Luis Atencio

Programowanie funkcyjne z JavaScriptem

Sposoby na lepszy kod

Helion

Tytuł oryginału: Functional Programming in JavaScript: How to improve your JavaScript programs using functional techniques

Tłumaczenie: Tomasz Walczak

ISBN: 978-83-283-3252-2

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/prfujs.zip

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

Printed in Poland.

Spis treści

2 Operacje wyższego poziomu w JavaScripcie

Zawartość rozdziału:

- Dlaczego JavaScript może być językiem funkcyjnym?
- JavaScript jako język umożliwiający programowanie w wielu paradygmatach
- Niemodyfikowalność i polityka modyfikacji
- Funkcje wyższego poziomu i funkcje pełnoprawne
- Domknięcia i zasięg
- Praktyczne zastosowanie domknięć

> W językach naturalnych nie występuje dominujący paradygmat. Podobnie jest z JavaScriptem. Programiści mogą wybierać spośród wielu podejść — proceduralnego, funkcyjnego i obiektowego — oraz łączyć je w odpowiedni sposób.
>
> — Angus Croll, *If Hemingway Wrote JavaScript*

Wraz z rozrastaniem się aplikacji rośnie też ich złożoność. Niezależnie od tego, jak wysoko oceniasz swoje umiejętności, bez odpowiednich modeli programowania nie da się uniknąć chaosu. W rozdziale 1. wyjaśniłem powody, dla których programowanie funkcyjne jest atrakcyjnym paradygmatem. Jednak same paradygmaty to tylko modele, które trzeba ożywić za pomocą właściwego języka.

W tym rozdziale zaprezentuję Ci krótki przegląd hybrydowego języka łączącego cechy obiektowe i funkcyjne. Jest to JavaScript. Oczywiście nie będzie to kompletne omówienie języka. Zamiast tego skoncentruję się na aspektach, które umożliwiają

funkcyjne używanie JavaScriptu, a także wyjaśnię jego wady. Jedną z nich jest brak zapewniania niemodyfikowalności. W tym rozdziale omawiam też funkcje wyższego poziomu i domknięcia (ang. *closure*). Te techniki to podstawa umożliwiająca pisanie kodu w JavaScripcie w funkcyjny sposób. To tyle tytułem wstępu — pora zaczynać.

2.1. Dlaczego JavaScript?

Zacząłem od wyjaśnienia, dlaczego warto stosować podejście funkcyjne. Inne pytanie, które można sobie zadać, brzmi: „Dlaczego JavaScript?". Odpowiedź jest prosta: „Ponieważ jest wszechobecny". JavaScript to obiektowy język ogólnego przeznaczenia z dynamicznie określanymi typami i bardzo ekspresywną składnią. Jest to jeden z najbardziej rozpowszechnionych języków programowania w historii. Możesz zetknąć się z nim w kontekście rozwijania aplikacji mobilnych, witryn, serwerów WWW, aplikacji na komputery stacjonarne, aplikacji wbudowanych, a nawet baz danych. Z powodu niezwykłej popularności JavaScriptu jako *języka internetu* można bezpiecznie założyć, że jest to zdecydowanie najczęściej stosowany język funkcyjny.

Choć składnia JavaScriptu przypomina składnię C, twórcy JavaScriptu w dużym stopniu inspirowali się językami funkcyjnymi takimi jak Lisp i Scheme. Podobieństwa polegają na obsłudze funkcji wyższego poziomu, domknięć, literałów tablicowych i innych mechanizmów, dzięki którym JavaScript to świetna technologia do stosowania technik funkcyjnych. Funkcje w JavaScripcie są główną *jednostką pracy*. Oznacza to, że funkcje nie tylko sterują operacjami aplikacji, ale też służą do definiowania obiektów, tworzenia modułów i obsługi zdarzeń.

JavaScript wciąż jest modyfikowany i usprawniany. Opis języka znajdziesz w standardzie ECMAScript (ES). W najnowszej jego wersji, ES6, pojawiło się wiele dodatkowych mechanizmów: funkcje ze strzałką, stałe, iteratory, obietnice (ang. *promises*) i inne konstrukty dobrze dostosowane do programowania funkcyjnego.

Choć JavaScript obejmuje wiele przydatnych mechanizmów funkcyjnych, warto zauważyć, że jest to język w równym stopniu obiektowy i funkcyjny. Niestety, aspekt funkcyjny często jest pomijany. Większość programistów stosuje operacje modyfikujące dane i imperatywne struktury kontrolne oraz zmienia stan obiektów. W podejściu funkcyjnym wszystko to trzeba wyeliminować. Mimo to uważam, że warto najpierw poświęcić trochę czasu na omówienie JavaScriptu jako języka obiektowego. Dzięki temu lepiej docenisz różnice między oboma paradygmatami i łatwiej będzie Ci zmienić podejście na funkcyjne.

2.2. Programowanie funkcyjne a programowanie obiektowe

Zarówno model funkcyjny, jak i model obiektowy umożliwiają rozwijanie średnich i dużych systemów. W językach hybrydowych (takich jak Scala i F#) oba te paradygmaty są łączone. JavaScript też to umożliwia. Aby dobrze opanować JavaScript, należy nauczyć się łączyć oba podejścia. Decyzję o ich proporcjach należy podejmować na podstawie osobistych preferencji i wymagań rozwiązywanego problemu. Zrozumienie,

w jakich miejscach podejścia funkcyjne i obiektowe się przenikają i różnią, pomoże Ci przechodzić od jednego do drugiego i myśleć w kategoriach dowolnego z tych modeli.

Zastanów się nad prostym modelem systemu do zarządzania uczelnią, obejmującego typ `Student`. W kontekście hierarchii klas lub typów naturalne jest przyjęcie, że `Student` to podtyp typu `Person` obejmującego podstawowe atrybuty takie jak imię, nazwisko, adres itd.

Obiektowy JavaScript

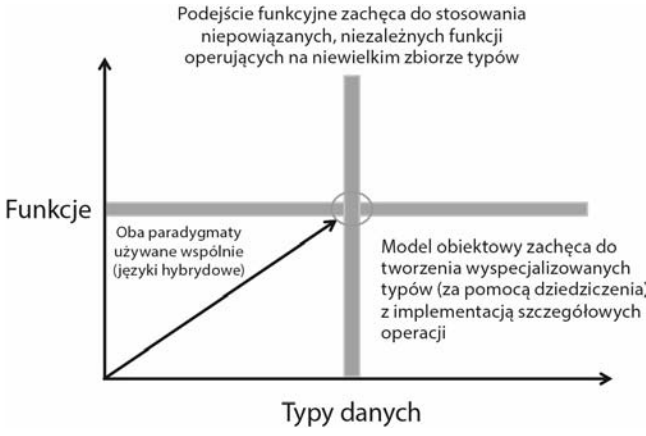
Gdy definiuję relację między obiektami i piszę, że jeden z nich jest **podtypem** (lub **typem pochodnym**) drugiego, mam na myśli relację między *prototypami* obiektów. Należy zauważyć, że choć JavaScript jest językiem obiektowym, nie występuje tu *klasyczne* dziedziczenie znane z innych języków (na przykład z Javy).

W wersji ES6 mechanizm do tworzenia relacji między prototypami obiektów został wzbogacony o „lukier składniowy" w postaci słów kluczowych `class` i `extends` (co zdaniem wielu osób było błędną decyzją). Nowa składnia pozwala jednoznacznie zapisać dziedziczenie obiektów, jednak ukrywa działanie i możliwości mechanizmu prototypów z JavaScriptu. W tej książce nie omawiam jednak obiektowego JavaScriptu (w końcowej części rozdziału polecam książkę ze szczegółowym omówieniem tego zagadnienia i innych tematów).

Nowe możliwości można dodać, tworząc bardziej specyficzny typ pochodny od `Student`, na przykład `CollegeStudent`. W programach obiektowych tworzenie nowych typów pochodnych jest podstawowym narzędziem umożliwiającym wielokrotne wykorzystanie kodu. Tu w typie `CollegeStudent` można wykorzystać wszystkie dane i operacje z typów nadrzędnych. Jednak dodawanie nowych mechanizmów do istniejących typów bywa kłopotliwe, jeśli dany mechanizm nie dotyczy wszystkich typów pochodnych. Choć pola `firstname` i `lastname` dotyczą typu `Person` i wszystkich typów pochodnych, pole `workAddress` powinno znaleźć się w typie `Employee` (pochodnym od `Person`), ale już nie w typie `Student`. Ten model omawiam dlatego, że podstawową różnicą między aplikacjami obiektowymi i funkcyjnymi jest sposób uporządkowania danych (właściwości obiektów) i operacji (funkcji).

W aplikacjach obiektowych (są one przede wszystkim imperatywne) często stosuje się opartą na obiektach hermetyzację, aby chronić integralność modyfikowalnego stanu — odziedziczonego lub utworzonego bezpośrednio. Aby modyfikować i pobierać stan, należy posługiwać się metodami instancji. Występuje tu więc ścisłe powiązanie między danymi obiektu a jego szczegółowymi operacjami. W ten sposób powstaje nierozłączna całość. Jest to cel programowania obiektowego i powód, dla którego główną formą abstrakcji w tym podejściu są obiekty.

W programowaniu funkcyjnym nie trzeba ukrywać danych przed jednostkami wywołującymi. W tym podejściu zwykle używany jest mniejszy zbiór bardzo prostych typów danych. Ponieważ wszystkie elementy są niemodyfikowalne, można bezpośrednio korzystać z obiektów za pomocą ogólnych funkcji znajdujących się poza zasięgiem obiektu. Oznacza to luźne powiązanie danych z operacjami. Na rysunku 2.1 widać, że w podejściu funkcyjnym zamiast szczegółowych metod instancji używane są ogólniejsze operacje, które mogą działać dla wielu typów danych. W tym paradygmacie to *funkcje stają się podstawową formą abstrakcji*.



Rysunek 2.1. Programowanie obiektowe zachęca do logicznego łączenia wielu typów danych z wyspecjalizowanymi operacjami, natomiast w programowaniu funkcyjnym operacje są łączone z typami danych za pomocą kompozycji. Istnieje optymalny punkt, w którym można produktywnie stosować oba paradygmaty. W językach hybrydowych, takich jak Scala, F# i JavaScript, można posługiwać się oboma podejściami

Na rysunku 2.1 widać, że oba paradygmaty różnią się w górnej i prawej części wykresu. W praktyce najlepsze fragmenty kodu obiektowego, jakie widziałem, były napisane z użyciem obu paradygmatów (co odpowiada przecięciu linii na wykresie). W tym modelu obiekty należy traktować jak niemodyfikowalne encje (wartości) i zapisywać związane z nimi operacje w postaci funkcji działających na tych obiektach. Przyjrzyj się następującej metodzie z typu `Person`:

```
get fullname() {
    return [this._firstname, this._lastname].join(' ');
}
```

W metodach łatwo jest używać słowa „this", aby uzyskać dostęp do stanu danego obiektu.

