002*, a zobaczysz działającą aplikację, jak pokazałem na rysunku 20.4.



Rysunek 20.4. Uruchomiona aplikacja przeznaczona dla środowiska produkcyjnego

Umieszczanie aplikacji w kontenerze

Ten rozdział zamierzam zakończyć utworzeniem dla przykładowej aplikacji kontenera, który następnie będzie można wdrożyć w środowisku produkcyjnym. Jeżeli w rozdziale 15. nie zainstalowałeś Dockera, będziesz to musiał zrobić teraz, jeśli chcesz wykonywać przykłady przedstawione w pozostałej części rozdziału.

Przygotowanie aplikacji

Pracę należy zacząć od utworzenia pliku konfiguracyjnego dla menedżera pakietów Node.js, który będzie użyty do pobrania pakietów dodatkowych wymaganych przez aplikację uruchamianą w kontenerze. W katalogu *reactapp* utwórz więc plik o nazwie *deploy-package.json* i umieść w nim kod przedstawiony na listingu 20.14.

Listing 20.14. Zawartość pliku *deploy-package.json* w katalogu *reactapp*

```
{
  "name": "reactapp",
  "description": "Aplikacja internetowa React",
  "repository": "https://github.com/Apress/essential-typescript",
  "license": "0BSD",
  "devDependencies": {
    "express": "4.17.1",
    "json-server": "0.16.3",
    "connect-history-api-fallback": "1.6.0"
  }
}
```

Sekcja *devDependencies* wymienia pakiety niezbędne do uruchomienia aplikacji w kontenerze. Wszystkie pakiety, dla których istnieją polecenia *import* w plikach kodu źródłowego aplikacji, zostały umieszczone w pliku paczki wygenerowanym przez pakiet *webpack*. Pozostałe opcje opisują aplikację i zostały umieszczone w kodzie w celu uniknięcia komunikatów ostrzeżeń generowanych podczas tworzenia kontenera.

Tworzenie kontenera Dockera

Aby zdefiniować kontener, należy dodać do katalogu *reactapp* plik o nazwie *Dockerfile* (bez żadnego rozszerzenia) i umieścić w nim kod przedstawiony na listingu 20.15.

Listing 20.15. Zawartość pliku *Dockerfile* w katalogu *reactapp*

```
FROM node:14.15.4

RUN mkdir -p /usr/src/reactapp

COPY build /usr/src/reactapp/build/
COPY data.json /usr/src/reactapp/
COPY server.js /usr/src/reactapp/
```

```
COPY deploy-package.json /usr/src/reactapp/package.json
```

```
WORKDIR /usr/src/reactapp
```

```
RUN echo 'package-lock=false' >> .npmrc
```

```
RUN npm install
```

```
EXPOSE 4002
```

```
CMD ["node", "server.js"]
```

Zawartość pliku *Dockerfile* wykorzystuje obraz bazowy skonfigurowany z Node.js i zawierający skopiowane pliki niezbędne do uruchomienia aplikacji, m.in. plik paczki z aplikacją oraz plik używany do zainstalowania pakietów Node.js wymaganych do uruchomienia aplikacji w środowisku produkcyjnym.

Aby przyspieszyć proces tworzenia kontenera, do katalogu *reactapp* warto również dodać plik *.dockerignore* o zawartości przedstawionej na listingu 20.16. W omawianym przykładzie ten plik nakazuje Dockerowi zignorowanie katalogu *node_modules*, który nie jest wymagany w kontenerze, a jego przetworzenie będzie wymagało ogromnej ilości czasu.

Listing 20.16. Zawartość pliku *.dockerignore* w katalogu *reactapp*

```
node_modules
```

Z poziomu katalogu *reactapp* w powłoce wydaj polecenie przedstawione na listingu 20.17, które rozpocznie proces tworzenia kontenera zawierającego przykładową aplikację i wszystkie wymagane pakiety.

Listing 20.17. Tworzenie obrazu Dockera

```
$ docker build . -t reactapp -f Dockerfile
```

Obraz jest szablonem dla kontenera. Docker przetworzy polecenia zdefiniowane w pliku *Dockerfile*, wskazane pakiety Node.js zostaną pobrane i zainstalowane, a pliki konfiguracyjne i kodu źródłowego będą skopiowane do obrazu.

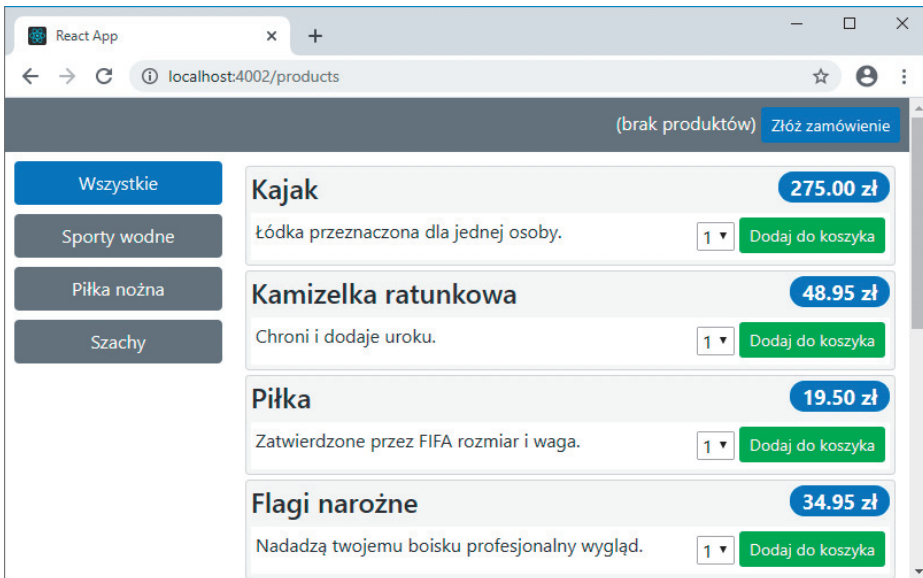
Uruchamianie aplikacji

Gdy obraz Dockera jest gotowy, można uruchomić nowy kontener za pomocą polecenia wymienionego na listingu 20.18.

Listing 20.18. Uruchamianie kontenera Dockera

```
$ docker run -p 4002:4002 reactapp
```

Aby przetestować aplikację, należy przejść w przeglądarce WWW pod adres *http://localhost:4002*, co spowoduje wyświetlenie odpowiedzi udzielonej przez serwer WWW uruchomiony w kontenerze, jak pokazałem na rysunku 20.5.



Rysunek 20.5. Przykładowa aplikacja uruchomiona w kontenerze

Jeżeli chcesz zatrzymać działanie kontenera, musisz zacząć od wydania polecenia przedstawionego na listingu 20.19.

Listing 20.19. Wyświetlenie listy kontenerów Dockera

```
$ docker ps
```

Dane wyjściowe tego polecenia to lista uruchomionych kontenerów (w celu zachowania zwięzłości i czytelności pominąłem część danych wyjściowych):

CONTAINER ID	IMAGE	COMMAND	CREATED
82352eba95a2	reactapp	"docker-entrypoint.s..."	51 seconds ago

Wykorzystując wartość wyświetloną w kolumnie *CONTAINER ID*, wydaj polecenie przedstawione na listingu 20.20.

Listing 20.20. Zatrzymywanie kontenera Dockera

```
$ docker stop 82352eba95a2
```

W tym momencie aplikacja jest gotowa do wdrożenia na dowolnej platformie obsługującej kontenery Dockera.

Podsumowanie

W tym rozdziale dokończyłem pracę nad przykładową aplikacją React przez wykorzystanie funkcjonalności routingu URL i dodanie pozostałych komponentów. Podobnie jak w przypadku pozostałych przykładowych aplikacji w tej części książki, zobaczyłeś, jak wygląda proces przygotowania jej do wdrożenia w środowisku produkcyjnym. Ponadto przygotowałem obraz Dockera z przykładową aplikacją, co pozwala na łatwe wdrożenie. W następnym rozdziale zajmę się tworzeniem tej samej aplikacji internetowej, ale tym razem za pomocą frameworka Vue.js i TypeScriptu.



Tworzenie aplikacji internetowej Vue.js — część I

W tym rozdziale rozpocznę proces tworzenia przykładowej aplikacji internetowej za pomocą Vue.js, który jest najnowszym z trzech głównych frameworków przedstawionych w tej części książki. Mimo to ma entuzjastyczną i wierną bazę użytkowników. W tabeli 21.1 wymienię opcje kompilatora TypeScriptu użyte w rozdziale.

Tabela 21.1. Opcje kompilatora TypeScriptu użyte w rozdziale

Opcja	Opis
<code>allowSyntheticDefaultImports</code>	Ta opcja powoduje importowanie modułów, które nie deklarują wyraźnie eksportowanych funkcji. Jest używana w celu zwiększenia zgodności kodu źródłowego
<code>baseUrl</code>	Ta opcja określa katalog główny używany podczas rozwiązywania zależności modułów
<code>esModuleInterop</code>	Ta opcja dodaje kod pomocniczy pozwalający na importowanie funkcjonalności z modułów niedefiniujących domyślnie eksportowanych funkcji i jest używana z opcją <code>allowSyntheticDefaultImports</code>
<code>importHelpers</code>	Ta opcja określa, czy kod pomocniczy zostanie dodany do JavaScriptu w celu zmniejszenia ogólnej ilości wygenerowanego kodu
<code>jsx</code>	Ta opcja określa sposób przetwarzania elementów HTML-a w plikach TSX
<code>lib</code>	Ta opcja pozwala na wybór używanych przez kompilator plików deklaracji typu
<code>module</code>	Ta opcja określa format używany dla modułów

Tabela 21.1. Opcje kompilatora TypeScriptu użyte w rozdziale (ciąg dalszy)

Opcja	Opis
<code>moduleResolution</code>	Ta opcja określa styl używany podczas rozwiązywania zależności modułów
<code>paths</code>	Ta opcja określa katalogi używane podczas rozwiązywania zależności modułu
<code>skipLibCheck</code>	Ta opcja przyspiesza kompilację przez pominięcie standardowo stosowanej operacji sprawdzania plików deklaracji
<code>sourceMap</code>	Ta opcja określa, czy kompilator ma generować pliki map źródłowych stosowane podczas debugowania
<code>strict</code>	Ta opcja włącza ściślejsze sprawdzanie kodu TypeScriptu
<code>target</code>	Ta opcja określa docelową wersję języka JavaScript, dla której kompilator wygeneruje kod
<code>types</code>	Ta opcja określa listę plików deklaracji dołączanych w trakcie procesu kompilacji

Przygotowanie projektu

Projekt oparty na frameworku Vue.js najłatwiej jest utworzyć za pomocą działającego z poziomu powłoki pakietu Vue zapewniającego możliwość zdefiniowania projektu Vue.js z obsługą języka TypeScript. W powłoce wydaj polecenie przedstawione na listingu 21.1, aby w ten sposób zainstalować niezbędny pakiet.

■ **Wskazówka** Przykładowy projekt dla tego rozdziału, a także projekty dla pozostałych rozdziałów książki zostały umieszczone w archiwum dostępnym pod adresem <https://ftp.helion.pl/przyklady/tys4o2.zip>.

Listing 21.1. Instalowanie pakietu pozwalającego na utworzenie projektu wykorzystującego framework Vue.js

```
$ npm install --global @vue/cli@4.5.11
```

Pierwszy znak @ jest częścią nazwy pakietu: @vue-cli. Z kolei drugi znak @ to separator między nazwą pakietu a jego wymaganą nazwą: 4.5.11.

Po zainstalowaniu pakietu przejdź do katalogu, w którym chcesz utworzyć aplikację Vue.js, a następnie wydaj polecenie przedstawione na listingu 21.2, aby rozpocząć przygotowywanie nowego projektu.

Listing 21.2. Tworzenie nowego projektu

```
$ vue create vueapp
```

Proces konfiguracji pakietu jest interaktywny. W jego trakcie będziesz musiał odpowiedzieć na kilka pytań — podczas udzielania odpowiedzi możesz się posilkować tabelą 21.2.

