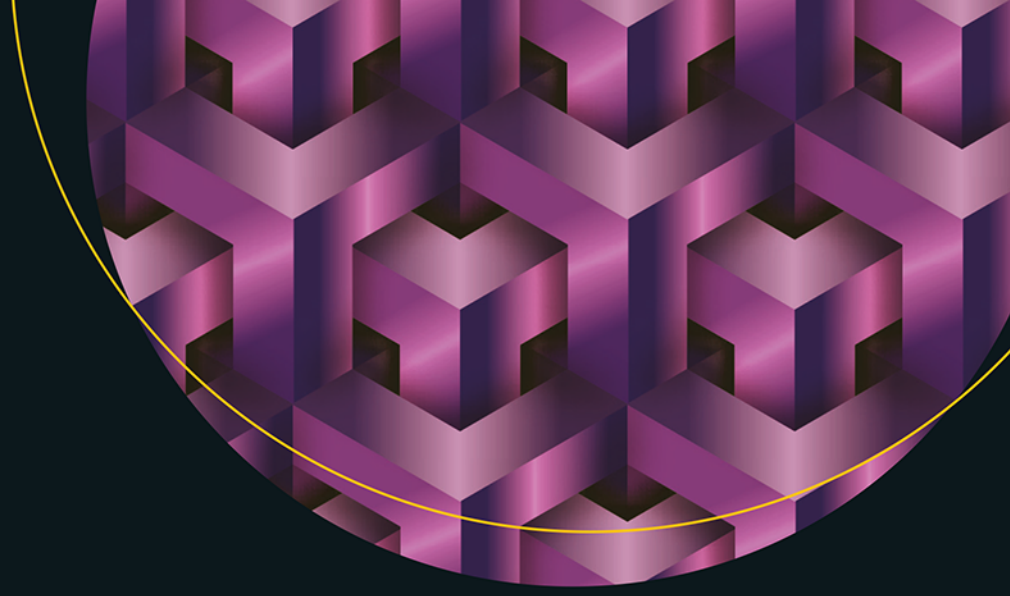


TypeScript 4

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

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.

Spis treści

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 obsłudze 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.

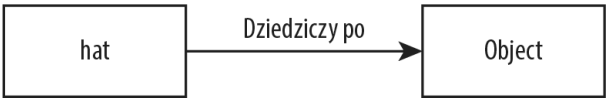
Listing 4.3. *Używanie obiektu w kodzie pliku index.js w katalogu primer*

```
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
`getPrototypeOf()`	Metoda zwraca prototyp obiektu
`setPrototypeOf()`	Metoda zmienia prototyp obiektu
`getOwnPropertyNames()`	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*

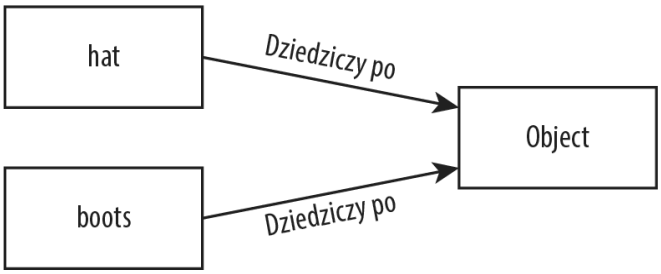
```
let hat = {
    name: "czapka",
    price: 100,
    getPriceIncTax() {
        return Number(this.price) * 1.2;
    }
};

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 łącze 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

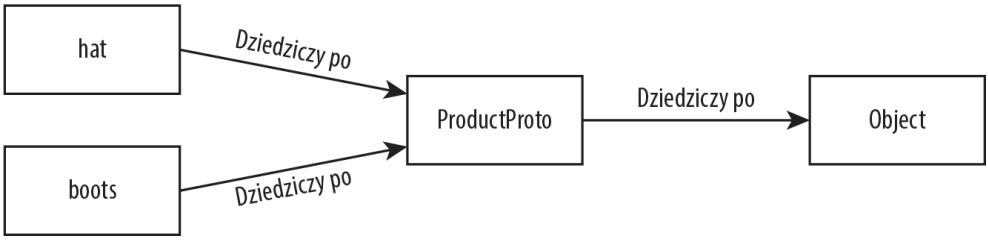
Zmiany w prototypie należy wprowadzać z zachowaniem dużej ostrożności, ponieważ mają one wpływ na wszystkie pozostałe obiekty aplikacji. Nowa funkcja `toString()` wykorzystana w kodzie na listingu 4.5 spowodowała wygenerowanie znacznie bardziej użytecznych danych wyjściowych dla obiektów `hat` i `boots`, 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*

Efektem 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());
```

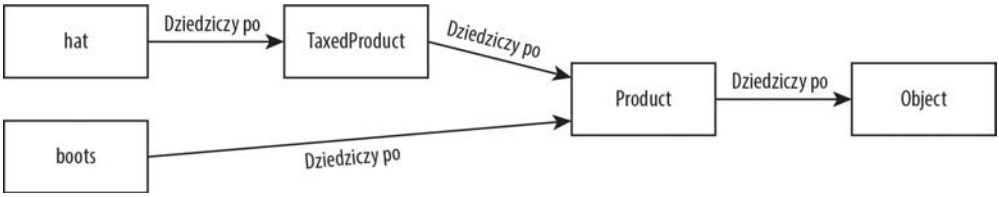
W celu przygotowania konstruktorów i ich prototypów w łańcuchu muszą być wykonane dwa kroki. Pierwszy krok to użycie metody `call()` do wywołania następnego konstruktora, aby nowe obiekty były tworzone prawidłowo. W omawianym przykładzie chcę, aby konstruktor `TaxedProduct()` został utworzony na podstawie 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 JavaScripcie 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.

Wprawdzie 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(`Product: ${ 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 JavaScripcie 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
`Object.keys(obiekt)`	Metoda zwraca tablicę zawierającą nazwy właściwości zdefiniowanych przez obiekt
`Object.values(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
`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
`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 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 przeznaczonej 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.

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 TypeScripcie, 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 produktywne 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
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-283-8829-1
9 788328 388291
Cena: 109,00 zł
INFORMATYKA W NAJLEPSZYM WYDANIU
Apress