Można ją przenieść poza typ w następujący sposób:

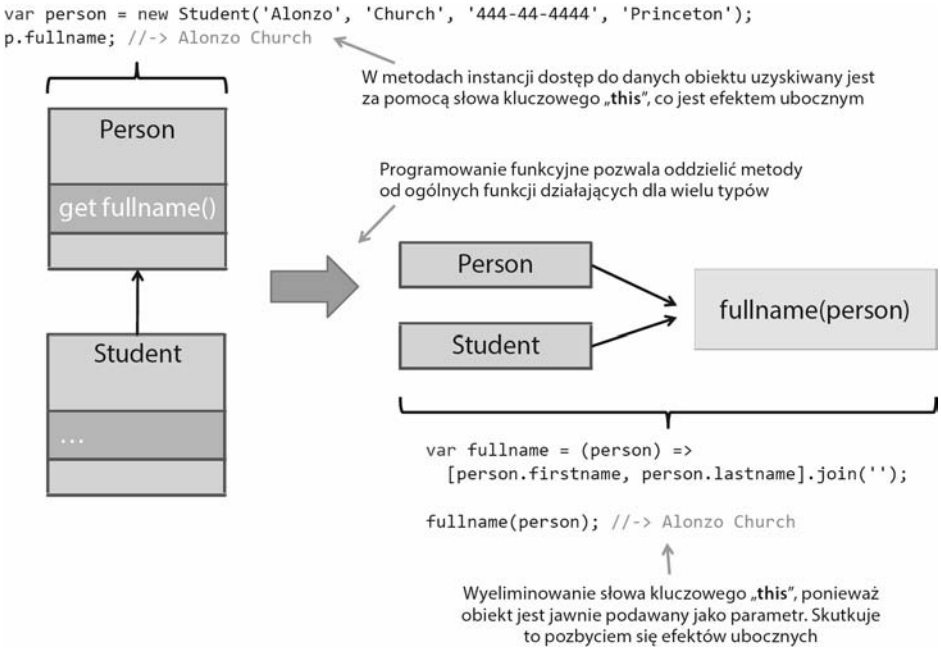
```
var fullname =
    person => [person.firstname, person.lastname].join(' ');
```

Obiekt „this" jest zastępowany przekazanym obiektem.

Wiesz już, że JavaScript to język z dynamicznie określanymi typami. To oznacza, że nie musisz jawnie podawać typu obok nazw obiektów. Dlatego wywołanie `fullname()` działa dla dowolnego typu pochodnego od `Person` (i dowolnego obiektu z właściwościami `firstname` oraz `lastname`), co ilustruje rysunek 2.2. Z powodu dynamicznej natury JavaScript umożliwia stosowanie uogólnionych funkcji polimorficznych. To oznacza, że funkcje używające referencji do typów bazowych (na przykład typu `Person`) działają także dla obiektów typów pochodnych (takich jak `Student` lub `CollegeStudent`).

Na rysunku 2.2 widać, że przekształcenie `fullname()` w niezależną funkcję pozwala zrezygnować z używania słowa kluczowego `this` przy dostępie do danych obiektów. Używanie słowa `this` to problematyczne rozwiązanie, ponieważ daje dostęp do danych z poziomu instancji poza zasięgiem metody, co oznacza efekty uboczne. W programowaniu funkcyjnym dane obiektu nie są ściśle powiązane z konkretnymi fragmentami rozwiązania, co ułatwia wielokrotne wykorzystanie kodu i jego konserwację.

Zamiast tworzyć wiele typów pochodnych, możesz rozbudować działanie funkcji, przekazując jako argumenty inne funkcje. Aby to zilustrować, na listingu 2.1 definiuję prosty model danych. Listing ten obejmuje klasę `Student` pochodną od `Person`. Model ten jest używany w większości przykładów z tej książki.



Rysunek 2.2. W programowaniu obiektowym istotne jest tworzenie hierarchii dziedziczenia (na przykład typu Student po typie Parent), w której metody i dane są ze sobą ściśle powiązane. W programowaniu funkcyjnym preferowane są ogólne funkcje polimorficzne działające dla różnych typów danych. W tym podejściu programiści unikają stosowania słowa this

Listing 2.1. Definicje klas Person i Student

```
class Person {
   constructor(firstname, lastname, ssn) {
      this._firstname = firstname;
      this._lastname = lastname;
      this._ssn = ssn;
      this._address = null;
      this._birthYear = null;
   }

   get ssn() {
      return this._ssn;
   }

   get firstname() {
      return this._firstname;
   }

   get lastname() {
      return this._lastname;
   }

   get address() {
      return this._address;
   }
```

```
   get birthYear() {
      return this._birthYear;
   }

   set birthYear(year) {
      this._birthYear = year;
   }

   set address(addr){
      this._address = addr;
   }

   toString() {
      return `Person(${this._firstname}, ${this._lastname})`;
   }
}

class Student extends Person {
   constructor(firstname, lastname, ssn, school) {
      super(firstname, lastname, ssn);
      this._school = school;
   }

   get school() {
      return this._school;
   }
}
```

Settery nie mają umożliwiać modyfikowania obiektów. Mają natomiast pozwalać łatwo tworzyć obiekty o różnych właściwościach bez używania długich konstruktorów. Po utworzeniu obiektów i określeniu ich danych stan nigdy się nie zmienia. W dalszej części rozdziału zobaczysz, jak to zapewnić.

Pobieranie i uruchamianie przykładowego kodu

Oryginalny przykładowy kod z tej książki znajdziesz na stronach *http://www.manning.com/books/functionalprogramming-in-javascript* i *https://github.com/luijar/functional-programming-js*. Spolszczona wersja jest dostępna pod adresem *ftp://ftp.helion.pl/przyklady/prfujs.zip*. Możesz swobodnie wypróbować projekty i samodzielnie przećwiczyć programowanie funkcyjne. Zachęcam do uruchamiania dowolnych testów jednostkowych i eksperymentowania z różnymi programami. W czasie, gdy powstaje ta książka, nie wszystkie mechanizmy z wersji ES6 JavaScriptu są zaimplementowane w każdej przeglądarce. Dlatego używam transpilatora Babel (jego wcześniejsza nazwa to 6to5) do przekształcania kodu z wersji ES6 na ES5.

Niektóre mechanizmy nie wymagają transpilacji i można je stosować po włączeniu odpowiedniego ustawienia w przeglądarce (na przykład *Eksperymentalny JavaScript* w przeglądarce Chrome). Jeśli uruchamiasz przeglądarkę w trybie eksperymentalnym, ważne jest, aby włączyć **tryb strict**. W tym celu dodaj instrukcję `'use strict';` na początku pliku z kodem w JavaScripcie.

Zadanie polega na tym, aby po pobraniu danych osoby znaleźć wszystkich jej znajomych mieszkających w tym samym kraju. Inne zadanie to wyszukiwanie na podstawie danych studenta innych studentów z tego samego państwa i tej samej uczelni. W rozwiązaniu obiektowym operacje są ściśle powiązane (za pomocą słów `this` i `super`) z typami bazowym i pochodnym.

```
// W klasie Person
peopleInSameCountry(friends) {
   var result = [];
   for (let idx in friends) {
```

```
      var friend = friends [idx];
      if (this.address.country === friend.address.country) {
         result.push(friend);
      }
   }
   return result;
};

// W klasie Student
studentsInSameCountryAndSchool(friends) {
   var closeFriends = super.peopleInSameCountry(friends);
   var result = [];
   for (let idx in closeFriends) {
      var friend = closeFriends[idx];
      if (friend.school === this.school) {
         result.push(friend);
      }
   }
   return result;
};
```

Słowo „super" służy do żądania danych od klasy bazowej.

Natomiast w programowaniu funkcyjnym ważne są czystość funkcji i przejrzystość referencyjna, dlatego dzięki odizolowaniu działań od stanu można dodawać nowe operacje, definiując i tworząc za pomocą kompozycji nowe funkcje operujące na określonych typach. W ten sposób powstają proste obiekty odpowiedzialne za przechowywanie danych i wszechstronne funkcje, które mogą operować na tych obiektach przekazywanych jako argumenty. Funkcje te można łączyć za pomocą kompozycji, aby uzyskać wyspecjalizowane możliwości. Na razie nie uczyłeś się jeszcze kompozycji (omawiam ją w rozdziale 4.), jest ona jednak istotna, aby można było pokazać następną ważną różnicę między opisywanymi paradygmatami. To, co dziedziczenie zapewnia w kontekście obiektowym, kompozycja umożliwia w programowaniu funkcyjnym, ponieważ pozwala stosować nowe operacje do różnych typów danych[1]. Do wykonywania kodu posłuży następujący zbiór danych:

```
var curry = new Student('Haskell', 'Curry',
   '111-11-1111', 'Penn State');
curry.address = new Address('USA');

var turing = new Student('Alan', 'Turing',
   '222-22-2222', 'Princeton');
turing.address = new Address('Anglia');

var church = new Student('Alonzo', 'Church',
   '333-33-3333', 'Princeton');
church.address = new Address('USA');

var kleene = new Student('Stephen', 'Kleene',
   '444-44-4444', 'Princeton');
kleene.address = new Address('USA');
```

[1] Dotyczy to w większym stopniu użytkowników modelu obiektowego niż samego paradygmatu. Wielu ekspertów, w tym „banda czterech", zaleca stosowanie kompozycji obiektów zamiast dziedziczenia klas (zgodnie z zasadą podstawiania Liskov).

W podejściu obiektowym do wyszukiwania wszystkich innych studentów uczęszczających na tę samą uczelnię służy metoda z typu `Student`:

```
church.studentsInSameCountryAndSchool([curry, turing, kleene]);
//-> [kleene]
```

W rozwiązaniu funkcyjnym problem jest rozbity na mniejsze funkcje:

```
function selector(country, school) {
   return function(student) {
      return student.address.country() === country &&
         student.school() === school;
   };
}

var findStudentsBy = function(friends, selector) {
   return friends.filter(selector);
};

findStudentsBy([curry, turing, church, kleene],
   selector('USA', 'Princeton'));
```

Funkcja selector potrafi porównywać państwa i uczelnie powiązane ze studentami.

Poruszanie się po grafach obiektów. Dalej w rozdziale pokazuję lepszy sposób dostępu do atrybutów obiektów.

Użycie operacji filter do tablicy i dodanie specjalnej operacji za pomocą funkcji selector.

//-> [church, kleene]

Dzięki zastosowaniu programowania funkcyjnego możesz utworzyć zupełnie nową funkcję, `findStudentsBy`, która jest znacznie łatwiejsza w użyciu niż kod obiektowy. Pamiętaj, że nowa funkcja działa dla dowolnych obiektów powiązanych z typem `Person`, a także dla dowolnej kombinacji uczelni i państwa.

Ten przykład dobrze ilustruje różnice między omawianymi paradygmatami. W projektowaniu obiektowym nacisk położony jest na naturę danych i relacje między nimi, natomiast w programowaniu funkcyjnym ważne są wykonywane operacje, czyli działanie kodu. Tabela 2.1 zawiera podsumowanie najważniejszych różnic, na które warto zwrócić uwagę. Omawiam je w tym rozdziale i w dalszych fragmentach książki.