Tabela 21.2. Pytania pojawiające się w trakcie tworzenia projektu *Vue.js* i odpowiedzi na nie

Pytanie	Odpowiedź
Czy chcesz skorzystać z wstępnie przygotowanych ustawień?	Nie, wybierz niezbędną funkcjonalność ręcznie
Jakie komponenty będą potrzebne w projekcie?	Babel, TypeScript, router, Vuex. Nie wybieraj lintera Po wskazaniu wymienionych komponentów wybierz wersję Vue 3.x (w czasie powstawania książki obsługa tej wersji jest w fazie eksperymentalnej)
Czy użyć dla komponentu składni w stylu klasy?	Nie
Czy używać kompilatorów Babel i TypeScriptu?	Tak
Czy używać trybu historii dla routera?	Tak
Gdzie ma być umieszczona konfiguracja kompilatorów Babel, PostCSS, ESL i innych komponentów?	W przeznaczonych do tego celu plikach konfiguracyjnych
Czy zapisać te informacje jako ustawienia na przyszłość?	Nie
Który menedżer pakietów ma zostać użyty podczas instalowania zależności?	npm

Po udzieleniu odpowiedzi na te pytania projekt zostanie utworzony, a następnie będą zainstalowane pakiety wymagane przez ten projekt.

Konfigurowanie usługi sieciowej

Po utworzeniu projektu wydaj polecenia przedstawione na listingu 21.3, aby przejść do katalogu projektu i dodać pakiety zapewniające obsługę usługi sieciowej oraz pozwalające na uruchamianie wielu pakietów za pomocą jednego polecenia.

Listing 21.3. Dodawanie pakietów do projektu

```
$ cd vueapp
$ npm install --save-dev json-server@0.16.3
$ npm install --save-dev npm-run-all@4.1.5
$ npm install axios@0.21.1
```

W celu przygotowania danych dla usługi sieciowej dodaj do katalogu *vueapp* plik o nazwie *data.js* zawierający kod przedstawiony na listingu 21.4.

Listing 21.4. Zawartość pliku *data.js* w katalogu *vueapp*

```
module.exports = function () {
  return {
```

```

products: [
  { id: 1, name: "Kajak", category: "Sporty wodne",
    description: "Łódka przeznaczona dla jednej osoby.", price: 275 },
  { id: 2, name: "Kamizelka ratunkowa", category: "Sporty wodne",
    description: "Chroni i dodaje uroku.", price: 48.95 },
  { id: 3, name: "Piłka", category: "Piłka nożna",
    description: "Zatwierdzone przez FIFA rozmiar i waga.", price: 19.50 },
  { id: 4, name: "Flagi narożne", category: "Piłka nożna",
    description: "Nadadzą twojemu boisku profesjonalny wygląd.",
    price: 34.95 },
  { id: 5, name: "Stadion", category: "Piłka nożna",
    description: "Składany stadion na 35 000 osób.", price: 79500 },
  { id: 6, name: "Czapka", category: "Szachy",
    description: "Zwiększa efektywność mózgu o 75%.", price: 16 },
  { id: 7, name: "Niestabilne krzesło", category: "Szachy",
    description: "Zmniejsza szanse przeciwnika.",
    price: 29.95 },
  { id: 8, name: "Ludzka szachownica", category: "Szachy",
    description: "Przyjemna gra dla całej rodziny.", price: 75 },
  { id: 9, name: "Błyszczący król", category: "Szachy",
    description: "Pokryty złotem i wysadzany diamentami król.", price: 1200 }
],
orders: []
}
}

```

Uaktualnij sekcję *scripts* pliku *package.json* i tym samym skonfiguruj narzędzia programistyczne w taki sposób, aby zbiór narzędzi Vue.js i usługa sieciowa były uruchamiane jednocześnie. Niezbędne zmiany w wymienionym pliku przedstawiłem na listingu 21.5.

Listing 21.5. Konfigurowanie narzędzi w pliku *package.json* w katalogu *vueapp*

```

...
"scripts": {
  "start": "npm-run-all -p serve json",
  "json": "json-server data.js -p 4600",
  "serve": "vue-cli-service serve",
  "build": "vue-cli-service build"
},
...

```

Wprowadzone zmiany pozwalają na uruchamianie za pomocą tylko jednego polecenia zarówno usługi sieciowej dostarczającej dane, jak i zestawu narzędzi programistycznych Vue.js.

Instalowanie pakietu Bootstrap CSS

Polecenie przedstawione na listingu 21.6 wydane z poziomu katalogu *vueapp* w powłoce powoduje dodanie do projektu obsługi frameworka Bootstrap CSS.

Listing 21.6. Dodawanie do projektu pakietu Bootstrap CSS

```
$ npm install bootstrap@4.6.0
```

Narzędzia programistyczne Vue.js wymagają zmiany konfiguracyjnej pozwalającej na wykorzystanie arkusza stylów Bootstrap CSS w aplikacji. Przejdź do pliku *main.ts* w katalogu *src*, a następnie wprowadź zmianę przedstawioną na listingu 21.7.

Listing 21.7. Dodawanie arkusza stylów w pliku *main.ts* w katalogu *src*

```
import { createApp } from 'vue'
import App from './App.vue'
import router from './router'
import store from './store'
import "bootstrap/dist/css/bootstrap.min.css";

createApp(App).use(store).use(router).mount('#app')
```

Uruchamianie przykładowej aplikacji

Polecenie wymienione na listingu 21.8 wydane z poziomu katalogu *vueapp* w powłoce spowoduje uruchomienie przykładowej aplikacji.

Listing 21.8. Uruchamianie narzędzi programistycznych

```
$ npm start
```

Nastąpi uruchomienie narzędzi programistycznych Vue.js i rozpoczęcie początkowej kompilacji projektu, co spowoduje wygenerowanie następujących danych wyjściowych:

```
...
DONE Compiled successfully in                                17:12:21
2905ms

App running at:
- Local:   http://localhost:8080/
- Network: http://192.168.1.10:8080/

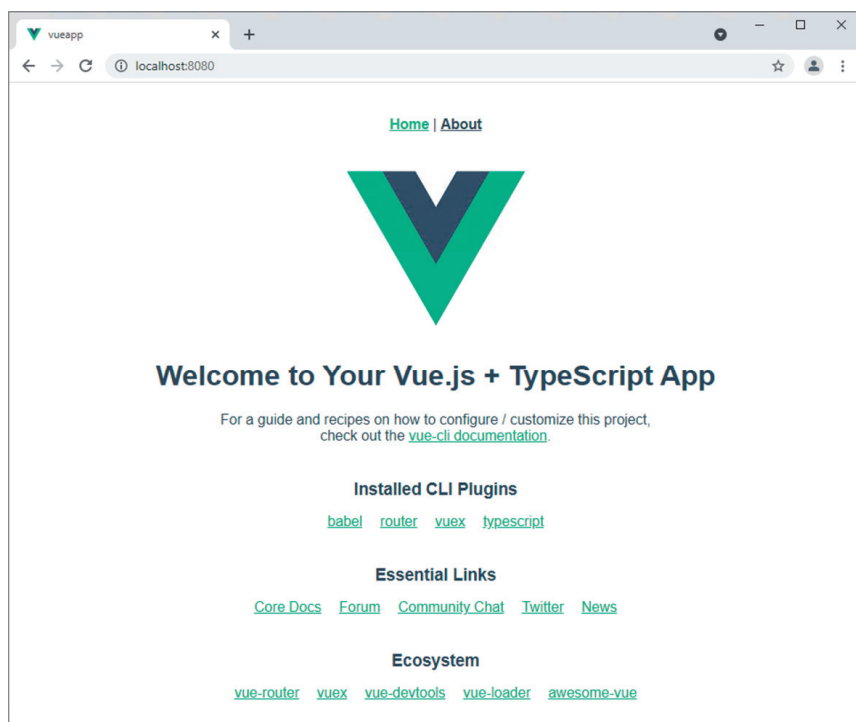
Note that the development build is not optimized.
To create a production build, run npm run build.

No issues found.
...
```

Po zakończeniu początkowej kompilacji otwórz nowe okno przeglądarki WWW i przejdź pod adres *http://localhost:8080*, aby zobaczyć miejsce zarezerwowane na treść, które zostało przygotowane przez polecenie na listingu 21.2, jak pokazałem na rysunku 21.1.

TypeScript i programowanie w Vue.js

Język TypeScript nie jest wymagany podczas programowania z użyciem frameworka Vue.js, ale ze względu na to, że jest często wybierany, podstawowe pakiety Vue.js zawierają pliki deklaracji typów, a pakiet Vue CLI ma możliwość tworzenia projektów już skonfigurowanych do obsługi TypeScriptu.



Rysunek 21.1. Wynik uruchomienia przykładowej aplikacji

Pliki Vue.js wykorzystujące funkcje oferowane przez TypeScript mają rozszerzenie *.vue*, mogą zawierać elementy `<template>`, `<style>` i `<script>`; powszechnie są określane mianem *komponentów w postaci pojedynczego pliku*. Element `<template>` zawiera szablon przeznaczony do wygenerowania treści HTML, element `<style>` zawiera style CSS dla treści, a element `<script>` zawiera kod wspomagający działanie szablonu. Przykład znajdziesz w pliku *Home.vue* w katalogu *src/views*.

```
<template>
  <div class="home">
    
    <HelloWorld msg="Witaj w aplikacji Your Vue.js + TypeScript"/>
  </div>
</template>

<script lang="ts">
import { defineComponent } from 'vue';
import HelloWorld from '@components/HelloWorld.vue'; // @ to alias dla /src.

export default defineComponent({
  name: 'Home',
  components: {
    HelloWorld,
  },
});
</script>
```

Język, w którym został utworzony kod zdefiniowany w elemencie `<Script>`, jest wymieniony w atrybucie `lang`:

```
...
<script lang="ts">
...
```

Ta wartość wskazuje na użycie TypeScriptu i gwarantuje, że kod zostanie przetworzony przez kompilator TypeScriptu. Komponenty są definiowane za pomocą funkcji `defineComponent()` i wyrażone z użyciem składni literału obiektu JavaScriptu. Ewentualnie do zdefiniowania komponentów można wykorzystać klasy, ale w książce nie skorzystałem z tej możliwości, ponieważ jej przyszłość jest niepewna, a składnia literału jest powszechnie stosowana.

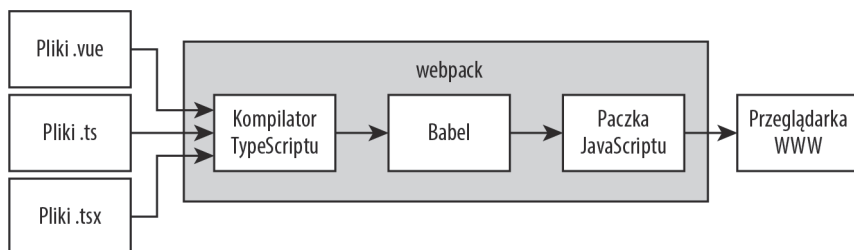
Zestaw narzędzi TypeScriptu podczas programowania z użyciem frameworka Vue.js

Narzędzia programistyczne Vue.js opierają swoje działanie na pakietach webpack i Webpack Development Server, które wykorzystałem w rozdziale 15., a także podczas tworzenia aplikacji z użyciem frameworków Angular i React. Gdy projekt oparty na Vue.js ma korzystać z języka TypeScript, zostanie utworzony plik `tsconfig.json` konfigurujący kompilator TypeScriptu następującymi ustawieniami:

```
{
  "compilerOptions": {
    "target": "esnext",
    "module": "esnext",
    "strict": true,
    "jsx": "preserve",
    "importHelpers": true,
    "moduleResolution": "node",
    "skipLibCheck": true,
    "esModuleInterop": true,
    "allowSyntheticDefaultImports": true,
    "sourceMap": true,
    "baseUrl": ".",
    "types": ["webpack-env"],
    "paths": {
      "@/*": ["src/*"]
    },
    "lib": ["esnext", "dom", "dom.iterable", "scripthost"]
  },
  "include": ["src/**/*.ts", "src/**/*.tsx", "src/**/*.vue",
    "tests/**/*.ts", "tests/**/*.tsx"],
  "exclude": ["node_modules"]
}
```

Narzędzia programistyczne Vue.js współdziałają z plikami `.vue` poprzez konwersję zawartości elementu `<template>` na polecenia kodu oraz wykorzystują kompilator TypeScriptu do przetworzenia zawartości elementu `<script>`. Skompilowany kod jest przekazywany do pakietu Babel, który następnie pozwala na wygenerowanie kodu dla wskazanej wersji JavaScriptu. Zwykłe pliki

TypeScriptu i w formacie JSX również są obsługiwane, a dane wyjściowe trafiają do plików paczek udostępnianych przeglądarce WWW za pomocą serwera Webpack Development Server, jak pokazałem na rysunku 21.2.



