



TypeScript 4

Od początkującego do profesjonalisty

Wydanie II

Adam Freeman

Helion 

Apress®

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

Tłumaczenie: Robert Górczyński

ISBN: 978-83-283-8829-1

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>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

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



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 powinniśmy używać języka TypeScript?	52
	Funkcje języka TypeScript zwiększające produktywność programisty	52
	Poznanie wersji JavaScriptu	53

Co powinieneś 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 znajdziesz 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 TypeScriptu	125
Przygotowanie projektu	125
Struktura projektu	127
Używanie menedżera pakietów Node	128
Plik konfiguracyjny kompilatora TypeScriptu	131
Kompilacja kodu TypeScriptu	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 TypeScriptu	150
Przygotowanie projektu	150
Debugowanie kodu TypeScriptu	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 TypeScriptu	157
Wyłączanie reguł lintowania	159
Testy jednostkowe w TypeScriptie	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 złączeń	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

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*

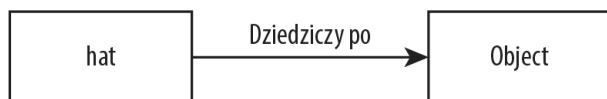
```
let hat = {
  name: "czapka",
  price: 100,
  getPriceIncTax() {
    return Number(this.price) * 1.2;
  }
};

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 wymieniłem 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}`);
  
```

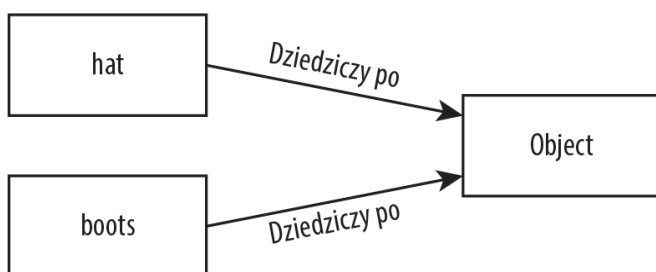


```
console.log(`czapka: ${hat.price}, ${hat.getPriceIncTax()}`);
console.log(`toString: ${hat.toString()}`);
```

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}`;
  }
}

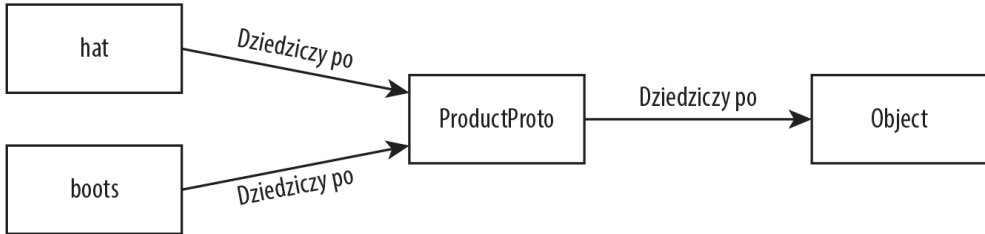
let hat = {
  name: "czapka",
  price: 100,
  getPriceIncTax() {
    return Number(this.price) * 1.2;
  }
};

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);
...

```

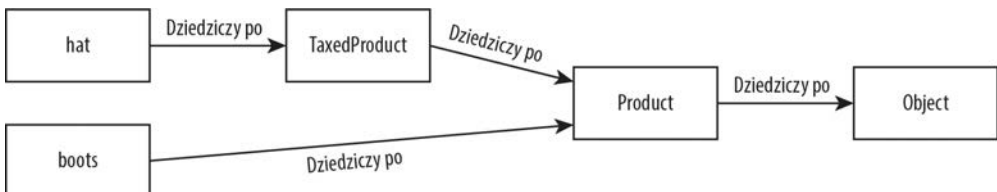
Metoda `call()` pozwala na przekazanie nowego obiektu następnemu konstruktorowi za pomocą jego wartości `this`. Drugim krokiem jest połączenie prototypów ze sobą.