Tabela 2.1. Porównanie wybranych ważnych cech programowania obiektowego i funkcyjnego. Cechy te omawiam w książce

	Programowanie funkcyjne	Programowanie obiektowe
Jednostka kompozycji	Funkcje	Obiekty (klasy)
Styl programowania	Deklaratywny	Imperatywny
Dane i operacje	Luźno powiązane za pomocą czystych i niezależnych funkcji	Ściśle powiązane za pomocą klas z metodami
Zarządzanie stanem	Obiekty są traktowane jak niemodyfikowalne wartości	Głównie modyfikowanie obiektów za pomocą metod instancji
Sterowanie przepływem	Funkcje i rekurencja	Pętle i instrukcje warunkowe
Bezpieczeństwo wątków	Umożliwia programowanie współbieżne	Trudne do osiągnięcia
Hermetyzacja	Niepotrzebna, ponieważ dane są niemodyfikowalne	Potrzebna do ochrony integralności danych

Mimo różnic między tymi paradygmatami tworzenie aplikacji z użyciem ich obu może okazać się bardzo przydatnym podejściem. Z jednej strony uzyskujesz rozbudowany

model dziedziny z naturalnymi relacjami między tworzącymi go typami. Z drugiej strony otrzymujesz zestaw czystych funkcji operujących na tych typach. To, z którego modelu będziesz korzystał w większym stopniu, zależy od tego, na ile komfortowo czujesz się, posługując się oboma paradygmatami. Ponieważ JavaScript jest w równym stopniu obiektowy i funkcyjny, używanie go w sposób funkcyjny wymaga poświęcenia specjalnej uwagi kontrolowaniu zmian stanu.

2.2.1. Zarządzanie stanem obiektów w JavaScripcie

Stan programu można zdefiniować jako zapis danych przechowywanych w określonym momencie we wszystkich obiektach. Niestety, JavaScript jest jednym z najgorszych języków, jeśli chodzi o zabezpieczanie stanu obiektów. Obiekty w JavaScripcie są wysoce dynamiczne. Można w dowolnym momencie modyfikować, dodawać i usuwać właściwości obiektów. Założenie, że na listingu 2.1 właściwość `_address` w typie `Person` jest hermetyczna, jest błędne (podkreślenie w nazwie to zabieg wyłącznie składniowy). Także poza klasą `Person` można uzyskać pełny dostęp do tej właściwości i wykonywać na niej dowolne operacje (a nawet ją usunąć).

Taka swoboda pociąga za sobą dużą odpowiedzialność. Choć opisany model pozwala wykonywać wiele wygodnych operacji, na przykład dynamicznie tworzyć właściwości, może prowadzić do powstawania bardzo trudnego w konserwacji kodu, gdy program jest średni lub duży.

W rozdziale 1. wspomniałem, że praca z czystymi funkcjami ułatwia konserwację i analizowanie kodu. Czy istnieje coś takiego jak „czysty obiekt"? Niemodyfikowalny obiekt z niemodyfikującymi danych operacjami można uznać za czysty. To samo wnioskowanie, które przedstawiłem dla funkcji, można zastosować także do prostych obiektów. Zarządzanie stanem w JavaScripcie jest niezwykle istotne, jeśli chcesz używać go jak języka funkcyjnego. Znane są praktyki i wzorce pozwalające zapewnić niemodyfikowalność (poznasz je w dalszych punktach), jednak zapewnienie kompletnej hermetyzacji i ochrony danych zależy od dyscypliny programisty.

2.2.2. Traktowanie obiektów jak wartości

Łańcuchy znaków i liczby to prawdopodobnie najłatwiejsze typy danych do obsługi w dowolnym języku programowania. Jak myślisz, dlaczego tak się dzieje? Po części wynika to z tego, że te proste typy są z natury niemodyfikowalne. Zapewnia to programistom spokój, którego nie gwarantują typy definiowane przez użytkownika. W programowaniu funkcyjnym typy działające w ten sposób są nazywane **wartościami** (ang. *values*). W rozdziale 1. nauczyłeś się zwracać uwagę na niemodyfikowalność. Wymaga to traktowania każdego obiektu jak wartości. Można wtedy korzystać z funkcji przekazujących obiekty i nie martwić się o to, że obiekty zostaną zmodyfikowane.

Mimo związanego z klasami „lukru składniowego" dodanego w wersji ES6 obiekty w JavaScripcie nie są niczym więcej niż zbiorami atrybutów, które można w dowolnym momencie dodawać, usuwać i modyfikować. Jak można zapobiec takim operacjom? Wiele języków programowania udostępnia konstrukty zapewniające niemodyfikowalność właściwości obiektu. Jest to na przykład słowo kluczowe `final` w Javie.

W językach takich jak F# zmienne są domyślnie niemodyfikowalne, chyba że programista postanowi inaczej. Obecnie w JavaScripcie brakuje podobnych możliwości. Choć nie da się zmienić wartości typu prostego, możliwa jest modyfikacja stanu zmiennej wskazującej taką wartość. Dlatego potrzebna jest możliwość tworzenia, a przynajmniej symulowania niemodyfikowalnych referencji, co pozwoli korzystać z obiektów typów definiowanych przez użytkownika w taki sposób, jakby były niemodyfikowalne.

W wersji ES6 do tworzenia stałych referencji służy słowo kluczowe `const`. Jest to krok w dobrym kierunku, ponieważ zadeklarowanych tak stałych nie można ponownie zadeklarować. Nie można też zmienić przypisanych do nich wartości. W programowaniu funkcyjnym w praktyce możesz stosować słowo kluczowe `const` do dodawania do programu prostych danych konfiguracyjnych (łańcuchów znaków z adresami URL, nazw baz danych itd.), gdy są one potrzebne. Choć odczyt danych z zewnętrznej zmiennej to efekt uboczny, stałe działają w specjalny sposób, dzięki czemu nie zmieniają się nieoczekiwanie między wywołaniami funkcji. Oto przykład ilustrujący deklarowanie stałej:

```
const gravity_ms = 9.806;
gravity_ms = 20;
```

Środowisko uruchomieniowe JavaScriptu nie pozwoli na to ponowne przypisanie.

Jednak to nie zapewnia niemodyfikowalności na poziomie potrzebnym w programowaniu funkcyjnym. Możesz zapobiec ponownemu przypisaniu wartości do zmiennej, jak jednak uniemożliwić zmianę wewnętrznego stanu obiektu? Pokazany poniżej kod jest w pełni dozwolony:

```
const student = new Student('Alonzo', 'Church',
   '666-66-6666', 'Princeton');

student.lastname = 'Mourning';
```

Modyfikacja właściwości.

Potrzebniejsza jest bardziej ścisła polityka zapewniania niemodyfikowalności. Dobrą strategią ochrony przed modyfikacjami jest hermetyzacja. W prostych strukturach obiektowych jedną z możliwości jest zastosowanie wzorca **obiekt traktowany jak wartość** (ang. *Value Object*). Równość obiektów traktowanych jak wartości nie jest oparta na tożsamości lub referencji, a jedynie na samej wartości. Po zadeklarowaniu takiego obiektu jego stan nie może się zmieniać. Oprócz liczb i łańcuchów znaków przykładowymi obiektami tego rodzaju są `tuple`, `pair`, `point`, `zipCode`, `coordinate`, `money`, `date` itd. Oto kod funkcji `zipCode`:

```
function zipCode(code, location) {
   let _code = code;
   let _location = location || '';

   return {
      code: function () {
         return _code;
      },
      location: function () {
         return _location;
      },
      fromString: function (str) {
         let parts = str.split('-');
```

```
        return zipCode(parts[0], parts[1]);
      },
      toString: function () {
        return _code + '-' + _location;
      }
   };
}

const princetonZip = zipCode('08544', '3345');
princetonZip.toString(); //-> '08544-3345'
```

W JavaScripcie możesz zastosować funkcje i chronić dostęp do wewnętrznego stanu kodu pocztowego, zwracając *interfejs literału obiektowego*, który udostępnia jednostce wywołującej niewielki zbiór metod i traktuje pola `_code` i `_location` jako zmienne pseudoprywatne. Te zmienne są dostępne w literale obiektowym tylko za pomocą *domknięć*, z którymi zapoznasz się w dalszej części rozdziału.

Zwracany obiekt działa jest wartość typu prostego bez metod modyfikujących stan[2]. Dlatego choć metoda `toString` nie jest czystą funkcją, działa jak ona i w czysty sposób zwraca łańcuch znaków reprezentujący dany obiekt. Obiekty traktowane jak wartości są „lekkie" i łatwo można z nich korzystać zarówno w modelu funkcyjnym, jak i w podejściu obiektowym. Za pomocą takich obiektów i słowa kluczowego `const` można tworzyć obiekty działające podobnie jak łańcuchy znaków i liczby. Przyjrzyj się następnemu przykładowi:

```
function coordinate(lat, long) {
   let _lat = lat;
   let _long = long;

   return {
      latitude: function () {
        return _lat;
      },
      longitude: function () {
        return _long;
      },
      translate: function (dx, dy) {
        return coordinate(_lat + dx, _long + dy);
      },
      toString: function () {
        return '(' + _lat + ',' + _long + ')';
      }
   };
}

const greenwich = coordinate(51.4778, 0.0015);
greenwich.toString(); //-> '(51.4778, 0.0015)'
```

Zwraca nowy obiekt z przekształconymi współrzędnymi.

Używanie metod (takich jak `translate`) do zwracania nowych obiektów to inny sposób zapewniania niemodyfikowalności. Przekształcenie danego obiektu skutkuje utworzeniem nowego obiektu typu `coordinate`:

[2] Wewnętrzny stan obiektu można chronić, jednak działanie obiektu może się zmieniać, ponieważ dozwolone jest dynamiczne usuwanie lub zastępowanie jego metod.

```
greenwich.translate(10, 10).toString(); //-> '(61.4778, 10.0015)'
```

„Obiekt traktowany jak wartość" to obiektowy wzorzec projektowy zainspirowany programowaniem funkcyjnym. Jest to kolejny przykład pokazujący, że oba omawiane paradygmaty w elegancki sposób się uzupełniają. Ten wzorzec ilustruje idealne rozwiązanie, jednak w praktyce nie wystarcza do modelowania całych dziedzin problemowych. Kod zwykle musi obsługiwać dane hierarchiczne (takie jak w przykładzie z typami `Person` i `Student`), a także komunikować się z już istniejącymi obiektami. Na szczęście JavaScript udostępnia funkcję `Object.freeze`, która pozwala radzić sobie w takich sytuacjach.