Rysunek 21.2. Zestaw narzędzi programistycznych Vue.js

Tworzenie klas encji

W celu zdefiniowania typów danych, którymi będzie zarządzała aplikacja, należy utworzyć katalog `src/data` i dodać do niego plik o nazwie `entities.ts` zawierający kod przedstawiony na listingu 21.9.

Listing 21.9. Zawartość pliku `entities.ts` w katalogu `src/data`

```
export class Product {
  constructor(
    public id: number,
    public name: string,
    public description: string,
    public category: string,
    public price: number) {}
};

export class OrderLine {
  constructor(public product: Product, public quantity: number) {
    // Polecenia nie są wymagane.
  }

  get total(): number {
    return this.product.price * this.quantity;
  }
}

export class Order {
  private lines: OrderLine[] = [];

  constructor(initialLines?: OrderLine[]) {
    if (initialLines) {
      this.lines.push(...initialLines);
    }
  }
}
```

```

public addProduct(prod: Product, quantity: number) {
  let index = this.lines.findIndex(ol => ol.product.id === prod.id)
  if (index > -1) {
    if (quantity === 0) {
      this.removeProduct(prod.id);
    } else {
      this.lines[index].quantity += quantity;
    }
  } else {
    this.lines.push(new OrderLine(prod, quantity));
  }
}

public removeProduct(id: number) {
  this.lines = this.lines.filter(ol => ol.product.id !== id);
}

get orderLines(): OrderLine[] {
  return this.lines;
}

get productCount(): number {
  return this.lines.reduce((total, ol) => total += ol.quantity, 0);
}

get total(): number {
  return this.lines.reduce((total, ol) => total += ol.total, 0);
}
}

```

Te typy opisują produkty i zamówienia oraz zachodzące między nimi relacje.

W przeciwieństwie do aplikacji przedstawionych w pozostałych rozdziałach tej części książki, w omawianej tutaj aplikacji `Product` to klasa, a nie alias typu, ponieważ narzędzia programistyczne Vue.js opierają swoje działanie na konkretnych typach. System wykrywania zmian w Vue.js nie współpracuje zbyt dobrze z klasą `Map` w JavaScriptcie, więc klasa `Order` dla aplikacji Vue.js została utworzona na podstawie tablicy.

Wyświetlanie filtrowanej listy produktów

Vue.js obsługuje wiele różnych sposobów na definiowanie komponentów, które są kluczowymi elementami konstrukcyjnymi podczas wyświetlania treści użytkownikowi. W omawianym przykładzie zamierzam skorzystać z najpopularniejszego rozwiązania, czyli formatu komponentu w postaci pojedynczego pliku zawierającego treść HTML-a i obsługujący ją kod. (Takie pliki mogą zawierać również style CSS, ale nie wykorzystam tej możliwości, ponieważ na listingu 21.6 dodałem do aplikacji pakiet Bootstrap CSS).

Zgodnie z konwencją poszczególne komponenty są przechowywane w katalogu `src/components`, a przed wyświetleniem użytkownikowi są ze sobą łączone za pomocą kodu znajdującego się w katalogu `src/views`. W celu wyświetlenia informacji o pojedynczym produkcie dodaj do katalogu `src/components` plik o nazwie `ProductItem.vue` z kodem przedstawionym na listingu 21.10.

Listing 21.10. Zawartość pliku *ProductItem.vue* w katalogu *src/components*

```
<template>
  <div class="card m-1 p-1 bg-light">
    <h4>
      {{ product.name }}
      <span class="badge badge-pill badge-primary float-right">
        {{ product.price.toFixed(2) }} zł
      </span>
    </h4>
    <div class="card-text bg-white p-1">
      {{ product.description }}
      <button class="btn btn-success btn-sm float-right"
        @click="handleAddToCart">
        Dodaj do koszyka
      </button>
      <select class="form-control-inline float-right m-1"
        v-model.number="quantity">
        <option>1</option>
        <option>2</option>
        <option>3</option>
      </select>
    </div>
  </div>
</template>

<script lang="ts">

import { defineComponent, PropType } from "vue";
import { Product } from "../data/entities";

export default defineComponent({
  name: "ProductItem",
  props: {
    product: {
      type: Object as PropType<Product>
    }
  },
  data() {
    return {
      quantity: 1
    }
  },
  methods: {
    handleAddToCart(){
      this.$emit("addToCart",
        { product: this.product, quantity: this.quantity });
    }
  }
});
</script>
```

Element szablonu komponentu Vue.js wykorzystuje wiązanie danych, wskazywane przez podwójny nawias klamrowy, do wyświetlenia wartości danych. Natomiast atrybuty obsługi zdarzeń, zawierające prefiks @, są stosowane do zapewnienia obsługi zdarzeń. Wyrażenia określone

za pomocą wspomnianych wiązań danych oraz atrybuty zdarzeń są przetwarzane z użyciem funkcjonalności zdefiniowanej przez klasę w elemencie `<script>`.

Komponent na listingu 21.10 wyświetla szczegółowe informacje o obiekcie `Product` i emituje zdarzenie po kliknięciu przez użytkownika przycisku *Dodaj do koszyka*.

Kod komponentu jest zdefiniowany w elemencie `<script>`, sam komponent zaś jest tworzony za pomocą funkcji `defineComponent()` pochodzącej z pakietu `vue`.

```
...
export default defineComponent({
...
```

Właściwości obiektu przekazanego funkcji `defineComponent()` określają różne aspekty sposobu działania komponentu. Właściwość `props` służy do opisanie wartości danych, które komponent będzie otrzymywał od komponentu nadrzędnego, np.:

```
...
props: {
  product: {
    type: Object as PropType<Product>
  }
},
...
```

Ten komponent definiuje pojedynczą właściwość o nazwie `product`. W celu określenia typu oczekiwanej wartości danych właściwości `product` został przypisany obiekt definiujący parametr `type`, którego wartością jest następujące wyrażenie:

```
...
type: Object as PropType<Product>
...
```

Vue.js implementuje proste sprawdzanie typu właściwości. W celu uwzględnienia TypeScriptu to wyrażenie korzysta z typu generycznego `PropType<T>`, w którym oczekiwany typ został podany jako argument. W omawianym przykładzie wyrażenie określa, że oczekiwanym typem właściwości `product` jest `Product`.

Właściwość `data` zostaje przypisana funkcja zwracająca obiekt używany do zdefiniowania stanu danych, których wymaga komponent. Ten komponent definiuje pojedynczą właściwość stanu o nazwie `quantity` i wartości początkowej `1`.

Właściwość `methods` jest używana do zdefiniowania metod komponentu, które mogą być wywołane w odpowiedzi na zdarzenia. Ten komponent definiuje metodę o nazwie `handleAddToCart()`, która używa metody `$emit()` do wywołania zdarzenia niestandardowego. Ta metoda zostanie wywołana, gdy użytkownik w szablonie komponentu kliknie element `<button>`, dla którego istnieje procedura obsługi zdarzeń:

```
...
<button class="btn btn-success btn-sm float-right" @click="handleAddToCart">
...
```

W efekcie kliknięcie przycisku spowoduje, że komponent wygeneruje zdarzenie, które trafi do komponentu nadrzędnego. Dane przekazywane z tym zdarzeniem zawierają obiekt `Product` otrzymany jako właściwość i wartość bieżącą właściwości stanu `quantity`.

Wyświetlanie listy kategorii i nagłówka

W celu wyświetlenia przycisków kategorii dodaj do katalogu `src/components` plik o nazwie `CategoryList.vue` z kodem przedstawionym na listingu 21.11.

Listing 21.11. Zawartość pliku `CategoryList.vue` w katalogu `src/components`

```
<template>
  <div>
    <button v-for="c in categories"
      v-bind:key="c"
      v-bind:class="getButtonClasses(c)"
      @click="selectCategory(c)">
      {{ c }}
    </button>
  </div>
</template>

<script lang="ts">

import { defineComponent, PropType } from "vue";

export default defineComponent({
  name: "CategoryList",
  props: {
    categories: {
      type: Object as PropType<string[]>
    },
    selected: {
      type: String as PropType<string>
    }
  },
  methods: {
    selectCategory(category: string) {
      this.$emit("selectCategory", category);
    }

    getButtonClasses(category: string) : string {
      let btnClass = this.selected === category
        ? "btn-primary": "btn-secondary";
      return `btn btn-block ${btnClass}`;
    }
  }
});
</script>
```

Ten komponent wyświetla listę przycisków i powoduje zaznaczenie tego, który odpowiada aktualnie wybranej kategorii. Atrybuty elementu w sekcji `<template>` są traktowane jako literały ciągu tekstowego, o ile nie zostaną poprzedzone prefiksem `v-bind`, który nakazuje Vue.js utworzenie wiązania danych między kodem elementu `<script>` a wartością przypisywaną atrybutowi. Jest to przykład tzw. *dyrektywy* w Vue.js pozwalającej, aby wynik działania metod zdefiniowanych przez klasę komponentu był wstawiany do kodu HTML-a w sekcji szablonu, np.:


```
...
v-bind:class="getButtonClasses(c)"
...
```

Ten fragment wskazuje Vue.js, że wartością atrybutu `class` powinien być wynik wywołania metody `getButtonClasses()`. Argument metody jest pobierany z innej dyrektywy, `v-for`, powtarzającej element dla każdego obiektu sekwencji.

```
...
<button v-for="c in categories" v-bind:key="c" v-bind:class="getButtonClasses(c)"
  @click="selectCategory(c)">
  {{ c }}
</button>
...
```

Ta dyrektywa `v-for` nakazuje Vue.js utworzenie elementu `<button>` dla każdej wartości w sekwencji zwróconej przez właściwość `categories`. W celu przeprowadzenia efektywnego uaktualnienia Vue.js wymaga przypisania atrybutu `key` poszczególnym elementom, stąd użycie razem dyrektyw `v-for` i `v-bind:key`.

Wynikiem jest seria elementów `<button>` dla każdej kategorii. Kliknięcie przycisku wywołuje metodę `selectCategory()`, która z kolei wywołuje niestandardowe zdarzenia i pozwala, aby komponent sygnalizował innej części aplikacji kategorię wybraną przez użytkownika.

Podczas sprawdzania typu właściwości w Vue.js używane są nazwy funkcji konstruktorów typów wbudowanych JavaScriptu. To oznacza użycie typu `String` w trakcie definiowania właściwości `string`, jak pokazałem w kolejnym fragmencie kodu:

```
...
selected: {
  type: String as PropType<string>
}
...
```

To może się okazać nieco dezorientujące, ale szybko wejdzie Ci w krew, gdy tylko nauczysz się tworzyć komponenty. Aby wskazać, że wartość właściwości `order` jest wymagana, do definicji została dodana właściwość `required`, jak pokazałem w kolejnym fragmencie kodu:

```
...
categories: {
  type: Object as PropType<string[]>,
  required: true
},
...
```

Bez właściwości `required` kompilator TypeScriptu zdefiniowałby typ właściwości `categories` jako `string[] | undefined`, co następnie wymagałoby wyrażenia definiującego właściwość `selected` w celu sprawdzania pod kątem wartości `undefined`, aby uniknąć błędów kompilacji.