```

...
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 rozwinięcia, 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 rozwinięcia 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</code> (obiekt)	Metoda zwraca tablicę zawierającą nazwy właściwości zdefiniowanych przez obiekt
<code>Object.values</code> (obiekt)	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
<code>set(klucz, wartość)</code>	Metoda przechowuje wartość z określonym kluczem
<code>get(klucz)</code>	Metoda wyszukuje wartości przechowywane w określonym kluczu
<code>keys()</code>	Metoda zwraca iterator dla kluczy w obiekcie Map
<code>values()</code>	Metoda zwraca iterator dla wartości w obiekcie Map
<code>entries()</code>	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
add(wartość)	Metoda dodaje wartość do zbioru
entries()	Wartość zwraca iterator dla elementów zbioru w kolejności ich dodawania
has(wartość)	Wartość zwraca wartość true, gdy zbiór zawiera określoną wartość
forEach(wywołanieZwrotne)	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 przedstawiam 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.

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 

TypeScript: koduj jak profesjonalista i pisz bogate aplikacje!

JavaScript jest jednym z najwartościowszych języków programowania, cechuje go wszechstronność i elastyczność. Ucząc się JavaScriptu, adept programowania nabiera dobrych nawyków i nabywa umiejętności wymaganych od profesjonalistów. Warto też poznać język TypeScript wraz z towarzyszącymi mu narzędziami. Projektant aplikacji internetowych, który programuje w TypeScriptie, może łatwo skorzystać z wielu dopracowanych frameworków. W ten sposób w pełni wykorzystuje się możliwości nowoczesnych przeglądarek i urządzeń mobilnych, a tworzony kod jest łatwy w utrzymaniu i wielokrotnym używaniu.

To drugie wydanie cenionego podręcznika dla początkujących programistów. Zawarty w nim materiał — podany jasno i przystępnie — pozwoli Ci w pełni poznać możliwości języka TypeScript 4. Naukę rozpoczniesz od zdobycia solidnych podstaw, a po przeanalizowaniu przejrzystych przykładów poznasz korzyści wynikające z używania TypeScriptu w rzeczywistych projektach. Nauczysz się pracy z API DOM, a także z takimi frameworkami jak Angular, Vue.js i React. Stopniowo będziesz nabierać wprawy w stosowaniu w praktyce najbardziej zaawansowanych funkcji. Dowiesz się też, z jakimi problemami najczęściej borykają się programiści TypeScript i jak je rozwiązywać. W efekcie uzyskasz bezpieczniejsze i bardziej produktywnie środowisko do tworzenia aplikacji internetowych.

W książce między innymi:

- przygotowanie systemu i zasady pracy z projektami
- solidne podstawy JavaScriptu i używanie języka TypeScript
- zaawansowane zagadnienia JavaScriptu
- pisanie kodu TypeScript działającego po stronie klienta i po stronie serwera
- tworzenie i rozwijanie aplikacji za pomocą API DOM, a także frameworków: Angular, React i Vue.js
- testowanie, debugowanie i wdrażanie kodu

Adam Freeman jest doświadczonym programistą, autorem wielu świetnie przyjętych książek o programowaniu. Tworzył duże systemy rozproszone (platformy e-commerce). Zajmował stanowiska kierownicze w różnych firmach, w tym w Netscape, Sun Microsystems, na giełdzie NASDAQ i w bankach o międzynarodowym zasięgu. Obecnie jest na emeryturze, poświęca czas na pisanie i bieganie na długich dystansach.

 Helion	<i>Sprawdź nasze szkolenia!</i>	KOD KORZYŚCI <i>Sięgnij po więcej!</i>	
 helion.pl	 SZKOLENIA AKADEMIA IT & BUSINESS	ISBN 978-83-283-8829-1	
 HELION SA ul. Kościuszki 1c 44-100 Gliwice tel.: 32 230 98 63 helion@helion.pl	HELIONSZKOLENIA.PL	 9 788328 388291	
INFORMATYKA W NAJLEPSZYM WYDANIU		Cena: 109,00 zł	

apress®