2.2.3. Głębokie zamrażanie potencjalnie zmiennych elementów

W nowej składni klas w JavaScripcie nie występują słowa kluczowe służące do oznaczania pól jako niemodyfikowalnych, jednak obsługiwany jest wewnętrzny mechanizm do blokowania modyfikacji pól, wykorzystujący ukryte metawłaściwości obiektów, takie jak `writable`. Dzięki ustawieniu tej właściwości na `false` funkcja `Object.freeze()` JavaScriptu zapobiega zmianom stanu obiektu. Zacznijmy od zamrożenia obiektu `person` z listingu 2.1:

```
var person = Object.freeze(new Person('Haskell', 'Curry', '444-44-4444'));
person.firstname = 'Bob';                                    ← Niedozwolone
```

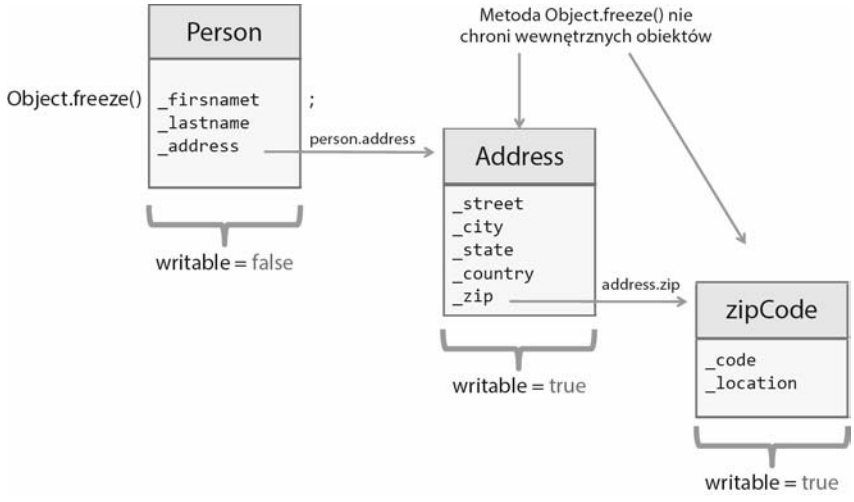
Wykonanie tego kodu sprawia, że atrybuty obiektu `person` stają się przeznaczone tylko do odczytu. Próba ich modyfikacji (tu dotyczy to pola `_firstname`) prowadzi do błędu:

```
TypeError: Cannot assign to read only property '_firstname' of #<Person>
```

Wywołanie `Object.freeze()` zamraża także odziedziczone atrybuty. Dlatego zamrożenie obiektu typu `Student` działa w ten sam sposób i uwzględnia łańcuch prototypów obiektu, zabezpieczając wszystkie atrybuty odziedziczone po typie `Person`. Technika ta nie zamraża jednak atrybutów obiektów zagnieżdżonych (zobacz rysunek 2.3).

Oto definicja typu `Address`:

```
class Address {
   constructor(country, state, city, zip, street) {
      this._country = country;
      this._state = state;
      this._city = city;
      this._zip = zip;
      this._street = street;
   }

   get street() {
      return this._street;
   }

   get city() {
      return this._city;
   }

   get state() {
      return this._state;
```



Rysunek 2.3. Choć typ Person został zamrożony, nie dotyczy to jego wewnętrznych właściwości obiektowych (na przykład _address). Dlatego w dowolnym momencie można zmodyfikować pole person.address.country. Ponieważ technika ta dotyczy tylko zmiennych najwyższego poziomu, jest to zamrażanie płytkie

```
   }

   get zip() {
      return this._zip;
   }

   get country() {
      return this._country;
   }
}
```

Niestety, pokazany poniżej kod nie powoduje żadnych błędów:

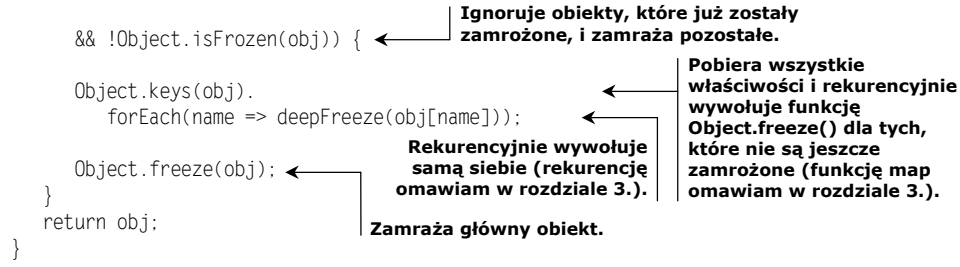
```
var person = new Person('Haskell', 'Curry', '444-44-4444');
person.address = new Address(
   'USA', 'NJ', 'Princeton',
   zipCode('08544','1234'), 'Alexander St.');

person = Object.freeze(person);

person.address._country = 'Francja'; //-> dozwolone!
person.address.country; //-> 'Francja'
```

`Object.freeze()` to płytka operacja. Aby rozwiązać ten problem, trzeba ręcznie zamrozić zagnieżdżone elementy obiektu, co ilustruje listing 2.2.

Listing 2.2. Funkcja rekurencyjna do głębokiego zamrażania obiektów

```
var isObject = (val) => val && typeof val === 'object';

   function deepFreeze(obj) {
      if(isObject(obj)
```

Pomija funkcje. Choć technicznie w JavaScripcie funkcje można modyfikować, tu koncentrujemy się na właściwościach w postaci danych.

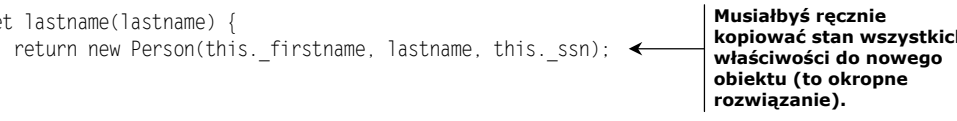



```
    && !Object.isFrozen(obj)) {

    Object.keys(obj).
       forEach(name => deepFreeze(obj[name]));

    Object.freeze(obj);
  }
  return obj;
}
```

Przedstawiłem tu wybrane techniki, które można zastosować, aby zapewnić odpowiedni poziom niemodyfikowalności w kodzie. Nierealne jest jednak oczekiwanie, że można tworzyć całe aplikacje bez modyfikowania stanu. Dlatego ścisłe reguły tworzenia nowych obiektów na podstawie pierwotnych (tak jak za pomocą wywołania `coordinate.trans` ↪`late()`) są bardzo przydatne, jeśli chcesz ograniczyć złożoność i zawiłość aplikacji w JavaScripcie. Dalej omawiam najlepszy sposób scentralizowanego zarządzania zmianami obiektów z zachowaniem niemodyfikowalności za pomocą funkcyjnej techniki o nazwie *soczewki* (ang. *lenses*).

2.2.4. Poruszanie się po grafach obiektów i ich modyfikowanie za pomocą soczewek

W programowaniu obiektowym programiści przyzwyczajeni są do wywoływania metod, które zmieniają wewnętrzną zawartość stanowych obiektów. Wadą tego podejścia jest to, że nie można zagwarantować, jaki skutek będzie miało pobranie stanu. Ponadto fragmenty systemu oczekujące, że obiekt nie będzie się zmieniał, mogą wtedy działać nieprawidłowo. Możesz zdecydować się na strategię *kopiowania przy zapisie* i zwracać nowe obiekty dla każdego wywołania metody. Jest to jednak żmudna i narażona na błędy technika (delikatnie mówiąc). Prosty setter w klasie `Person` wyglądałby wtedy tak:




```
set lastname(lastname) {
   return new Person(this._firstname, lastname, this._ssn);
};
```

Teraz wyobraź sobie, że podobne zabiegi musisz stosować do każdej właściwości każdego typu w modelu dziedziny. Potrzebny jest sposób zmieniania obiektów stanowych z zachowaniem niemodyfikowalności, bez nadmiernej ingerencji w rozwiązanie i bez konieczności pisania za każdym razem szablonowego kodu. **Soczewki** (nazywane też **referencjami funkcyjnymi**; ang. *functional references*) to dostępne w programowaniu funkcyjnym rozwiązanie problemu dostępu do atrybutów stanowych typów danych i operowania nimi z zachowaniem niemodyfikowalności. Wewnętrznie soczewki działają podobnie jak strategia kopiowania przy zapisie. Używany jest wewnętrzny komponent do obsługi przechowywania stanu, który potrafi poprawnie zarządzać stanem i go kopiować. Nie musisz samodzielnie pisać tego komponentu. Wystarczy zastosować funkcyjną bibliotekę JavaScript Ramda.js (szczegółowe informacje o niej i o innych bibliotekach znajdziesz w dodatku). Ramda domyślnie udostępnia wszystkie możliwości za pomocą globalnego obiektu `R`. Za pomocą wywołania `R.lensProp` możesz utworzyć soczewkę, która stanowi nakładkę na właściwość `lastname` z typu `Person`:

```
var person = new Person('Alonzo', 'Church', '444-44-4444');
var lastnameLens = R.lenseProp('lastName');
```

Wywołanie `R.view` pozwala wczytać zawartość tej właściwości:

```
R.view(lastnameLens, person); //-> 'Church'
```

W praktyce to rozwiązanie działa podobnie jak metoda `get lastname()`. Nie ma w tym nic nadzwyczajnego. A co z setterem? Tu ujawniają się możliwości biblioteki. Wywołanie `R.set` tworzy i zwraca nowy egzemplarz obiektu zawierający nową wartość i zachowujący stan pierwotnego obiektu. Otrzymujesz więc „za darmo" mechanizm kopiowania przy zapisie:

```
var newPerson = R.set(lastnameLens, 'Mourning', person);
newPerson.lastname; //-> 'Mourning'
person.lastname; //-> 'Church'
```

Soczewki są przydatne, ponieważ zapewniają niewymagający ingerencji w kod mechanizm operowania obiektami — nawet gdy są to istniejące już obiekty lub obiekty poza kontrolą programisty. Soczewki obsługują też właściwości zagnieżdżone, takie jak właściwość `address` z typu `Person`:

```
person.address = new Address(
   'USA', 'NJ', 'Princeton', zipCode('08544','1234'),
   'Alexander St.');
```

Utwórzmy teraz soczewkę dla właściwości `address.zip`:

```
var zipPath = ['address', 'zip'];
var zipLens = R.lens(R.path(zipPath), R.assocPath(zipPath));
R.view(zipLens, person); //-> zipCode('08544', '1234')
```

Definiuje działanie gettera i settera.

Ponieważ soczewki tworzą settery zachowujące niemodyfikowalność, to po zmodyfikowaniu zagnieżdżonego obiektu można zwrócić nowy obiekt:

```
var newPerson = R.set(zipLens, zipCode('90210', '5678'), person);
var newZip = R.view(zipLens, newPerson); //-> zipCode('90210', '5678')
var originalZip = R.view(zipLens, person);     //-> zipCode('08544', '1234')
newZip.toString() !== originalZip.toString();        //-> true
```

Jest to świetne rozwiązanie, ponieważ otrzymujesz gettery i settery działające w funkcyjny sposób. Soczewki nie tylko zapewniają nakładkę zachowującą niemodyfikowalność, ale też doskonale wpisują się w filozofię programowania funkcyjnego, ponieważ oddzielają logikę dostępu do pól od samego obiektu. Eliminuje to zależność od obiektu `this` i daje przydatne funkcje, które potrafią dotrzeć do zawartości dowolnego obiektu i operować nią.

Gdy już wiesz, jak poprawnie pracować z obiektami, pora zmienić temat i przyjrzeć się funkcjom. To właśnie one sterują pracą aplikacji i są istotą programowania funkcyjnego.

2.3. Funkcje

W programowaniu funkcyjnym to funkcje są podstawową jednostką pracy. To oznacza, że wszystko związane jest właśnie z nimi. **Funkcja** to dowolne wywoływalne wyrażenie, które można przetworzyć za pomocą operatora `()`. Funkcje mogą zwracać do jednostki wywołującej albo obliczoną wartość, albo wynik `undefined` (dotyczy to funkcji bez zwracanej wartości). Ponieważ programowanie funkcyjne działa w matematyczny sposób, funkcje mają znaczenie tylko wtedy, gdy generują *możliwy do użycia wynik* (różny od `null` lub `undefined`). W przeciwnym razie należy założyć, że funkcja modyfikuje zewnętrzne dane i powoduje efekty uboczne. Na potrzeby tej książki rozróżniam *wyrażenia* (funkcje generujące wynik) i *instrukcje* (funkcje, które nie generują wyniku). Platformy imperatywne i proceduralne składają się głównie z uporządkowanych sekwencji instrukcji. Jednak programowanie funkcyjne jest oparte na wyrażeniach, dlatego funkcje bez zwracanej wartości nie są w tym paradygmacie przydatne.