W celu utworzenia komponentu wyświetlającego nagłówki dodaj do katalogu `src/components` plik o nazwie `Header.vue` z kodem przedstawionym na listingu 21.12.

Listing 21.12. Zawartość pliku *Header.vue* w katalogu *src/components*

```
<template>

  <div class="p-1 bg-secondary text-white text-right">
    {{ displayText }}
    <button class="btn btn-sm btn-primary m-1">
      Złóż zamówienie
    </button>
  </div>

</template>

<script lang="ts">

import { defineComponent, PropType } from "vue";
import { Order } from "../data/entities";

export default defineComponent({
  name: "Header",
  props: {
    order: {
      type: Object as PropType<Order>,
      required: true
    }
  },
  computed: {
    displayText(): string {
      const count = this.order.productCount;
      return count === 0 ? "(brak produktów)"
        : `Liczba produktów: ${ count }, ${ this.order.total.toFixed(2)} zł`
    }
  }
})
</script>
```

Komponent Header wyświetla podsumowanie bieżącego zamówienia. Właściwość `computed` jest używana do zdefiniowania funkcji, której wynik działania jest pochodną danych komponentu, w tym także jego właściwości. To umożliwia Vue.js buforowanie wartości wygenerowanych przez te funkcje oraz wywołanie funkcji jedynie w przypadku zmiany danych komponentu. W omawianym przykładzie komponent Header definiuje funkcję o nazwie `displayText()`, której wynik działania będzie zależny od jej właściwości `order`.

Tworzenie i testowanie komponentów

Aby utworzyć komponent wyświetlający nagłówek, listę produktów i przyciski kategorii, dodaj do katalogu *src/views* plik o nazwie *ProductList.vue* i umieść w nim kod przedstawiony na listingu 21.13. Położenie tego pliku wskazuje na to, że będzie on wyświetlał widok utworzony na podstawie innych komponentów. Wprawdzie jest to często spotykana konwencja, ale nie musisz się do niej stosować we własnych projektach.

Listing 21.13. Zawartość pliku *ProductList.vue* w katalogu *src/views*

```

<template>
  <div>
    <Header v-bind:order="order" />
    <div class="container-fluid">
      <div class="row">
        <div class="col-3 p-2">
          <CategoryList v-bind:categories="categories"
            v-bind:selected="selectedCategory"
            @selectCategory="handleSelectCategory" />
        </div>
        <div class="col-9 p-2">
          <ProductItem v-for="p in filteredProducts" v-bind:key="p.id"
            v-bind:product="p" @addToCart="handleAddToCart" />
        </div>
      </div>
    </div>
  </div>
</template>

<script lang="ts">

import { defineComponent } from "vue";
import { Product, Order } from "../data/entities";
import ProductItem from "../components/ProductItem.vue";
import CategoryList from "../components/CategoryList.vue";
import Header from "../components/Header.vue";

export default defineComponent({
  name: "ProductList",
  components: { ProductItem, CategoryList, Header },
  data() {
    const products: Product[] = [];
    [1, 2, 3, 4, 5].map(num =>
      products.push(new Product(num, `Prod${num}`, `Produkt ${num}`,
        `Kat${num % 2}`, 100)));
    return {
      products,
      selectedCategory: "Wszystkie",
      order: new Order()
    }
  },
  computed: {
    categories(): string[] {
      return ["Wszystkie", ...new Set<string>(this.products.map(p => p.category))];
    },

    filteredProducts(): Product[] {
      return this.products.filter(p =>
        this.selectedCategory == "Wszystkie"
        || this.selectedCategory === p.category);
    },
  },
  methods: {

```

```

        handleSelectCategory(category: string) {
            this.selectedCategory = category;
        },

        handleAddToCart(data: {product: Product, quantity: number}) {
            this.order.addProduct(data.product, data.quantity);
        }
    })
</script>

```

Komponent `ProductList` łączy komponenty `ProductItem`, `CategoryList` i `Header` w celu wyświetlenia odpowiedniej treści użytkownikowi. Wykorzystanie innych komponentów to proces wieloetapowy. Przede wszystkim komponent musi być zaimportowany za pomocą polecenia `import`.

```

...
import Header from "../components/Header.vue";
...

```

Zwróć uwagę na brak użycia nawiasu klamrowego w poleceniu `import`, a także na podanie rozszerzenia w nazwie pliku. Obiekt przekazany funkcji `defineComponent()` używa właściwości `components` do określenia wymaganych komponentów.

```

...
components: { ProductItem, CategoryList, Header },
...

```

Ostatnim krokiem jest dodanie elementów do sekcji `<template>` w pliku, co powoduje zastosowanie komponentów i dostarczenie wartości dla właściwości.

```

...
<Header v-bind:order="order" />
...

```

Element `<Header>` ma zastosowanie dla komponentu `Header`. `Vue.js` używa dyrektywy `v-bind` w celu zapewnienia wiązania danych, przypisującej właściwości `order` komponentu `Header` wartość właściwości `order` zdefiniowanej przez klasę `ProductList`. W ten sposób jeden komponent dostarcza wartości danych drugiemu.

Aby mieć pewność, że komponent będzie mógł wyświetlić treść użytkownikowi, należy zastąpić zawartość pliku `App.vue` kodem przedstawionym na listingu 21.14.

Listing 21.14. Nowa zawartość pliku `App.vue` w katalogu `src`

```

<template>
  <ProductList />
</template>

<script lang="ts">

import { defineComponent } from "vue";
import ProductList from "../views/ProductList.vue";

export default defineComponent({
  name: "App",

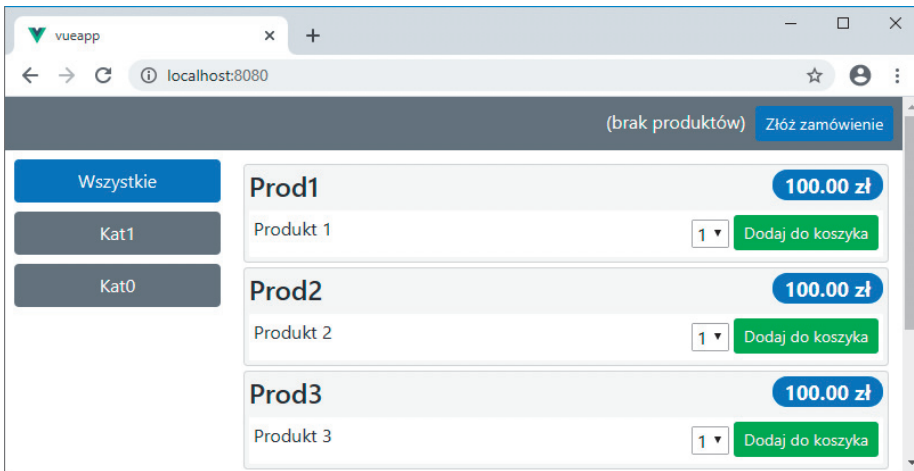
```

```

    components: { ProductList }
  });
</script>

```

Komponent App został uaktualniony do wyświetlania komponentu ProductList, zastępując tym samym miejsce zarezerwowane zdefiniowane w chwili konfigurowania projektu. Po zapisaniu zmian w komponencie App przeglądarka WWW będzie miała dane wystarczające do wyświetlenia listy produktów, jak pokazałem na rysunku 21.3. Wprawdzie usługa sieciowa dostarczająca rzeczywiste dane zostanie zaimplementowana nieco później, ale obecnie stosowane dane testowe są w zupełności wystarczające do przetestowania podstawowej funkcjonalności aplikacji.



Rysunek 21.3. Testowanie komponentu wyświetlającego listę produktów

Tworzenie magazynu danych

W większości projektów Vue.js danymi aplikacji zarządza pakiet Vuex, który dostarcza funkcjonalność magazynu danych zintegrowaną z API Vue.js. Odpowiedzi udzielone podczas konfigurowania projektu spowodowały dodanie do niego pakietu Vuex i zdefiniowanie miejsca zarezerwowanego dla magazynu danych, o czym możesz się przekonać, analizując zawartość pliku *index.ts* znajdującego się w katalogu *src/store*.

```

import { createStore } from 'vuex'

export default createStore({
  state: {
  },
  mutations: {
  },
  actions: {
  },
  modules: {
  }
})

```

Magazyn danych Vuex jest skonfigurowany z czterema właściwościami: `state`, `mutations`, `actions` i `modules`. Właściwość `state` jest przeznaczona do zdefiniowania stanu zarządzania danymi przez magazyn danych. Właściwość `mutations` jest używana do zdefiniowania funkcji modyfikujących dane stanu. Z kolei właściwość `actions` pozwala na zdefiniowanie zadań asynchronicznych, które będą uaktualniały magazyn danych. Natomiast właściwość `modules` służy do zarządzania skomplikowanymi magazynami danych, które są definiowane w wielu plikach. W omawianej aplikacji nie będę korzystał z tej funkcjonalności.

Magazyn danych może również definiować właściwość `getters` używaną do obliczania wartości na podstawie danych znajdujących się w magazynie danych. W kodzie na listingu 21.15 przedstawiłem dodanie podstawowych danych stanu, funkcji i metod dla przykładowej aplikacji z użyciem danych testowych, co pozwoli na rozpoczęcie pracy z magazynem danych.

Listing 21.15. Konfigurowanie magazynu danych w pliku `index.ts` w katalogu `src/store`

```
import { createStore, Store } from "vuex";
import { Product, Order } from '../data/entities';

export interface StoreState {
  products: Product[],
  order: Order,
  selectedCategory: string
}

type ProductSelection = {
  product: Product,
  quantity: number
}

export default createStore<StoreState>({
  state: {
    products: [1, 2, 3, 4, 5].map(num => new Product(num, `Superprod${num}`,
      `Produkt ${num}`, `Kat${num % 2}`, 450)),
    order: new Order(),
    selectedCategory: "Wszystkie"
  },
  mutations: {
    selectCategory(currentState: StoreState, category: string) {
      currentState.selectedCategory = category;
    },

    addToOrder(currentState: StoreState, selection: ProductSelection) {
      currentState.order.addProduct(selection.product, selection.quantity);
    }
  },
  getters: {
    categories(state): string[] {
      return ["Wszystkie", ...new Set(state.products.map(p => p.category))];
    },

    filteredProducts(state): Product[] {
      return state.products.filter(p => state.selectedCategory === "Wszystkie"
        || state.selectedCategory === p.category);
    }
  }
});
```

```

    },
    actions: {
    },
    modules: {
    }
  })

```

Projekt został skonfigurowany z plikami deklaracji dla Vuex, co pozwala na utworzenie magazynu danych z argumentem typu generycznego opisującego typy danych stanu, które później TypeScript wykorzysta w trakcie operacji sprawdzania typów. Na omawianym listingu został zdefiniowany interfejs `StoreState` opisujący wartości typów `product`, `order` i `selectedCategory`, którymi będzie zarządzać magazyn danych. Interfejs wykorzystam jako argument typu podczas tworzenia magazynu danych.

```

...
export default createStore<StoreState>({
...

```

Łączenie komponentów z magazynem danych

Połączenie komponentów z magazynem danych odbywa się za pomocą funkcji pomocniczych, które funkcjonalność magazynu integrują z funkcjonalnością dostarczaną przez komponent. Na listingu 21.16 pokazałem, jak połączyć komponent `Header` z magazynem danych.