Funkcje w JavaScripcie mają dwie cechy typowe dla stylu funkcyjnego: są pełnoprawnymi obiektami i mogą być funkcjami wyższego poziomu. Dalej szczegółowo omawiam oba te aspekty.

2.3.1. Funkcje jako pełnoprawne obiekty

W JavaScripcie pojęcie **pełnoprawne obiekty** (ang. *first-class citizens*) związane jest z tym, że funkcje działają w nim jak obiekty. Prawdopodobnie przyzwyczaiłeś się do funkcji deklarowanych w następujący sposób:

```
function multiplier(a,b) {
   return a * b;
}
```

Jednak JavaScript udostępnia też inne możliwości. Oto co można zrobić z funkcjami (podobnie jak z obiektami):

- Przypisywać do zmiennych jako funkcje anonimowe lub wyrażenia lambda (szczegółowe omówienie lambd przedstawiam w rozdziale 3.):

```
var square = function (x) {
   return x * x;
}
```

Funkcja anonimowa

```
var square = x => x * x;
```

Wyrażenie lambda

- Przypisywać do właściwości obiektów jako metody:

```
var obj = {
   method: function (x) { return x * x; }
};
```

W wywołaniach funkcji używany jest operator `()`, na przykład `square(2)`, natomiast obiekt reprezentujący funkcję jest wyświetlany w następujący sposób:

```
square;
// function (x) {
//    return x * x;
// }
```

Choć nie jest to powszechnie stosowane rozwiązanie, funkcje można tworzyć także za pomocą konstruktorów. Dowodzi to, że w JavaScripcie funkcje to pełnoprawne obiekty. Konstruktor przyjmuje zbiór parametrów formalnych i ciało funkcji, a wywoływany jest za pomocą słowa kluczowego `new`:

```
var multiplier = new Function('a', 'b', 'return a * b');
multiplier(2, 3); //-> 6
```

W JavaScripcie każda funkcja to obiekt typu `Function`. Właściwość `length` funkcji pozwala pobrać liczbę parametrów formalnych, a metody `apply()` i `call()` można stosować do wywoływania funkcji z użyciem kontekstu. Więcej na temat tych metod dowiesz się z następnego punktu.

Po prawej stronie wyrażenia z funkcją anonimową znajduje się obiekt reprezentujący funkcję z pustą właściwością `name`. Za pomocą funkcji anonimowych przekazywanych jako argument możesz tworzyć rozszerzone lub wyspecjalizowane wersje funkcji. Zastanów się nad natywną funkcją `Array.sort(comparator)` z JavaScriptu. Przyjmuje ona obiekt reprezentujący funkcję porównującą wartości. Domyślnie funkcja `sort` przekształca sortowane wartości na łańcuchy znaków i używa ich wartości z kodowania Unicode jako naturalnego kryterium sortowania. Ma to pewne ograniczenia i często nie jest zgodne z potrzebami programisty. Przyjrzyj się kilku przykładom:

```
var fruit = ['Cebula', 'ananas'];
fruit.sort(); //->['Cebula', 'ananas']
```

W kodowaniu Unicode wielkie litery znajdują się przed małymi.

```
var ages = [1, 10, 21, 2];
ages.sort(); //->[1, 10, 2, 21]
```

Liczby są przekształcane na łańcuchy znaków i porównywane zgodnie z ich wartościami w kodowaniu Unicode.

Tak więc `sort()` to funkcja, której działanie często zależy od kryteriów określonych w funkcji `comparator`. Sama w sobie funkcja `sort()` jest prawie bezużyteczna. Możesz wymusić poprawne porównywanie na podstawie liczb i posortować listę osób według wieku, podając jako argument niestandardową funkcję:

```
people.sort((p1, p2) => p1.getAge() > p2.getAge());
```

Funkcja `comparator` przyjmuje tu dwa parametry, `p1` i `p2`, i działa zgodnie z następującym kontraktem:

- jeśli `comparator` zwraca wartość mniejszą niż 0, `p1` ma znajdować się przed `p2`;
- jeśli `comparator` zwraca wartość 0, należy zachować pozycje `p1` i `p2`;
- jeśli `comparator` zwraca wartość większą niż 0, `p1` ma znajdować się po `p2`.

Oprócz tego, że funkcje można przypisywać do zmiennych, niektóre funkcje JavaScriptu (na przykład `sort()`) przyjmują jako argumenty inne funkcje i należą do grupy tak zwanych *funkcji wyższego poziomu*.

2.3.2. Funkcje wyższego poziomu

Ponieważ funkcje działają jak zwykłe obiekty, zgodne z intuicją jest założenie, że można je przekazywać jako argumenty i zwracać w innych funkcjach. Te „inne funkcje" to **funkcje wyższego poziomu**. Poznałeś już funkcję `comparator` w funkcji `Array.sort()`. Przyjrzyj się teraz pokrótce innym przykładom.

W poniższym przykładzie pokazuję, że funkcje można przekazywać do innych funkcji. Funkcja `applyOperation` przyjmuje dwa argumenty i stosuje do nich funkcję `opt`:

```
function applyOperation(a, b, opt) {
   return opt(a,b);
}

var multiplier = (a, b) => a * b;

applyOperation(2, 3, multiplier); //-> 6
```

Funkcję opt() można przekazywać jako argument do innych funkcji.

W następnym przykładzie funkcja `add` przyjmuje argument i zwraca funkcję, która przyjmuje drugi argument i zwraca sumę obu argumentów:

```
function add(a) {
   return function (b) {
      return a + b;
   }
}
add(3)(3); //-> 6
```

Funkcja jest zwracana przez inną funkcję.

Ponieważ funkcje to pełnoprawne obiekty i można tworzyć funkcje wyższego poziomu, w JavaScripcie funkcje mogą *działać jak wartości*. Z tego wynika, że funkcja to przeznaczona do wykonania wartość zdefiniowana z uwzględnieniem niemodyfikowalności na podstawie przekazywanych do niej danych wejściowych. Ta reguła jest wbudowana w programowanie funkcyjne, a zwłaszcza w łańcuchy funkcji, o czym przekonasz się w rozdziale 3. Gdy tworzysz łańcuchy funkcji, zawsze korzystasz z nazw funkcji do wskazywania fragmentu programu, który zostanie wykonany w ramach całego wyrażenia.

Funkcje wyższego poziomu można łączyć ze sobą, aby tworzyć sensowne wyrażenia na podstawie mniejszych fragmentów kodu i upraszczać programy, których pisanie w innym paradygmacie byłoby żmudne. Oto przykład: załóżmy, że chcesz wyświetlić listę osób mieszkających w Stanach Zjednoczonych. Twój pierwszy pomysł może wyglądać tak jak w pokazanym poniżej kodzie imperatywnym:

```
function printPeopleInTheUs(people) {
   for (let i = 0; i < people.length; i++) {
      var thisPerson = people[i];
      if(thisPerson.address.country === 'USA') {
         console.log(thisPerson);
      }
   }
}
printPeopleInTheUs([p1, p2, p3]);
```

Wywołuje metodę toString każdego obiektu.

p1, p2 i p3 to obiekty typu Person.

Teraz załóżmy, że chcesz dodać obsługę wyświetlania osób mieszkających w innych państwach. Za pomocą funkcji wyższego poziomu możesz przedstawić w abstrakcyjnej postaci operacje wykonywane na każdej osobie (tu jest to wyświetlanie informacji o osobie w konsoli). Możesz swobodnie przekazać dowolną funkcję `action` do funkcji wyższego poziomu `printPeople`:

```
function printPeople(people, action) {
   for (let i = 0; i < people.length; i++) {
      action (people[i]);
```

```
   }
}

var action = function (person) {
   if(person.address.country === 'USA') {
      console.log(person);
   }
}

printPeople(people,action);
```

W językach takich jak JavaScript widoczny jest wzorzec, zgodnie z którym nazwami funkcji mogą być rzeczowniki takie jak `multiplier`, `comparator` i `action`. Ponieważ funkcje są pełnoprawnymi obiektami, można je przypisywać do zmiennych i wywoływać później. Przekształć funkcję `printPeople`, aby w pełni wykorzystać możliwości funkcji wyższego poziomu:

```
function printPeople(people, selector, printer) {
   people.forEach(function (person) {
      if(selector(person)) {
         printer(person);
      }
   });
}

var inUs = person => person.address.country === 'USA';

printPeople(people, inUs, console.log);
```

Wywołanie forEach to preferowany sposób tworzenia pętli zgodny ze stylem funkcyjnym. Omawiam tę kwestię w dalszej części rozdziału.

Dzięki użyciu funkcji wyższego poziomu widoczny staje się deklaratywny charakter kodu. To wyrażenie jednoznacznie określa, co program robi.

Takie nastawienie powinieneś w sobie rozwijać, aby w pełni wykorzystać programowanie funkcyjne. Zaprezentowane ćwiczenie pokazuje, że nowa wersja kodu jest dużo bardziej elastyczna niż początkowe rozwiązanie. Jest tak, ponieważ można szybko zmienić (lub skonfigurować) kryteria wyboru osób, a także określić miejsce, gdzie dane mają trafić. W rozdziałach 3. i 4. koncentruję się na tym zagadnieniu i przedstawiam specjalne biblioteki pozwalające płynnie łączyć operacje w łańcuchy oraz tworzyć złożone programy z prostych komponentów.

W JavaScripcie funkcje są nie tylko wywoływane, ale też stosowane. Warto wspomnieć o tym wyjątkowym aspekcie mechanizmu uruchamiania funkcji w JavaScripcie.

2.3.3. Sposoby uruchamiania funkcji

Mechanizm uruchamiania funkcji w JavaScripcie to ciekawy aspekt tego języka, działający inaczej niż w innych językach. JavaScript daje programiście pełną swobodę w określaniu kontekstu, w którym funkcja ma być uruchamiana. Kontekst ten jest określany za pomocą słowa `this` w ciele funkcji. Funkcje w JavaScripcie można uruchamiać na wiele sposobów:

- *Jako funkcje globalne*. W tym modelu referencja `this` prowadzi do obiektu `globalnego` albo ma wartość `undefined` (w trybie `strict`):

Wybiegając naprzód...

Przerywam na chwilę omawianie podstaw związanych z JavaScriptem, aby dokładniej opisać program z tego punktu i połączyć niektóre pokrótce poruszone zagadnienia. Na razie mogą wydać Ci się one skomplikowane, jednak szybko nauczysz się budować programy w przedstawiony tu sposób za pomocą technik funkcyjnych. Przy użyciu soczewek możesz utworzyć funkcje pomocnicze i uzyskać dzięki nim dostęp do właściwości obiektu:

```
var countryPath = ['address', 'country'];
var countryL = R.lens(R.path(countryPath), R.assocPath(countryPath));
var inCountry = R.curry((country, person) =>
   R.equals(R.view(countryL, person), country));
```

Poniższe wywołanie jest dużo bardziej funkcyjne niż wcześniejsza wersja kodu:

```
people.filter(inCountry('USA')).map(console.log);
```

Nazwa państwa jest tu następnym parametrem, który można dowolnie zmieniać. W dalszych rozdziałach znajdziesz więcej kodu działającego w podobny sposób.

```
function doWork() {
   this.myVar = 'Jakaś wartość';
}
doWork();
```

Wywołanie doWork() na poziomie globalnym powoduje, że referencja this prowadzi do obiektu globalnego.

- *Jako metody*. Referencja `this` jest ustawiana na jednostkę, do której metoda należy. Jest to ważny aspekt obiektowej natury JavaScriptu:

```
var obj = {
   prop: 'Jakaś właściwość',
      getProp: function () {return this.prop}
};
obj.getProp();
```

Wywołanie metody obiektu sprawia, że this prowadzi do tego obiektu.

- *Jako konstruktor (wywołanie jest wtedy poprzedzone słowem* `new`*)*. Takie wywołanie niejawnie zwraca referencję do nowo utworzonego obiektu:

```
function MyType(arg) {
   this.prop = arg;
}

var someVal = new MyType('Jakiś argument');
```

Wywołanie funkcji z użyciem słowa new sprawia, że this prowadzi do tworzonego i niejawnie zwracanego obiektu.

Te przykłady pokazują, że (inaczej niż w innych językach programowania) obiekt wskazywany przez `this` zależy od sposobu uruchomienia funkcji (globalnie, jako metody obiektu, jako konstruktora itd.), a nie od jej kontekstu leksykalnego (czyli od miejsca użycia w kodzie). Może to prowadzić do trudnego do zrozumienia kodu, ponieważ trzeba zwracać baczną uwagę na kontekst wykonywania funkcji.

Dodałem ten fragment, ponieważ programiści używający JavaScriptu powinni znać ten materiał. Jednak w kodzie funkcyjnym, o czym już kilkakrotnie wspomniałem, referencja `this` jest spotykana rzadko (należy jej unikać za wszelką cenę). Natomiast często posługują się nią autorzy bibliotek i narzędzi w specjalnych sytuacjach, które wymagają modyfikacji kontekstu języka w celu wykonania zaskakujących sztuczek. Te sztuczki nieraz obejmują wywołania używanych dla funkcji metod `apply` i `call`.

2.3.4. Metody używane dla funkcji

JavaScript umożliwia uruchamianie funkcji za pomocą przeznaczonych dla nich metod `call` i `apply` (możesz traktować je jak metafunkcje), należących do prototypu funkcji. Obie te metody są często używane w kodzie rusztowania (ang. *scaffolding*), co pozwala użytkownikom interfejsu API tworzyć nowe funkcje na podstawie istniejących. Zobacz, jak można napisać funkcję `negate`:

Tworzenie funkcji wyższego poziomu o nazwie negate. Przyjmuje ona funkcję wejściową i zwraca funkcję negującą wynik tej pierwszej.

Użycie wywołania Function.apply() w celu uruchomienia funkcji wejściowej z pierwotnymi argumentami.

```
function negate(func) {
   return function() {
      return !func.apply(null, arguments);
   };
}
```

Definicja funkcji isNull.

```
function isNull(val) {
   return val === null;
}
```

Definicja funkcji isNotNull jako negacji funkcji isNull.

```
var isNotNull = negate(isNull);

isNotNull(null); //-> false
isNotNull({});   //-> true
```

Funkcja `negate` tworzy nową funkcję, która wykonuje funkcję wejściową z jej argumentami i neguje jej wynik. W tym przykładzie używana jest metoda `apply`, w ten sam sposób możesz jednak zastosować także metodę `call`. Różnica polega na tym, że `call` przyjmuje listę argumentów, natomiast `apply` — tablicę argumentów. Pierwszy argument, `thisArg`, pozwala w razie potrzeby zmodyfikować kontekst funkcji. Oto sygnatury obu omawianych metod:

```
Function.prototype.apply(thisArg, [argsArray])
Function.prototype.call(thisArg, arg1,arg2,...)
```

Jeśli `thisArg` prowadzi do obiektu, jest nim obiekt, dla którego wywoływana jest dana metoda. Jeżeli `thisArg` to `null`, kontekstem funkcji jest obiekt `globalny`, a funkcja działa jak prosta funkcja globalna. Gdy jednak używany jest tryb `strict`, przekazywana jest sama wartość `null`.

Modyfikowanie kontekstu funkcji za pomocą argumentu `thisArg` pozwala stosować wiele różnych technik. Jednak w programowaniu funkcyjnym używanie tego argumentu nie jest zalecane, ponieważ nie należy polegać na stanie kontekstu (pamiętaj, że wszystkie dane są przekazywane do funkcji jako argumenty). Z tego powodu nie omawiam więcej tego mechanizmu.

Choć współużytkowany kontekst globalny i kontekst obiektu nie odgrywają w funkcyjnym JavaScripcie dużego znaczenia, jeden kontekst jest istotny. Chodzi tu o kontekst funkcji. Aby go zrozumieć, trzeba zapoznać się z domknięciami i zasięgiem.

2.4. Domknięcia i zasięg

Przed pojawieniem się JavaScriptu domknięcia występowały tylko w językach funkcyjnych używanych w pewnych specyficznych zastosowaniach. W JavaScripcie po raz pierwszy pojawiły się one w standardowych zastosowaniach i znacznie zmieniły sposób pisania kodu. Wróćmy do typu `zipCode`:

```
function zipCode(code, location) {
   let _code = code;
   let _location = location || '';

   return {
      code: function () {
         return _code;
      },
      location: function () {
         return _location;
      },
   ...
   };
}
```

Gdy przeanalizujesz ten kod, zauważysz, że funkcja `zipCode` zwraca literał obiektowy, który najwyraźniej ma pełny dostęp do zmiennych zadeklarowanych poza jego zasięgiem. Oznacza to, że po zakończeniu pracy przez funkcję `zipCode` wynikowy obiekt nadal ma dostęp do zadeklarowanych w niej informacji:

```
const princetonZip = zipCode('08544', '3345');
princetonZip.code(); //-> '08544'
```

Trudno jest to zrozumieć. Technika ta działa dzięki domknięciu tworzonemu w JavaScripcie wokół deklaracji obiektu i funkcji. Możliwość dostępu do danych w ten sposób ma wiele praktycznych zastosowań. W tym podrozdziale pokazuję, jak za pomocą domknięć zasymulować tworzenie zmiennych prywatnych, pobierać dane z serwera i tworzyć zmienne o zasięgu ograniczonym do danego bloku.

Domknięcie to struktura danych, która wiąże funkcję z jej środowiskiem w momencie jej deklaracji. Technika ta oparta jest na lokalizacji tekstu deklaracji funkcji. Dlatego domknięcie można nazwać *statycznym* lub *leksykalnym zasięgiem* otaczającym definicję funkcji. Ponieważ zapewnia to funkcji dostęp do otaczającego ją stanu, kod jest przejrzysty i czytelny. Wkrótce zobaczysz, że domknięcia są bardzo istotne nie tylko w programach funkcyjnych (gdzie używane są funkcje wyższego poziomu), ale też do obsługi zdarzeń i wywołań zwrotnych, symulowania zmiennych prywatnych i radzenia sobie z niektórymi pułapkami z JavaScriptu.

Reguły rządzące działaniem domknięć są ściśle związane z regułami określania zasięgu w JavaScripcie. Zasięg grupuje zbiór wiązań zmiennych i definiuje fragment kodu, w którym dana zmienna jest zdefiniowana. Domknięcie to wynik dziedziczenia zasięgu przez funkcję. Podobnie metody obiektu mają dostęp do odziedziczonych zmiennych instancji. W obu przypadkach używane są referencje do jednostki nadrzędnej. Domknięcia często występują przy stosowaniu funkcji zagnieżdżonych. Oto krótki przykład:

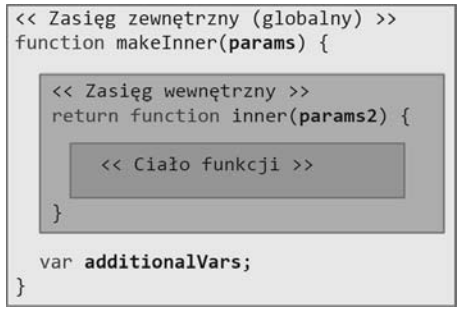
```
function makeAddFunction(amount) {
   function add(number) {
      return number + amount;
   }
   return add;
}

function makeExponentialFunction(base) {
   function raise (exponent) {
      return Math.pow(base, exponent);
   }
   return raise;
}
var addTenTo = makeAddFunction(10);
addTenTo(10); //-> 20

var raiseThreeTo = makeExponentialFunction(3);
raiseThreeTo(2); //-> 9
```

Funkcja add() jest leksykalnie wiązana w funkcji makeAddFunction i ma dostęp do zmiennej amount.

Funkcja raise() jest leksykalnie wiązana w funkcji makeExponentialFunction i ma dostęp do zmiennej base.

W tym przykładzie warto zauważyć, że choć zmienne `amount` i `base` w obu funkcjach nie znajdują się w aktywnym zasięgu, są dostępne w zwracanych funkcjach. Wyobraź sobie, że zagnieżdżone funkcje `add` i `raise` obejmują nie tylko obliczenia, ale też zapis wszystkich otaczających je zmiennych. W bardziej ogólnym ujęciu (pokazanym na rysunku 2.4) domknięcie funkcji obejmuje:

- wszystkie parametry funkcji (`params` i `params2` na rysunku),
- wszystkie zmienne z zewnętrznego zasięgu (w tym oczywiście wszystkie zmienne lokalne), a także zmienne zadeklarowane po funkcji (na przykład zmienną `addi↪tionalVars`).



Rysunek 2.4. Domknięcie obejmuje zmienne z zewnętrznego (globalnego) zasięgu i wewnętrznego zasięgu funkcji nadrzędnej, a także parametry funkcji nadrzędnej i dodatkowe zmienne zadeklarowane po samej funkcji. Kod z ciała funkcji ma dostęp do zmiennych i obiektów zdefiniowanych we wszystkich tych zasięgach. Zasięg globalny jest współużytkowany przez wszystkie funkcje

Na listingu 2.3 pokazane jest działanie domknięć.

Listing 2.3. Działanie domknięć

```
var outerVar = 'Zewnętrzna';
function makeInner(params) {
   var innerVar = 'Wewnętrzna';

   function inner() {
      console.log(
         `Dostępne dane: ${outerVar}, ${innerVar}, ${params}`);
```

Deklaracja zmiennej globalnej outerVar.

Deklaracja zmiennej lokalnej makeInner.

Deklaracja funkcji inner. Zmienne innerVar i outerVar należą do jej domknięcia.

```
    }
    return inner;
}

var inner = makeInner('Parametry');
inner();
```

Wywołanie funkcji makeInner w celu zwrócenia funkcji inner.

Tu inner to prawidłowa funkcja, która istnieje także po wykonaniu jej funkcji zewnętrznej.