Listing 21.16. Połączenie z magazynem danych zdefiniowane w pliku `Header.vue` w katalogu `src/components`

```

<template>

  <div class="p-1 bg-secondary text-white text-right">
    {{ displayText }}
    <button class="btn btn-sm btn-primary m-1">
      Złóż zamówienie
    </button>
  </div>

</template>

<script lang="ts">

import { defineComponent, PropType } from "vue";
import { Order } from "../data/entities";
import { useStore } from "vuex";

export default defineComponent({
  name: "Header",
  setup() {
    return { store: useStore() }
  },
  // props: {
  //   order: {
  //     type: Object as PropType<Order>,

```

```
//      required: true
//    }
// },
computed: {
  displayText(): string {
    const count = this.store.state.order.productCount;
    return count === 0 ? "(Brak produktów)"
      : `Liczba produktów ${ count }, `
        + `${ this.store.state.order.total.toFixed(2)} zł`;
  }
})
</script>
```

Funkcja `setup()` zdefiniowana przez obiekt przekazany funkcji `defineComponent()` jest używana do przeprowadzenia konfiguracji początkowej wymaganej przez komponent. Funkcja zdefiniowana przez komponent `Header` wywołuje funkcję `useStore()` w celu uzyskania dostępu do magazynu danych i udostępnienia go pozostałej części komponentu za pomocą zwróconego obiektu razem z właściwością `store`.

```
...
return { store: useStore() }
...
```

Zamiast używać właściwości funkcja `displayText()` wykorzystuje wartość stanu `Order` w magazynie, która zapewnia dostęp za pomocą właściwości `store` zdefiniowanej przez funkcję `setup()`.

```
...
const count = this.store.state.order.productCount;
...
```

Właściwości danych stanu zdefiniowane przez magazyn danych są dostępne za pomocą składni `store.state`. Na listingu 21.17 pokazałem przykład połączenia komponentu `ProductList` z magazynem danych.

Listing 21.17. Połączenie z magazynem danych w kodzie pliku `ProductList.vue` w katalogu `src/views`

```
<template>
  <div>
    <Header />
    <div class="container-fluid">
      <div class="row">
        <div class="col-3 p-2">
          <CategoryList v-bind:categories="categories"
            v-bind:selected="selectedCategory"
            @selectCategory="handleSelectCategory" />
        </div>
        <div class="col-9 p-2">
          <ProductItem v-for="p in filteredProducts" v-bind:key="p.id"
            v-bind:product="p" @addToCart="handleAddToCart" />
        </div>
      </div>
    </div>
  </div>
```



```

</template>

<script lang="ts">

import { defineComponent } from "vue";
import { Product, Order } from "../data/entities";
import ProductItem from "../components/ProductItem.vue";
import CategoryList from "../components/CategoryList.vue";
import Header from "../components/Header.vue";
import { mapMutations, mapState, mapGetters } from "vuex";
import { StoreState } from "../store";

export default defineComponent({
  name: "ProductList",
  components: { ProductItem, CategoryList, Header },
  // data() {
  //   const products: Product[] = [];
  //   [1, 2, 3, 4, 5].map(num =>
  //     products.push(new Product(num, `Prod${num}`, `Produkt ${num}`,
  //       `Kat${num % 2}`, 100)));
  //   return {
  //     products,
  //     selectedCategory: "Wszystkie",
  //     order: new Order()
  //   }
  // },
  computed: {
    ...mapState<StoreState>({
      selectedCategory: (state: StoreState) => state.selectedCategory,
      products: (state: StoreState) => state.products,
      order: (state: StoreState) => state.order
    }),
    ...mapGetters(["filteredProducts", "categories"])
  },
  methods: {
    ...mapMutations({
      handleSelectCategory: "selectCategory",
      handleAddToCart: "addToOrder"
    }),
  }
})
</script>

```

W komponencie Header został pobrany obiekt magazynu danych i następnie użyty w celu uzyskania dostępu do wartości danych stanu. Jednak takie rozwiązanie może być uciążliwe, gdy zachodzi potrzeba wykorzystania funkcjonalności wielu magazynów danych. Vuex oferuje funkcje mapujące funkcjonalność magazynu danych na komponent. Funkcje `mapState()` i `mapGetters()` udostępniają dane stanu i metody getter jako właściwości obliczane, a funkcja `mapMutations()` udostępnia mutacje jako metody. Te funkcje są używane razem z operatorem rozwinęcia, jak pokazałem w kolejnym fragmencie kodu:

```
...
...mapGetters(["filteredProducts", "categories"])
...
```

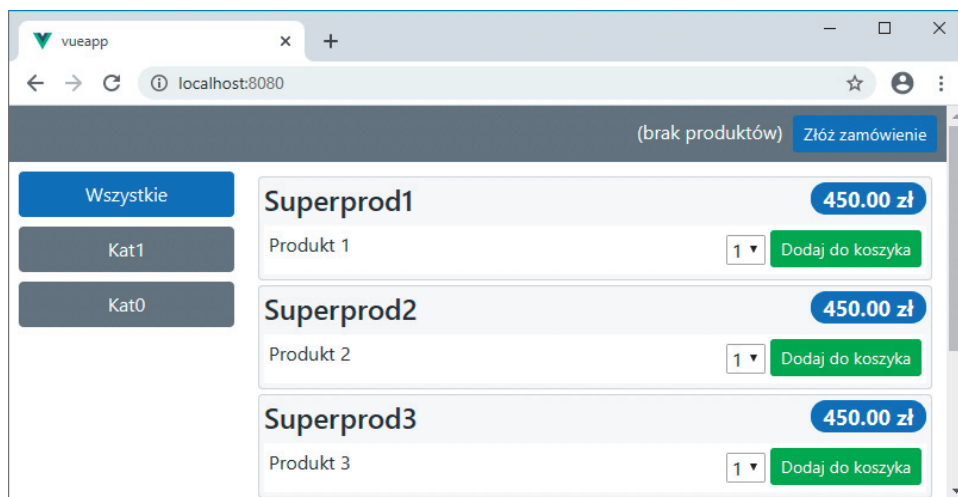
Operator rozwinęcia gwarantuje, że właściwości wygenerowane przez funkcje mapowania będą wykorzystane w funkcjonalności komponentu.

W szablonie komponentu element `<header>` został uaktualniony, ponieważ komponent nie otrzymuje już obiektu `Order` i jest teraz połączony z magazynem danych użytkownika.

Po zapisaniu zmian magazyn danych będzie wyświetlał dane testowe, jak pokazałem na rysunku 21.4.

Dodawanie obsługi usługi sieciowej

Aby przygotować magazyn danych do pracy z usługą sieciową, konieczne jest dodanie akcji przedstawionych na listingu 21.18. Akcja to po prostu operacja asynchroniczna, która może stosować pewne działania i modyfikować magazyn danych.



Rysunek 21.4. Przykładowa aplikacja wykorzystująca magazyn danych

Listing 21.18. Dodawanie akcji do pliku `index.ts` w katalogu `src/store`

```
import { createStore, Store } from "vuex";
import { Product, Order } from '../data/entities';

export interface StoreState {
  products: Product[],
  order: Order,
  selectedCategory: string,
  storedId: number
}

type ProductSelection = {
  product: Product,
```

```

    quantity: number
  }
}

export default createStore<StoreState>({
  state: {
    products: [],
    order: new Order(),
    selectedCategory: "Wszystkie",
    storedId: -1
  },
  mutations: {
    selectCategory(currentState: StoreState, category: string) {
      currentState.selectedCategory = category;
    },

    addToOrder(currentState: StoreState, selection: ProductSelection) {
      currentState.order.addProduct(selection.product, selection.quantity);
    },

    addProducts(currentState: StoreState, products: Product[]) {
      currentState.products = products;
    },

    setOrderId(currentState: StoreState, id: number) {
      currentState.storedId = id;
    },

    resetOrder(currentState: StoreState) {
      currentState.order = new Order();
    }
  },
  getters: {
    categories(state): string[] {
      return ["Wszystkie", ...new Set(state.products.map(p => p.category))];
    },

    filteredProducts(state): Product[] {
      return state.products.filter(p => state.selectedCategory === "Wszystkie"
        || state.selectedCategory === p.category);
    }
  },
  actions: {
    async loadProducts(context, task: () => Promise<Product[]>) {
      let data = await task();
      context.commit("addProducts", data);
    },

    async storeOrder(context, task: (order: Order) => Promise<number>) {
      context.commit("setOrderId", await task(context.state.order));
      context.commit("resetOrder");
    }
  },
  modules: {
  }
})

```

Akcje mają możliwość modyfikowania magazynu danych tylko za pomocą specjalnych funkcji. Zmiany wprowadzone w kodzie przedstawionym na listingu 21.18 definiują akcje pozwalające na wczytywanie produktów i dodawanie ich do magazynu danych, a także na przekazywanie zamówień do serwera.

Vue.js nie oferuje zintegrowanej obsługi żądań HTTP. Popularnym pakietem pozwalającym na pracę z żądaniami HTTP jest Axios, z którego korzystałem w innych aplikacjach zbudowanych w tej części książki (do tego projektu został dodany na listingu 21.3). W celu zdefiniowania operacji HTTP wymaganych przez tę aplikację należy dodać do katalogu *src/data* plik o nazwie *httpHandler.ts* zawierający kod przedstawiony na listingu 21.19.

Listing 21.19. Zawartość pliku *httpHandler.ts* w katalogu *src/data*

```
import Axios from "axios";
import { Product, Order } from "../entities";

const protocol = "http";
const hostname = "localhost";
const port = 4600;

const urls = {
  products: `${protocol}://${hostname}:${port}/products`,
  orders: `${protocol}://${hostname}:${port}/orders`
};

export class HttpHandler {
  loadProducts(): Promise<Product[]> {
    return Axios.get<Product[]>(urls.products).then(response => response.data);
  }

  storeOrder(order: Order): Promise<number> {
    let orderData = {
      lines: [...order.orderLines.values()].map(ol => ({
        productId: ol.product.id,
        productName: ol.product.name,
        quantity: ol.quantity
      })))
    }
    return Axios.post<{id : number}>(urls.orders, orderData)
      .then(response => response.data.id);
  }
}
```

Na listingu 21.20 pokazałem zmiany konieczne do wprowadzenia w komponencie App, aby można było wczytywać dane produktów z usługi sieciowej.

Listing 21.20. Używanie usługi sieciowej w kodzie pliku *App.vue* w katalogu *src*

```
<template>
  <ProductList />
</template>

<script lang="ts">

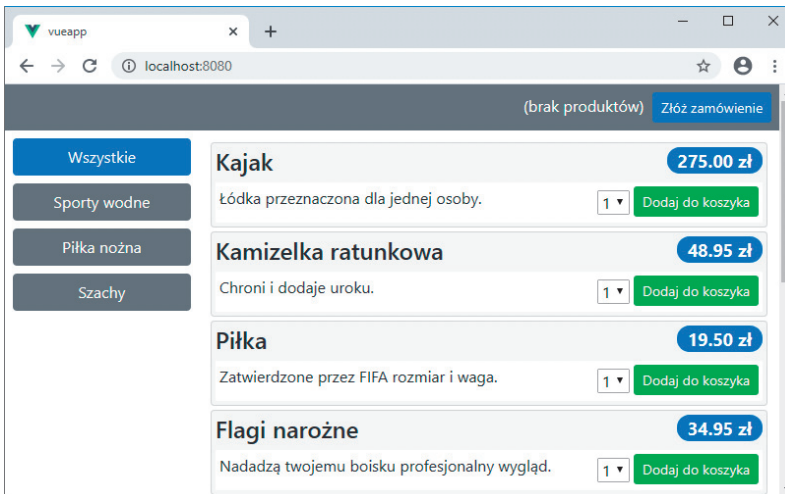
import { defineComponent, onMounted } from "vue";
```

```
import ProductList from "../views/ProductList.vue";
import { HttpHandler } from "../data/httpHandler";
import { useStore } from "vuex";

export default defineComponent({
  name: "App",
  components: { ProductList },

  setup() {
    const store = useStore();
    const handler = new HttpHandler();
    onMounted(() => store.dispatch("loadProducts", handler.loadProducts));
  }
});
</script>
```

Funkcja `onMounted()` to jedna z dostarczanych przez Vue.js metod cyklu życiowego komponentu. Akceptuje funkcję przeznaczoną do wywołania po zamontowaniu komponentu, co następuje po wygenerowaniu zawartości komponentu po raz pierwszy — to jest typowy moment, w którym następuje wczytywanie danych zewnętrznych. Wynikiem jest pobranie rzeczywistych danych produktów z magazynu danych, jak pokazałem na rysunku 21.5.