Ten kod po uruchomieniu wyświetli następujący tekst:

```
Dostępne dane: Zewnętrzna, Wewnętrzna, Parametry
```

Na pozór jest to niezgodne z intuicją i zagadkowe. Mógłbyś oczekiwać, że zmienne lokalne (tu jest nią `innerVar`) powinny przestać istnieć i zostać usunięte przez mechanizm odzyskiwania pamięci po zwróceniu sterowania przez funkcję `makeInner`. Wtedy kod powinien wyświetlić wartość `undefined`. Jednak na zapleczu działa magia domknięć. Funkcja zwracana przez funkcję `makeInner` zachowuje wszystkie zmienne dostępne w zasięgu w momencie jej deklarowania, a także chroni je przed usunięciem. Częścią domknięcia jest też zasięg globalny, dlatego dostępna jest zmienna `outerVar`. Do domknięć i tego, co znajduje się w kontekście funkcji, wracam w rozdziale 7.

Może się zastanawiasz, w jaki sposób zmienne (na przykład `additionalVars`) zadeklarowane po funkcji mogą stać się częścią jej domknięcia. Aby to zrozumieć, należy uwzględnić fakt, że w JavaScripcie występują trzy rodzaje zasięgu: zasięg globalny, zasięg funkcji i zasięg pseudobloku.

2.4.1. Problemy z zasięgiem globalnym

Zasięg globalny to najprostszy rodzaj zasięgu, ale też najbardziej problematyczny. W JavaScripcie wszystkie obiekty i zmienne zadeklarowane na najwyższym poziomie skryptu (nieznajdujące się w żadnej funkcji) należą do *zasięgu globalnego* i są dostępne w całym kodzie. Pamiętaj, że w programowaniu funkcyjnym celem jest zapobieganie wykraczaniu obserwowalnych zmian poza funkcje. Jednak gdy używany jest zasięg globalny, każdy wykonany wiersz może spowodować widoczne zmiany.

Stosowanie zmiennych globalnych jest kuszące, jednak są one współużytkowane przez wszystkie skrypty wczytane dla strony. Może to prowadzić do konfliktów nazw, jeśli kod w JavaScripcie nie jest umieszczony w modułach. Zaśmiecanie globalnej przestrzeni nazw może sprawiać problemy, ponieważ występuje ryzyko zastąpienia zmiennych i funkcji zadeklarowanych w innych plikach.

Dane globalne utrudniają analizowanie kodu, ponieważ wymagają pamiętania o stanie wszystkich zmiennych z danego punktu w czasie. Jest to jeden z głównych powodów, dla których wraz z rozrastaniem się kodu rośnie też złożoność programu. Ponadto stosowanie danych globalych ma też skutki uboczne, ponieważ odczyt lub zapis takich danych tworzy zewnętrzne zależności. Dlatego na tym etapie powinno być już oczywiste, że gdy chcesz pisać kod w stylu funkcyjnym, powinieneś za wszelką cenę unikać zmiennych globalnych.

2.4.2. Zasięg funkcji w JavaScripcie

Jest to zalecany zasięg do stosowania w JavaScripcie. Wszystkie zmienne zadeklarowane w funkcji są w niej lokalne i nie są widoczne poza nią. Ponadto gdy funkcja zwraca sterowanie, wszystkie zadeklarowane w niej zmienne lokalne są usuwane razem z nią. Przyjrzyj się następującej funkcji:

```
function doWork() {
   let student = new Student(...);
   let address = new Address(...);
   // Inne operacje.
};
```

Zmienne `student` i `address` są wiązane w funkcji `doWork()` i niedostępne dla zewnętrznego kodu. Na rysunku 2.5 widać, że interpretowanie zmiennej za pomocą nazwy przypomina opisane wcześniej interpretowanie nazw w łańcuchu prototypów. Najpierw sprawdzany jest najbardziej wewnętrzny zasięg, a później bardziej zewnętrzne zasięgi. W JavaScripcie określanie zasięgu działa w następujący sposób:

1. Najpierw sprawdzany jest zasięg funkcji, w której zmienna jest używana.
2. Jeśli zmienna nie znajduje się w zasięgu lokalnym, kod szuka zmiennej w kolejnych zewnętrznych zasięgach leksykalnych, aż w końcu dociera do zasięgu globalnego.
3. Jeżeli nie udało się znaleźć zmiennej, JavaScript zwraca wartość `undefined`.

Przyjrzyj się przykładowemu kodowi:

```
var x = 'Jakaś wartość';
function parentFunction() {
   function innerFunction() {
      console.log(x);
   }
   return innerFunction;
}
var inner = parentFunction();
inner();
```

W momencie wywołania funkcji `inner` środowisko uruchomieniowe JavaScriptu rozpoczyna wyszukiwanie zmiennej `x` w sposób pokazany na rysunku 2.5.

Rysunek 2.5. Tak wygląda kolejność interpretowania nazw w JavaScripcie. Proces przebiega od zasięgu otaczającego zmienną do zewnątrz. Najpierw sprawdzany jest zasięg funkcji (lokalny), potem zasięg jednostki nadrzędnej (jeśli taka istnieje), a w ostatnim kroku zasięg globalny. Jeśli zmiennej x nie uda się znaleźć, funkcja zwraca wartość undefined

Jeśli masz doświadczenie w posługiwaniu się innymi językami programowania, prawdopodobnie jesteś przyzwyczajony do używania zasięgu funkcji. Jednak ponieważ składnia JavaScriptu jest podobna do składni C, możesz sądzić, że zasięg bloku działa podobnie do zasięgu funkcji.

2.4.3. Zasięg pseudobloku

Niestety, standardowa wersja ES5 JavaScriptu nie obsługuje zasięgu bloku (bloki tworzone są za pomocą nawiasów klamrowych `{}` w strukturach sterujących, takich jak `for`, `while`, `if` i `switch`). Wyjątkiem jest zmienna błędu przekazywana do bloku `catch`. Instrukcja `with` pozwala uzyskać pewnego rodzaju zasięg bloku, jednak nie zaleca się jej stosowania i w trybie `strict` jest ona usuwana. W innych językach podobnych do C poza blokiem kodu nie można uzyskać dostępu do zmiennej zadeklarowanej w instrukcji `if` (tu jest to zmienna `myVar`):

```
if (someCondition) {
   var myVar = 10;
}
```

Dla programistów, którzy są przyzwyczajeni do tego rozwiązania i dopiero poznają JavaScript, może być to mylące. Ponieważ w JavaScripcie używany jest zasięg funkcji, zmienne zadeklarowane w bloku są dostępne w dowolnym miejscu danej funkcji. Może to znacznie utrudniać pracę programistom używającym JavaScriptu, jednak istnieją rozwiązania omawianego problemu. Przyjrzyj się konkretnej sytuacji:

```
function doWork() {
   if (!myVar) {
      var myVar = 10;
   }
   console.log(myVar); //-> 10
}
doWork();
```

Zmienna `myVar` jest deklarowana w instrukcji `if`, ale pozostaje widoczna poza blokiem. Co dziwne, uruchomienie tego kodu powoduje wyświetlenie wartości `10`. Może to być zaskakujące — zwłaszcza dla programistów przyzwyczajonych do częściej używanego zasięgu bloku. Wewnętrzny mechanizm JavaScriptu umieszcza deklaracje zmiennych i funkcji na początku bieżącego zasięgu, którym tu jest zasięg funkcji. Dlatego używanie pętli może być ryzykowne. Przyjrzyj się listingowi 2.4.

Listing 2.4. Problem z licznikiem pętli wynikający z niejednoznaczności zmiennej

```
var arr = [1, 2, 3, 4];
function processArr() {

   function multipleBy10(val) {
      i = 10;
      return val * i;
   }

   for(var i = 0; i < arr.length; i++) {
      arr[i] = multipleBy10(arr[i]);
```

```
    }
    return arr;
}
processArr(); //-> [10, 2, 3, 4]
```

Licznik pętli, `i`, jest przenoszony na początek funkcji i należy do domknięcia funkcji `mul`
`tipleBy10`. Pominięcie słowa kluczowego `var` w deklaracji `i` sprawia, że w funkcji `multiplyBy` nie jest tworzona zmienna o zasięgu lokalnym, przez co zmienna licznika przyjmuje wartość `10`. Deklaracja licznika pętli jest przenoszona, licznik zostaje ustawiony na wartość `undefined`, a po uruchomieniu pętli przypisywana jest do niego wartość `0`. W rozdziale 8. zetkniesz się z problemem niejednoznaczności w kontekście nieblokujących operacji w pętlach.

Dobre środowiska IDE i lintery pozwalają złagodzić tego typu problemy, jednak nawet takie narzędzia nie pomogą, gdy program obejmuje setki wierszy kodu. W następnym rozdziale przedstawiam lepsze rozwiązania, które są bardziej eleganckie i mniej narażone na błędy niż standardowe pętle. Te techniki w pełni wykorzystują możliwości dawane przez funkcje wyższego poziomu i pomagają rozwiązać opisywane trudności. W tym rozdziale zobaczyłeś już, że wersja ES6 JavaScriptu udostępnia słowo kluczowe `let`, które pomaga rozwiązać niejednoznaczność zmiennej licznika pętli dzięki prawidłowemu powiązaniu licznika z zawierającym go blokiem:

```
for(let i = 0; i < arr.length; i++) {
    //...
}
```

Słowo let rozwiązuje problem przenoszenia zmiennej i zapewnia odpowiedni zasięg zmiennej i. Poza pętlą zmienna i nie jest zdefiniowana.

```
i; // i === undefined
```

Jest to krok we właściwym kierunku i powód, dla którego preferuję używanie `let` zamiast `var` dla zmiennych o ograniczonym zasięgu. Jednak ręcznie tworzone pętle mają też inne wady, których rozwiązanie przedstawiam w następnym rozdziale. Skoro już rozumiesz, co domknięcie funkcji zawiera i jak współdziała ono z zasięgiem, pora przyjrzeć się praktycznym zastosowaniom domknięć.

2.4.4. Praktyczne zastosowania domknięć

Jest wiele praktycznych zastosowań domknięć, ważnych w trakcie pisania rozbudowanych programów w JavaScripcie. Te zastosowania nie ograniczają się do programowania funkcyjnego, ale wykorzystują sposób działania funkcji w JavaScripcie. Oto te zastosowania:

- symulowanie zmiennych prywatnych,
- zgłaszanie asynchronicznych wywołań po stronie serwera,
- tworzenie zmiennych o zasięgu sztucznie ograniczonym do bloku.

SYMULOWANIE ZMIENNYCH PRYWATNYCH

Wiele języków innych niż JavaScript udostępnia wbudowany mechanizm definiowania wewnętrznych właściwości obiektów z użyciem modyfikatorów dostępu (na przykład `private`). W JavaScripcie nie występuje natywne słowo kluczowe służące do tworzenia prywatnych zmiennych i funkcji dostępnych tylko w zasięgu danego obiektu.

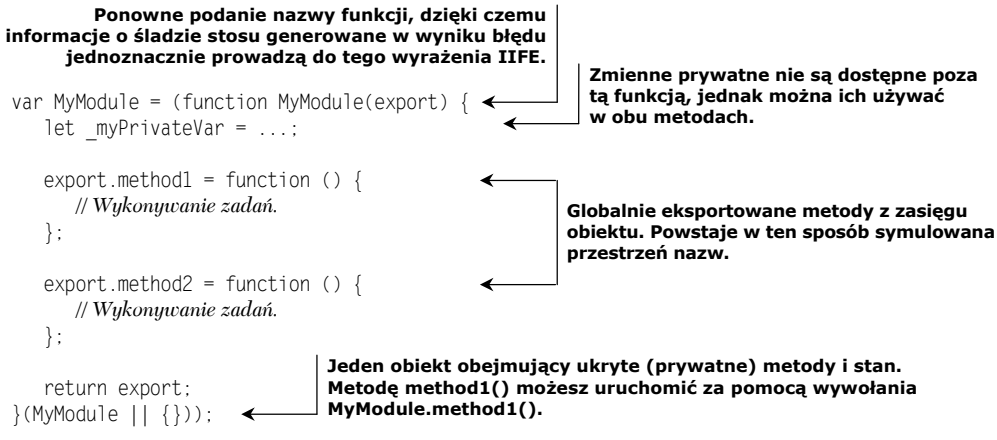
Hermetyzacja może ułatwiać zachowanie niemodyfikowalności, ponieważ nie można zmienić czegoś, co jest niedostępne.

Jednak za pomocą domknięć można zasymulować zmienne prywatne. Jednym z przykładów jest zwracanie obiektu, tak jak w przedstawionych wcześniej funkcjach `zipCode` i `coordinate`. Te funkcje zwracają literały obiektowe z metodami, które mają dostęp do zmiennych lokalnych funkcji zewnętrznej, ale nie udostępniają tych zmiennych. Dlatego zmienne te działają jak zmienne prywatne.

Domknięcia umożliwiają też zarządzanie globalną przestrzenią nazw w celu uniknięcia globalnie współużytkowanych danych. Autorzy bibliotek i modułów wykorzystują możliwości domknięć w nowy sposób i ukrywają przy ich użyciu prywatne metody oraz dane całych modułów. Stosują do tego **wzorzec modułów** (ang. *module pattern*) oparty na użyciu jednego **natychmiast wykonywanego wyrażenia funkcyjnego** (ang. *immediately invoked function expression* — IIFE) w celu ukrycia wewnętrznych zmiennych przy zachowaniu możliwości udostępnienia niezbędnego zestawu mechanizmów w zewnętrznym kodzie. Pozwala to znacznie ograniczyć liczbę globalnych referencji.

> **UWAGA** Zalecamy stosowanie dobrej praktyki polegającej na umieszczaniu całego kodu funkcyjnego w modułach z odpowiednią hermetyzacją. Wszystkie podstawowe zasady programowania funkcyjnego, jakie poznałeś w tej książce, możesz wykorzystać na poziomie modułów.

Oto krótki przykładowy szkielet modułu[3]:



Obiekt `MyModule` jest tworzony globalnie, przekazywany do wyrażenia reprezentującego funkcję (tworzonego za pomocą słowa kluczowego `function`) i natychmiast wykonywany w momencie wczytywania skryptu. Ponieważ w JavaScripcie obowiązuje zasięg funkcji, `_myPrivateVar` i inne zmienne prywatne są zmiennymi lokalnymi funkcji, w której się znajdują. Domknięcie otaczające dwie eksportowane metody umożliwia obiektowi

[3] Dużo dokładniejsze omówienie różnych rodzajów wzorców modułów znajdziesz w tekście Bena Cherry'ego „JavaScript Module Pattern: In-Depth", *Adequately Good*, 12 marca 2010, *http://mng.bz/H9hk*.

bezpieczny dostęp do wszystkich wewnętrznych właściwości modułu. Jest to atrakcyjne rozwiązanie, ponieważ pozwala ograniczyć liczbę globalnych elementów i udostępniać obiekt z hermetycznymi operacjami i stanem. Ten wzorzec modułów jest stosowany we wszystkich funkcyjnych bibliotekach używanych w tej książce.

ASYNCHRONICZNE WYWOŁANIA PO STRONIE SERWERA

Ponieważ w JavaScripcie funkcje to pełnoprawne obiekty, które mogą działać jako funkcje wyższego poziomu, można przekazywać je do innych funkcji jako wywołania zwrotne. Wywołania zwrotne są przydatne do obsługi zdarzeń bez ingerowania w pierwotny kod. Załóżmy, że chcesz zgłaszać żądanie do serwera i otrzymywać powiadomienie po otrzymaniu odpowiedzi. Tradycyjne rozwiązanie polega na podaniu wywoływanej zwrotnie funkcji, która obsłuży odpowiedź:

```
getJSON('/students',
   (students) => {
      getJSON('/students/grades',
         grades => processGrades(grades),
         error => console.log(error.message));
   },
   (error) =>
      console.log(error.message)
)
```

Przetwarzanie obu odpowiedzi.

Obsługuje błędy związane z pobieraniem ocen.

Obsługuje błędy związane z pobieraniem danych studenta.

W tym kodzie `getJSON` to funkcja wyższego poziomu, która przyjmuje jako argumenty dwie wywoływane zwrotnie funkcje: dla powodzenia i dla błędu. Standardowy wzorzec stosowany w asynchronicznym kodzie i do obsługi zdarzeń może prowadzić do głęboko zagnieżdżonych wywołań funkcji. Skutkuje to „przeklętą piramidą wywołań zwrotnych", gdy konieczna jest seria kolejnych zdalnych wywołań kierowanych do serwera. Prawdopodobnie doświadczyłeś już tego, że głęboko zagnieżdżony kod trudno jest zrozumieć. W rozdziale 8. poznasz najlepsze techniki pozwalające „spłaszczyć" kod do postaci bardziej płynnych i deklaratywnych wyrażeń, które można łączyć w łańcuch zamiast zagnieżdżać.

SYMULOWANIE ZMIENNYCH O ZASIĘGU BLOKU

Za pomocą domknięć można utworzyć inne rozwiązanie przedstawionego na listingu 2.4 problemu niejednoznacznych zmiennych licznika pętli. Problem wynika z braku obsługi zasięgu bloku w JavaScripcie, dlatego celem jest sztuczne utworzenie takiego zasięgu. Jak to zrobić? Użycie słowa kluczowego `let` rozwiązuje wiele kłopotów związanych z tradycyjnymi pętlami, jednak funkcyjne podejście polega na wykorzystaniu domknięć, obowiązującego w JavaScripcie zasięgu funkcji oraz konstruktu `forEach`. Zamiast martwić się o wiązanie licznika pętli i innych zmiennych w odpowiednim zasięgu, możesz umieścić kod pętli w funkcji, co pozwala zasymulować blok o zasięgu funkcji w instrukcji tworzącej pętlę. Dalej dowiesz się, że pomaga to w wywoływaniu asynchronicznych operacji w trakcie iteracyjnego przetwarzania elementów kolekcji:

```
arr.forEach(function(elem, i) {
   ...
});
```

W tym rozdziale przedstawiłem tylko podstawy JavaScriptu, aby pomóc Ci zrozumieć niektóre ograniczenia tego języka odczuwalne, gdy chcesz stosować go w modelu funkcyjnym. Ten rozdział miał też przygotować Cię do poznania technik funkcyjnych omawianych w dalszych rozdziałach. Jeśli chcesz dowiedzieć się więcej o JavaScripcie, dostępne są całe książki, w których obiekty, dziedziczenie i domknięcia są opisane bardziej szczegółowo.

Chcesz zostać ninją JavaScriptu?

Omawiane w tym rozdziale zagadnienia dotyczące obiektów, funkcji, zasięgu i domknięć są bardzo istotne, jeśli chcesz stać się ekspertem od JavaScriptu. Tu jednak przedstawiłem je tylko pobieżnie, aby zapewnić Ci informacje niezbędne do skoncentrowania się w dalszych rozdziałach na programowaniu funkcyjnym. Jeśli chcesz dowiedzieć się więcej i rozwinąć znajomość JavaScriptu do poziomu ninji, zachęcam do lektury książki *Secrets of the JavaScript Ninja*, wydanie drugie, autorstwa Johna Resiga, Beara Bibeaulta i Josipa Marasa (Manning, 2016, *http://www.manning.com/books/secrets-of-the-javascript-ninja-second-edition*).

Masz już solidne podstawy z zakresu JavaScriptu. W następnym rozdziale przyjrzysz się przetwarzaniu danych z użyciem popularnych operacji takich jak `map`, `reduce` i `filter` oraz rekurencji.

2.5. Podsumowanie

- JavaScript to wszechstronny język przeznaczony głównie do programowania obiektowego i funkcyjnego.
- Dodanie niemodyfikowalności do modelu obiektowego pozwala wygodnie łączyć to podejście z programowaniem funkcyjnym.
- W JavaScripcie funkcje są pełnoprawnymi obiektami i mogą być tworzone jako funkcje wyższego poziomu. Funkcje stanowią podstawę umożliwiającą programowanie funkcyjne w JavaScripcie.
- Domknięcia mają wiele praktycznych zastosowań. Umożliwiają ukrywanie danych, tworzenie modułów i przekazywanie sparametryzowanych operacji do ogólnych funkcji działających dla wielu typów danych.

Skorowidz

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

W

Z

Każdy paradygmat programowania zakłada inne podejście do rozwiązywania problemów. Mimo że podejście obiektowe wciąż jest podstawowym modelem projektowania programowania, podejście funkcyjne pozwala na uzyskanie kodu lepszej jakości: modularnego, ekspresywnego, odpornego na błędy, a przy tym zrozumiałego i łatwego w testowaniu. Szczególnie interesujące jest stosowanie w modelu funkcyjnym języka JavaScript. Chociaż jest to język obiektowy, okazuje się, że taki sposób programowania pozwala na uzyskiwanie wyjątkowo efektywnego i elastycznego kodu.

Niniejsza książka jest przeznaczona dla programistów, którzy chcą się nauczyć programowania funkcyjnego w JavaScripcie. Przedstawiono tu zarówno teoretyczne aspekty tego paradygmatu, jak i konkretne mechanizmy: funkcje wyższego poziomu, domknięcia, rozwijanie funkcji, kompozycje. Nieco trudniejszymi zagadnieniami, które tu omówiono, są monady i programowanie reaktywne. Ten poradnik pozwala też zrozumieć zasady tworzenia asynchronicznego kodu sterowanego zdarzeniami i w pełni wykorzystać możliwości JavaScriptu.

W książce omówiono:

- techniki programowania funkcyjnego w JavaScripcie
- stosowanie łańcuchów funkcji oraz korzystanie z rekurencji
- techniki rozwijania i kompozycji funkcji oraz modularność kodu
- testowanie aplikacji oparte na właściwościach
- model pamięci w JavaScripcie
- zasady programowania reaktywnego i bibliotekę RxJS

Luis Atencio — jest inżynierem oprogramowania. Zajmuje się tworzeniem architektury aplikacji dla różnych przedsiębiorstw. Tworzy kod w JavaScripcie, Javie i PHP. Obdarzony dużym talentem do przekazywania wiedzy, bardzo często dzieli się swoimi doświadczeniami podczas konferencji branżowych. Prowadzi bloga na temat inżynierii oprogramowania i pisze artykuły dla rozmaitych magazynów oraz serwisu DZone.

Programowanie funkcyjne — i kod staje się lepszy!

Helion

księgarnia internetowa
http://helion.pl

zamówienia telefoniczne
0 801 339900
0 601 339900

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

ISBN 978-83-283-3252-2

Informatyka w najlepszym wydaniu

cena: 49,00 zł