Rysunek 21.5. Przykładowa aplikacja korzystająca z usługi sieciowej

Podsumowanie

W tym rozdziale pokazałem, jak utworzyć opartą na frameworku Vue.js aplikację internetową wykorzystującą TypeScript. Pakiet umożliwiający utworzenie projektu zapewnia zintegrowaną obsługę TypeScriptu, co pozwala przygotować podstawową strukturę aplikacji. Wyjaśniłem również, jak połączyć komponenty z funkcjonalnością magazynu danych Vuex oraz jak odbywa się wczytywanie danych z usługi sieciowej. W następnym rozdziale zajmę się dokończeniem pracy nad aplikacją i przygotowaniem jej do wdrożenia.

ROZDZIAŁ 22.



Tworzenie aplikacji internetowej Vue.js — część II

W rozdziale dokończę proces tworzenia opartej na frameworku Vue.js aplikacji przez dodanie obsługi routingu URL oraz pozostałych komponentów. Na końcu pokażę, jak przygotować aplikację do wdrożenia w kontenerze. W tabeli 22.1 wymienię opcje kompilatora TypeScriptu użyte w rozdziale.

Tabela 22.1. Opcje kompilatora TypeScriptu użyte w rozdziale

Opcja	Opis
<code>allowSyntheticDefaultImports</code>	Ta opcja powoduje importowanie modułów, które nie deklarują wyraźnie eksportowanych funkcji. Jest używana w celu zwiększenia zgodności kodu źródłowego
<code>baseUrl</code>	Ta opcja określa katalog główny używany podczas rozwiązywania zależności modułów
<code>esModuleInterop</code>	Ta opcja dodaje kod pomocniczy pozwalający na importowanie funkcjonalności z modułów niedefiniujących domyślnie eksportowanych funkcji i jest używana z opcją <code>allowSyntheticDefaultImports</code>
<code>importHelpers</code>	Ta opcja określa, czy kod pomocniczy zostanie dodany do JavaScriptu w celu zmniejszenia ogólnej ilości wygenerowanego kodu
<code>jsx</code>	Ta opcja określa sposób przetwarzania elementów HTML-a w plikach TSX
<code>lib</code>	Ta opcja pozwala na wybór używanych przez kompilator plików deklaracji typu
<code>module</code>	Ta opcja określa format używany dla modułów
<code>moduleResolution</code>	Ta opcja określa styl używany podczas rozwiązywania zależności modułów
<code>paths</code>	Ta opcja określa katalogi używane podczas rozwiązywania zależności modułu
<code>skipLibCheck</code>	Ta opcja przyspiesza kompilację przez pominięcie standardowo stosowanej operacji sprawdzania plików deklaracji

Tabela 22.1. Opcje kompilatora TypeScriptu użyte w rozdziale (ciąg dalszy)

Opcja	Opis
sourceMap	Ta opcja określa, czy kompilator ma generować pliki map źródełowych stosowane podczas debugowania
strict	Ta opcja włącza ściślejsze sprawdzanie kodu TypeScriptu
target	Ta opcja określa docelową wersję języka JavaScript, dla której kompilator wygeneruje kod
types	Ta opcja określa listę plików deklaracji dołączanych w trakcie procesu kompilacji

Przygotowanie projektu

W rozdziale będę kontynuował pracę nad projektem utworzonym w poprzednim rozdziale. Otwórz nowe okno powłoki, przejdź do katalogu *vueapp*, a następnie wydaj polecenie przedstawione na listingu 22.1, aby w ten sposób uruchomić usługę sieciową i narzędzia programistyczne Vue.js.

■ **Wskazówka** Przykładowy projekt dla tego rozdziału, a także projekty dla pozostałych rozdziałów książki zostały umieszczone w archiwum dostępnym pod adresem <https://ftp.helion.pl/przyklady/tys4o2.zip>.

Listing 22.1. Uruchamianie narzędzi programistycznych

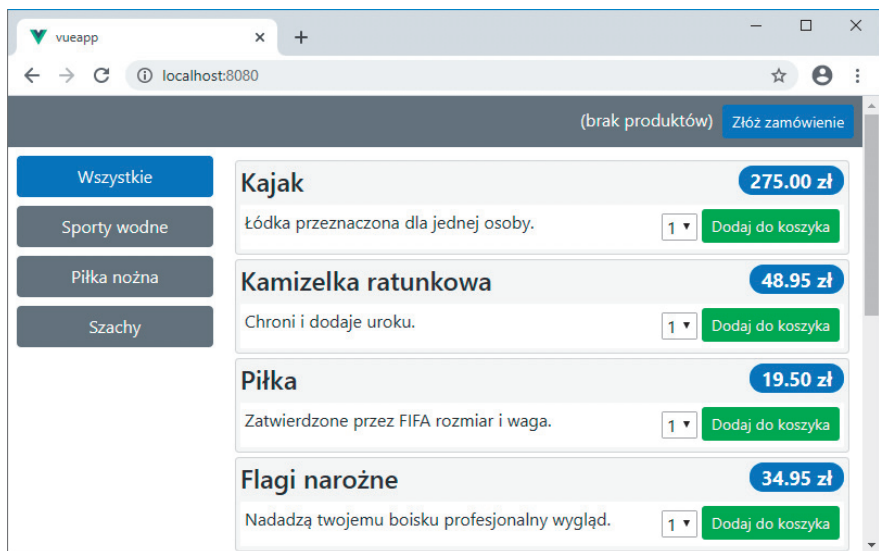
```
$ npm start
```

Po zakończeniu kompilacji początkowej otwórz nowe okno przeglądarki WWW i przejdź pod adres <http://localhost:8080>, a zobaczysz uruchomioną aplikację, jak pokazałem na rysunku 22.1.

Konfigurowanie routingu URL

Większość rzeczywistych projektów Vue.js wykorzystuje routing URL, czyli używa aktualnego adresu URL w przeglądarce WWW do wyboru komponentu wyświetlanego użytkownikowi. Odpowiedzi udzielone na pytania podczas tworzenia projektu spowodowały dodanie pakietu Vue Router i skonfigurowanie go do użycia (za pomocą pliku *index.ts* w katalogu *src/router*). W tabeli 22.2 wymieniałem obsługiwane przez przykładową aplikację adresy URL i ich przeznaczenie.

Nie wszystkie komponenty wymagane przez aplikację zostały już utworzone, więc na listingu 22.2 zaprezentowałem zmiany niezbędne do skonfigurowania adresów URL */products* i */*, a pozostałe będą dodawane w kolejnych sekcjach.



Rysunek 22.1. Przykładowa aplikacja uruchomiona w przeglądarce WWW

Tabela 22.2. Adresy URL obsługiwane przez aplikację

Adres URL	Opis
/products	Ten adres URL powoduje wyświetlenie komponentu ProductList zdefiniowanego w rozdziale 21.
/order	Ten adres URL powoduje wyświetlenie komponentu odpowiedzialnego za dostarczenie informacji o zamówieniu
/summary	Ten adres URL powoduje wyświetlenie podsumowania zamówienia po jego przekazaniu do serwera
/	Domyślny adres URL powoduje przekierowanie pod adres /products, aby został wyświetlony komponent ProductList

Listing 22.2. Konfigurowanie routingu w kodzie pliku index.ts w katalogu src/router

```
import { createRouter, createWebHistory, RouteRecordRaw } from 'vue-router'
import Home from '../views/Home.vue'
import ProductList from "../views/ProductList.vue";

const routes: Array<RouteRecordRaw> = [
  { path: "/products", component: ProductList},
  { path: "/", redirect: "/products"}
]

const router = createRouter({
  history: createWebHistory(process.env.BASE_URL),
  routes
})

export default router
```


Konfiguracja routingu powoduje zdefiniowanie adresu URL `/products` do wyświetlania komponentu `ProductList` i przekierowania z adresu URL `/` na `/products`. W celu wyświetlenia komponentu wybranego przez system routingu konieczne jest wprowadzenie w komponencie `App` zmian przedstawionych na listingu 22.3.

Listing 22.3. Wymagane przez system routingu zmiany w komponencie zdefiniowanym w pliku `App.vue` w katalogu `src`

```
<template>
  <router-view/>
</template>

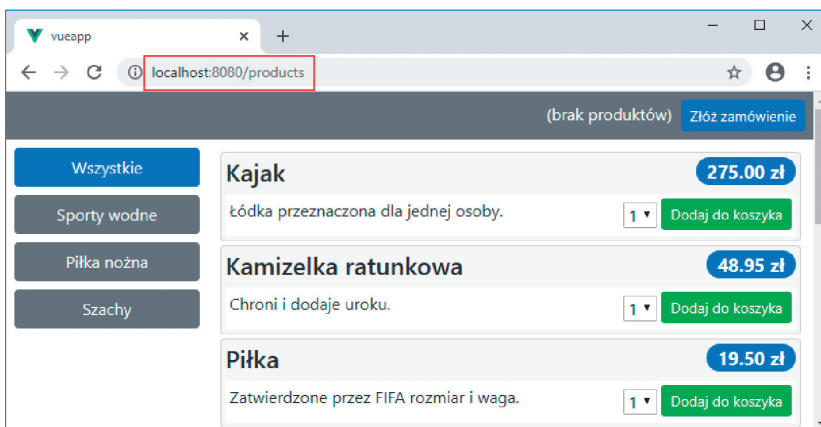
<script lang="ts">

import { defineComponent, onMounted } from "vue";
import ProductList from "../views/ProductList.vue";
import { HttpHandler } from "../data/httpHandler";
import { useStore } from "vuex";

export default defineComponent({
  name: "App",
  //components: { ProductList },
  setup() {
    const store = useStore();
    const handler = new HttpHandler();
    onMounted(() => store.dispatch("loadProducts", handler.loadProducts));
  }
});

</script>
```

Element `<router-view>` wyświetla wybrany komponent. Po zapisaniu zmian przedstawionych na listingu 22.3 aplikacja zostanie ponownie skompilowana, a przeglądarka WWW będzie przekierowana pod adres URL `/products` i wyświetli komponent `ProductList`, jak pokazałem na rysunku 22.2.



Rysunek 22.2. Używanie routingu URL w przykładowej aplikacji

Dokończenie pracy nad funkcjonalnością aplikacji

Skoro aplikacja może wyświetlać komponenty na podstawie bieżącego adresu URL, do projektu można dodać pozostałe komponenty. W celu umożliwienia poruszania się po adresach URL z poziomu przycisku wyświetlanego przez komponent Header dodaj do pliku *Header.vue* polecenia przedstawione na listingu 22.4.

Listing 22.4. Zapewnienie możliwości poruszania się za pomocą adresów URL poprzez modyfikację kodu pliku *Header.vue* w katalogu *src/components*

```
<template>

  <div class="p-1 bg-secondary text-white text-right">
    {{ displayText }}
    <router-link to="/order" class="btn btn-sm btn-primary m-1">
      Złóż zamówienie
    </router-link>
  </div>

</template>

<script lang="ts">

import { defineComponent, PropType } from "vue";
import { Order } from "../data/entities";
import { useStore } from "vuex";

export default defineComponent({
  name: "Header",
  setup() {
    return { store: useStore() }
  },
  computed: {
    displayText(): string {
      const count = this.store.state.order.productCount;
      return count === 0 ? "(brak produktów)"
        : `Liczba produktów ${count}, `
          + `${this.store.state.order.total.toFixed(2)} zł`;
    }
  }
})

</script>
```

Komponent `<router-link>` powoduje wygenerowanie elementu HTML-a, po którego kliknięciu następuje przejście pod podany adres URL. Klasa Bootstrap CSS zastosowana dla tego elementu nadaje mu wygląd przycisku.

Dodawanie komponentu obsługującego podsumowanie zamówienia

W celu wyświetlenia szczegółowych informacji o zamówieniu należy dodać do katalogu *src/views* plik o nazwie *OrderDetails.vue* z kodem przedstawionym na listingu 22.5.

Listing 22.5. Zawartość pliku *OrderDetails.vue* w katalogu *src/views*

```

<template>
  <div>
    <h3 class="text-center bg-primary text-white p-2">Informacje o zamówieniu</h3>
    <div class="p-3">
      <table class="table table-sm table-striped">
        <thead>
          <tr>
            <th>Ilość</th><th>Produkt</th>
            <th class="text-right">Cena</th>
            <th class="text-right">Wartość</th>
          </tr>
        </thead>
        <tbody>
          <tr v-for="line in order.lines" v-bind:key="line.product.id">
            <td>{{ line.quantity }}</td>
            <td>{{ line.product.name }}</td>
            <td class="text-right">
              {{ line.product.price.toFixed(2) }} zł
            </td>
            <td class="text-right">
              {{ line.total.toFixed(2) }} zł
            </td>
          </tr>
        </tbody>
        <tfoot>
          <tr>
            <th class="text-right" colSpan="3">Razem:</th>
            <th class="text-right">
              {{ order.total.toFixed(2) }} zł
            </th>
          </tr>
        </tfoot>
      </table>
    </div>
    <div class="text-center">
      <router-link to="/products" class="btn btn-secondary m-1">
        Wróć
      </router-link>
      <button class="btn btn-primary m-1" @click="submit">
        Złóż zamówienie
      </button>
    </div>
  </div>
</template>

<script lang="ts">

import { defineComponent, } from "vue";
import { Order } from "../data/entities";
import { HttpHandler } from '../data/httpHandler';
import { mapState, mapActions } from "vuex";
import { StoreState } from "../store";

```

```
export default defineComponent({
  name: "OrderDetails",
  computed: {
    ...mapState<StoreState>({
      order: (state: StoreState) => state.order
    })
  },
  methods: {
    ...mapActions(["storeOrder"]),
    submit() {
      this.storeOrder((order: Order) => {
        return new HttpHandler().storeOrder(order).then(id => {
          this.$router.push("/summary");
          return id;
        });
      });
    }
  }
})
</script>
```

Komponent `OrderDetails` wykorzystuje funkcję `Vuex mapActions()` do utworzenia metody wywołującej akcję `storeOrder`. Definiuje również metodę `submit()` odpowiedzialną za wywołanie mapowanej metody `storeOrder()`, która z kolei wykorzystuje klasę `HttpHandler` w celu przekazania egzemplarza `Order` usłudze sieciowej i przekierowania użytkownika pod adres URL `/summary`.

Dodawanie komponentu potwierdzającego złożenie zamówienia

W celu wyświetlenia użytkownikowi komunikatu po złożeniu zamówienia za pomocą usługi sieciowej należy dodać do katalogu `src/views` plik o nazwie `Summary.vue` z kodem przedstawionym na listingu 22.6.

Listing 22.6. Zawartość pliku `Summary.vue` w katalogu `src/views`

```
<template>

<div class="m-2 text-center">
  <h2>Dziękujemy!</h2>
  <p>Dziękujemy za złożenie zamówienia.</p>
  <p>Numer zamówienia #{{ id }}</p>
  <p>Zamówione produkty zostaną wkrótce wysłane.</p>
  <router-link to="/products" class="btn btn-primary">OK</router-link>
</div>
</template>

<script lang="ts">

import { defineComponent } from "vue";
import { mapState } from "vuex";
import { StoreState } from "../store";

export default defineComponent({
  name: "Summary",
```

```

    computed: {
      ...mapState<StoreState>({
        id: (state: StoreState) => state.storedId
      })
    }
  })
</script>

```

Komponent `Summary` musi otrzymać jedynie przypisany przez usługę sieciową numer zamówienia, który jest pobierany z magazynu danych. Element `<router-link>` pozwala użytkownikowi na powrót do adresu URL `/products`.

Dokończenie konfiguracji routingu

Ostatnim krokiem jest dokończenie konfiguracji routingu przez dodanie mapowania między adresami URL obsługiwanymi przez aplikację i odpowiadającymi im komponentami, jak pokazałem na listingu 22.7.

Listing 22.7. Dokończenie konfiguracji routingu w pliku `index.ts` w katalogu `src/router`

```

import { createRouter, createWebHistory, RouteRecordRaw } from 'vue-router'
import Home from '../views/Home.vue'
import ProductList from "../views/ProductList.vue";
import OrderDetails from "../views/OrderDetails.vue";
import Summary from "../views/Summary.vue";

const routes: Array<RouteRecordRaw> = [
  { path: '/products', component: ProductList },
  { path: "/order", component: OrderDetails },
  { path: "/summary", component: Summary },
  { path: "/", redirect: "/products" }
]

const router = createRouter({
  history: createWebHistory(process.env.BASE_URL),
  routes
})

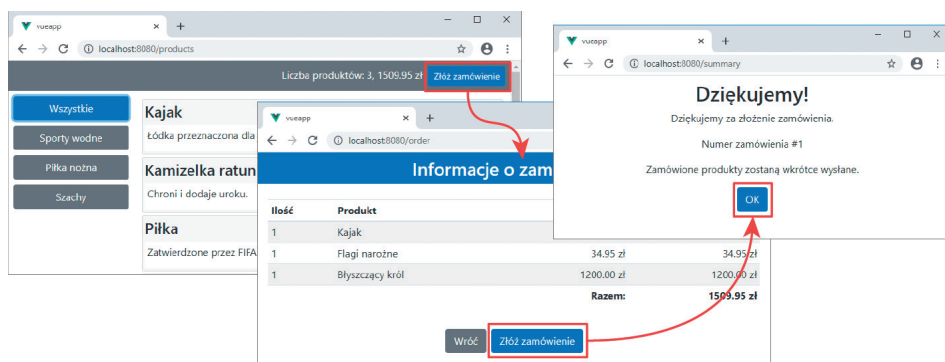
export default router

```

Po zapisaniu zmian produkty mogą być dodawane do zamówienia, które następnie można przekazać do usługi sieciowej, jak pokazałem na rysunku 22.3.

Wdrażanie aplikacji

Narzędzia programistyczne Vue.js opierają swoje działanie na serwerze Webpack Development Server, który nie jest odpowiedni do stosowania w środowisku produkcyjnym, ponieważ dodaje funkcje takie jak automatyczne odświeżanie strony po wygenerowaniu paczki JavaScriptu. W tym podrozdziale przedstawię proces przygotowania aplikacji Vue.js do wdrożenia — jest on podobny do zastosowanego podczas wdrażania pozostałych aplikacji internetowych, także tych utworzonych za pomocą innych frameworków.



Rysunek 22.3. Dokończenie pracy nad funkcjonalnością przykładowej aplikacji

Dodawanie pakietu produkcyjnego serwera HTTP

W przypadku środowiska produkcyjnego wymagany jest zwykle serwer HTTP, który będzie dostarczał pliki HTML-a, CSS i JavaScriptu przeglądarce WWW. W omawianym przykładzie zdecydowałem się na użycie serwera Express — znajduje się on w tym samym pakiecie, z którego korzystam we wszystkich przykładach w tej części książki. Ten serwer jest dobrym wyborem dla wielu aplikacji internetowych. Zatrzymaj działanie narzędzi programistycznych Vue.js przez naciśnięcie klawiszy `Ctrl+C`, a następnie z poziomu katalogu `vueapp` aplikacji w powłocie wydaj polecenia przedstawione na listingu 22.8, aby zainstalować niezbędne pakiety.

Listing 22.8. Dodawanie pakietów pozwalających na wdrożenie aplikacji Vue.js

```
$ npm install --save-dev express@4.17.1
$ npm install --save-dev connect-history-api-fallback@1.6.0
```

Drugie polecenie powoduje zainstalowanie pakietu `connect-history-api-fallback`, który okazuje się użyteczny podczas wdrażania aplikacji wykorzystujących routing URL. Działanie pakietu polega na mapowaniu obsługiwanych przez aplikację żądań prowadzących do pliku `index.html` i zagwarantowaniu, że odświeżenie strony w przeglądarce WWW nie spowoduje wyświetlenia użytkownikowi błędu informującego o nieznalezieniu zasobu.

Tworzenie pliku dla trwałego magazynu danych

W celu utworzenia trwałego magazynu danych dla usługi sieciowej dodaj do katalogu `vueapp` plik o nazwie `data.json` zawierający kod przedstawiony na listingu 22.9.

Listing 22.9. Zawartość pliku `data.json` w katalogu `vueapp`

```
{
  "products": [
    { "id": 1, "name": "Kajak", "category": "Sporty wodne",
      "description": "Łódka przeznaczona dla jednej osoby.", "price": 275 },
    { "id": 2, "name": "Kamizelka ratunkowa", "category": "Sporty wodne",
      "description": "Chroni i dodaje uroku.", "price": 48.95 },
  ]
}
```

```

    { "id": 3, "name": "Piłka", "category": "Piłka nożna",
      "description": "Zatwierdzone przez FIFA rozmiar i waga.", "price": 19.50 },
    { "id": 4, "name": "Flagi naróżne", "category": "Piłka nożna",
      "description": "Nadadzą twojemu boisku profesjonalny wygląd.",
      "price": 34.95 },
    { "id": 5, "name": "Stadion", "category": "Piłka nożna",
      "description": "Składany stadion na 35 000 osób.", "price": 79500 },
    { "id": 6, "name": "Czapka", "category": "Szachy",
      "description": "Zwiększa efektywność mózgu o 75%.", "price": 16 },
    { "id": 7, "name": "Niestabilne krzesło", "category": "Szachy",
      "description": "Zmniejsza szanse przeciwnika.",
      "price": 29.95 },
    { "id": 8, "name": "Ludzka szachownica", "category": "Szachy",
      "description": "Przyjemna gra dla całej rodziny.", "price": 75 },
    { "id": 9, "name": "Błyszczący król", "category": "Szachy",
      "description": "Pokryty złotem i wysadzany diamentami król.", "price": 1200 }
  ],
  "orders": []
}

```

Tworzenie serwera

Aby utworzyć serwer dostarczający przeglądarce WWW aplikację i jej dane, dodaj do katalogu *vueapp* plik o nazwie *server.js* i umieść w nim kod przedstawiony na listingu 22.10.

Listing 22.10. Zawartość pliku *server.js* w katalogu *vueapp*

```

const express = require("express");
const jsonServer = require("json-server");
const history = require("connect-history-api-fallback");

const app = express();
app.use(history());
app.use("/", express.static("dist"));

const router = jsonServer.router("data.json");
app.use(jsonServer.bodyParser)
app.use("/api", (req, resp, next) => router(req, resp, next));

const port = process.argv[3] || 4003;
app.listen(port, () => console.log(`Serwer nasłuchuje na porcie numer ${port}`));

```

Polecenia zdefiniowane w pliku *server.js* konfiguruje pakiety *express* i *json-server* w taki sposób, aby udostępniać zawartość katalogu *dist*, w którym *Vue.js* umieszcza pliki JavaScriptu i HTML-a oraz nakazuje przeglądarce WWW ich wczytanie. Adresy URL poprzedzone prefiksem */api* będą obsługiwane przez usługę sieciową.

Używanie względnych adresów URL do obsługi żądań danych

Usługa sieciowa dostarczająca dane aplikacji będzie działała równolegle z serwerem frameworka *Vue.js*. Aby przygotować aplikację na wykonywanie żądań za pomocą pojedynczego portu, wymagana jest zmiana kodu klasy *HttpHandler*, jak pokazałem na listingu 22.11.

Listing 22.11. Używanie względnych adresów URL w kodzie pliku *httpHandler.ts* w katalogu *src/data*

```
import Axios from "axios";
import { Product, Order } from "../entities";

// const protocol = document.location.protocol;
// const hostname = document.location.hostname;
// const port = 4600;

const urls = {
  // products: `${protocol}/${hostname}:${port}/products`,
  // orders: `${protocol}/${hostname}:${port}/orders`
  products: "/api/products",
  orders: "/api/orders"
};

export class HttpHandler {

  loadProducts(): Promise<Product[]> {
    return Axios.get<Product[]>(urls.products).then(response => response.data);
  }

  storeOrder(order: Order): Promise<number> {
    let orderData = {
      lines: [...order.orderLines.values()].map(ol => ({
        productId: ol.product.id,
        productName: ol.product.name,
        quantity: ol.quantity
      }))
    }
    return Axios.post<{id: number}>(urls.orders, orderData)
      .then(response => response.data.id);
  }
}
```

Podane adresy URL są względne dla adresów używanych do obsługi żądań dokumentu HTML-a. Oznacza to stosowanie się do konwencji, zgodnie z którą żądania danych są poprzedzone prefiksem */api*.

Kompilowanie aplikacji

Z poziomu katalogu *vueapp* w powłoce należy wydać polecenie przedstawione na listingu 22.12, które spowoduje utworzenie produkcyjnej wersji aplikacji.

Listing 22.12. Tworzenie paczki produkcyjnej

```
$ npm run build
```

W wyniku procesu kompilacji nastąpi utworzenie w katalogu *build* zestawu zoptymalizowanych plików. Kompilacja aplikacji może chwilę potrwać.

Testowanie gotowej aplikacji

Aby upewnić się o prawidłowym skompilowaniu aplikacji i zastosowaniu wszystkich wprowadzonych zmian konfiguracyjnych, z poziomu katalogu *vueapp* w powłoce wydaj polecenie przedstawione na listingu 22.13.

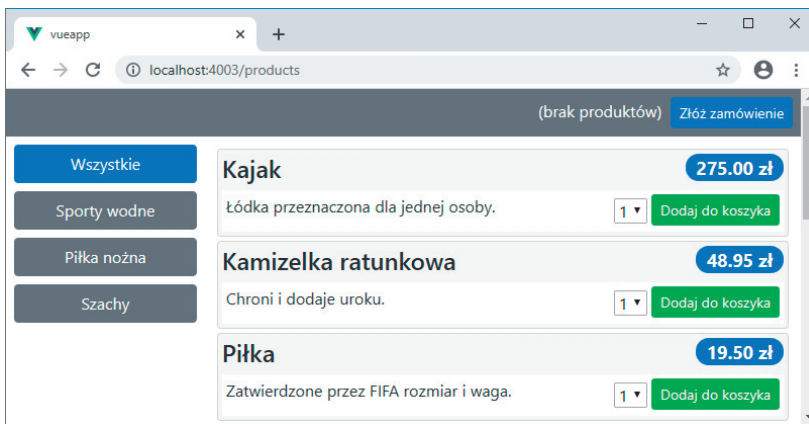
Listing 22.13. Uruchamianie serwera produkcyjnego

```
$ node server.js
```

Kod przedstawiony na listingu 22.13 zostanie wykonany i spowoduje wygenerowanie następujących danych wyjściowych:

```
Serwer nasłuchuje na porcie numer 4003
```

Otwórz okno przeglądarki WWW i przejdź pod adres *http://localhost:4003*, a zobaczysz działającą aplikację, jak pokazałem na rysunku 22.4.



Rysunek 22.4. Uruchomiona aplikacja dla środowiska produkcyjnego

Umieszczanie aplikacji w kontenerze

Ten rozdział zamierzam zakończyć utworzeniem dla przykładowej aplikacji kontenera, który następnie będzie można wdrożyć w środowisku produkcyjnym. Jeżeli w rozdziale 15. nie zainstalowałeś Dockera, będziesz to musiał zrobić teraz, jeśli chcesz wykonywać przykłady przedstawione w pozostałej części rozdziału.

Przygotowanie aplikacji

Pracę należy zacząć od utworzenia pliku konfiguracyjnego dla menedżera pakietów Node.js, który będzie użyty do pobrania pakietów dodatkowych wymaganych przez aplikację uruchamianą w kontenerze. W katalogu *vueapp* utwórz więc plik o nazwie *deploy-package.json* i umieść w nim kod przedstawiony na listingu 22.14.

Listing 22.14. Zawartość pliku *deploy-package.json* w katalogu *vueapp*

```
{
  "name": "vueapp",
  "description": "Aplikacja internetowa Vue.js",
  "repository": "https://github.com/Apress/essential-typescript",
  "license": "0BSD",
  "devDependencies": {
    "express": "4.17.1",
    "json-server": "0.16.3",
    "connect-history-api-fallback": "1.6.0"
  }
}
```

Sekcja `devDependencies` wymienia pakiety niezbędne do uruchomienia aplikacji w kontenerze. Wszystkie pakiety, dla których istnieją polecenia `import` w plikach kodu źródłowego aplikacji, zostały umieszczone w pliku paczki wygenerowanym przez webpack. Pozostałe opcje opisują aplikację i zostały umieszczone w kodzie w celu uniknięcia komunikatów ostrzeżeń generowanych podczas tworzenia kontenera.

Tworzenie kontenera Dockera

Aby zdefiniować kontener, należy dodać do katalogu *vueapp* plik o nazwie *Dockerfile* (bez żadnego rozszerzenia) i umieścić w nim kod przedstawiony na listingu 22.15.

Listing 22.15. Zawartość pliku *Dockerfile* w katalogu *vueapp*

```
FROM node:14.15.4

RUN mkdir -p /usr/src/vueapp

COPY dist /usr/src/vueapp/dist/
COPY data.json /usr/src/vueapp/
COPY server.js /usr/src/vueapp/
COPY deploy-package.json /usr/src/vueapp/package.json

WORKDIR /usr/src/vueapp

RUN echo 'package-lock=false' >> .npmrc
RUN npm install

EXPOSE 4003

CMD ["node", "server.js"]
```

Zawartość pliku *Dockerfile* wykorzystuje obraz bazowy skonfigurowany z Node.js i zawierający skopiowane pliki niezbędne do uruchomienia aplikacji, m.in. plik paczki z aplikacją oraz plik używany do zainstalowania pakietów Node.js wymaganych do uruchomienia aplikacji w środowisku produkcyjnym. Aby przyspieszyć proces tworzenia kontenera, do katalogu *vueapp* warto również dodać plik *.dockerignore* o zawartości przedstawionej na listingu 22.16. W omawianym przykładzie ten plik nakazuje Dockerowi zignorowanie katalogu *node_modules*, który nie jest wymagany w kontenerze, a jego przetworzenie będzie wymagało ogromnej ilości czasu.

Listing 22.16. Zawartość pliku `.dockerignore` w katalogu `vueapp`

```
node_modules
```

Z poziomu katalogu `vueapp` w powłoce wydaj polecenie przedstawione na listingu 22.17, które rozpocznie proces tworzenia kontenera zawierającego przykładową aplikację i wszystkie wymagane pakiety.

Listing 22.17. Tworzenie obrazu Dockera

```
$ docker build . -t vueapp -f Dockerfile
```

Obraz jest szablonem dla kontenera. Docker przetworzy polecenia zdefiniowane w pliku `Dockerfile`, wskazane pakiety Node.js zostaną pobrane i zainstalowane, a pliki konfiguracyjne i kodu źródłowego będą skopiowane do obrazu.

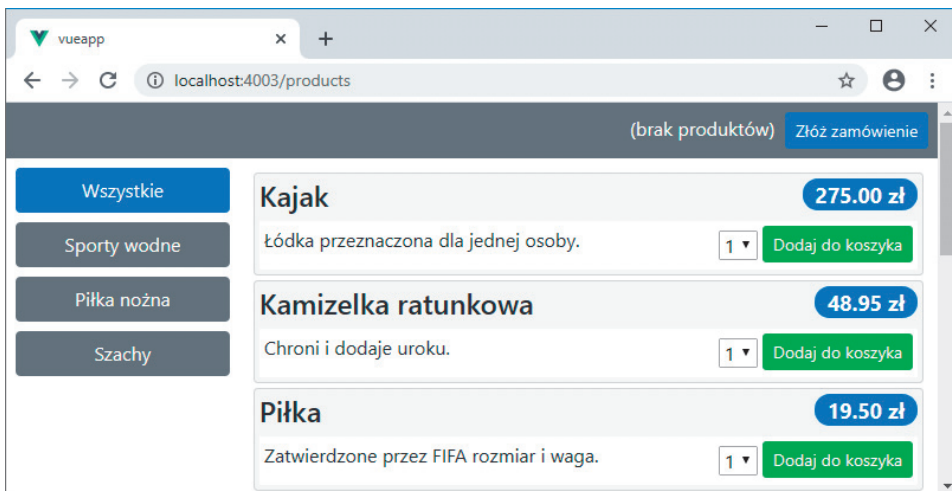
Uruchamianie aplikacji

Gdy obraz Dockera jest gotowy, można uruchomić nowy kontener za pomocą polecenia wymienionego na listingu 22.18.

Listing 22.18. Uruchamianie kontenera Dockera

```
$ docker run -p 4003:4003 vueapp
```

Gotową aplikację możesz przetestować w przeglądarce WWW. Wystarczy przejść pod adres `http://localhost:4003`, co spowoduje wyświetlenie odpowiedzi udzielonej przez serwer WWW uruchomiony w kontenerze, jak pokazałem na rysunku 22.5.



Rysunek 22.5. Przykładowa aplikacja uruchomiona w kontenerze

Jeżeli chcesz zatrzymać działanie kontenera, musisz zacząć od wydania polecenia przedstawionego na listingu 22.19.

Listing 22.19. Wyświetlenie listy kontenerów Dockera

```
$ docker ps
```

Dane wyjściowe tego polecenia to lista uruchomionych kontenerów (w celu zachowania zwięzłości i czytelności pominąłem część danych wyjściowych):

CONTAINER ID	IMAGE	COMMAND	CREATED
09761b008ab4	vueapp	"docker-entrypoint.s..."	43 seconds ago

Wykorzystując wartość wyświetloną w kolumnie *CONTAINER ID*, wydaj polecenie przedstawione na listingu 22.20.

Listing 22.20. Zatrzymywanie kontenera Dockera

```
$ docker stop 09761b008ab4
```

W tym momencie aplikacja jest gotowa do wdrożenia na dowolnej platformie obsługującej kontenery Dockera.

Podsumowanie

W tym rozdziale dokończyłem pracę nad przykładową aplikacją opartą na frameworku Vue.js i przygotowałem ją do wdrożenia w kontenerze Dockera. Każda z aplikacji internetowych utworzonych w tej części książki pokazuje odmienne podejście w zakresie użycia języka TypeScript w procesie tworzenia projektu i kładzie nacisk na różne funkcje oferowane przez TypeScript. W przypadku każdej z aplikacji otrzymany wynik jest dokładnie taki sam: znacznie lepsze środowisko pracy programisty, które może pomóc w zwiększeniu jego produktywności i unikaniu najczęściej popełnianych błędów podczas programowania w JavaScriptcie.

I to już wszystko, czego chciałem Cię nauczyć o języku TypeScript. Na początku pokazałem przykład utworzenia prostej aplikacji, a następnie dość dokładnie zaprezentowałem różne funkcje oferowane przez TypeScript i sposoby ich stosowania w systemie JavaScriptu. Życzę Ci wielu sukcesów podczas tworzenia własnych projektów TypeScriptu. Mam nadzieję, że lektura niniejszej książki dostarczyła Ci przynajmniej tyle radości, ile mnie dostarczyło jej napisanie.

PROGRAM PARTNERSKI

— GRUPY 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
Helion 



KOMPLEKSOWO SZKOLIMY NOWOCZESNY BIZNES



IT



BIZNES



PROJEKTY



PROCESY

NASZE SZKOLENIA SĄ PROWADZONE
ZGODNIE Z METODĄ

BLENDED LEARNING

modelem kształcenia, który łączy tradycyjne szkolenie
z dostępem do nowoczesnych narzędzi - wideokursów,
e-booków i audiobooków

T: 609 850 372 E: SZKOLENIA@HELION.PL

WWW.HELIONSZKOLENIA.PL