JavaScript. Programowanie obiektowe

Autor: Stoyan Stefanov
Tłumaczenie: Justyna Walkowska
ISBN: 978-83-246-2242-9
Tytuł oryginału: Object-Oriented JavaScript
Format: B5, stron: 336

Poznaj obiektowe możliwości JavaScript!

- Jak rozpocząć przygodę z językiem JavaScript?
- Jak rozszerzać obiekty wbudowane?
- Jak pracować w środowisku przeglądarki?

JavaScript jest obiektowym, skryptowym językiem programowania. Choć swą błyskotliwą karierę język ten rozpoczął ponad dwanaście lat temu, swoimi możliwościami wciąż potrafi zaskoczyć nawet doświadczonego programistę. Ostatnio – dzięki technologii AJAX – znów osiągnął on swą szczytową formę. Wykorzystując w odpowiedni sposób jego właściwości, sprawisz, że twój serwis WWW stanie się bardziej interaktywny i dynamiczny.

Dzięki tej książce dowiesz się, w jaki sposób użyć do swoich celów obiektowych możliwości języka JavaScript. Jednak zanim zapoznasz się z tymi tematami, autor w niezwykle przejrzysty sposób przedstawi Ci podstawy tego języka. Zobaczysz, w jaki sposób działają funkcje, pętle oraz model DOM. Ponadto nauczysz się korzystać ze wzorców projektowych, wyrażeń regularnych oraz prototypów. Pomimo zaawansowanej tematyki poruszanej przez autora tej książki dzięki przejrzystemu językowi i klarownemu układowi stanowi ona świetną lekturę również dla początkujących programistów.

- Pojęcia związane z programowaniem obiektowym
- Typy danych, tablice, pętle, sterowanie wykonaniem
- Wykorzystanie funkcji
- Domknięcia
- Obiekty wbudowane
- Zastosowanie konstruktorów
- Tablice asocjacyjne
- Użycie prototypów
- Rozszerzanie obiektów wbudowanych
- Dziedziczenie
- Praca w środowisku przeglądarki (modele BOM i DOM)
- Wzorce kodowania i wzorce projektowe

Od podstaw do sprawnego programowania obiektowego!

Helion SA
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

3

ROZDZIAŁ

Funkcje

Opanowanie funkcji ma kluczowe znaczenie podczas nauki każdego języka programowania, a w przypadku JavaScriptu jest jeszcze ważniejsze niż zwykle. Jest tak dlatego, że w tym języku funkcje mają bardzo wiele zastosowań i w dużej mierze to dzięki nim JavaScript jest tak elastyczny i ekspresywny. W miejscach, gdzie w innych językach programowania trzeba by było stosować specjalną składnię w celu wykorzystania obiektowości, JavaScript udostępnia funkcje. Ten rozdział omawia:

- definiowanie funkcji i korzystanie z nich,
- przekazywanie funkcjom parametrów,
- funkcje predefiniowane dostępne za darmo,
- zasięg zmiennych,
- podejście, zgodnie z którym funkcje to tylko dane specjalnego typu.

Zrozumienie powyższych tematów da nam solidne oparcie przed przejściem do kolejnej części rozdziału, w której przedstawione zostaną pewne ciekawe zastosowania funkcji:

- funkcje anonimowe;
- wywołania zwrotne;
- samowywołujące się funkcje;
- funkcje wewnętrzne (zdefiniowane wewnątrz innych funkcji);
- funkcje, które zwracają inne funkcje;
- funkcje, które zmieniają swoją definicję;
- domknięcia.

Czym jest funkcja?

Funkcje pozwalają zgrupować pewną ilość kodu, nadać jej nazwę, a następnie ponownie wykorzystać przy użyciu tej właśnie nazwy. Spójrzmy na przykład:

```
function sum(a, b) {
   var c = a + b;
return c;
}
```

Z jakich części składa się funkcja?

- Słowo kluczowe `function`.
- *Nazwa* funkcji, w przykładzie jest to `sum`.
- Oczekiwane parametry (argumenty), w tym wypadku `a` i `b`. Funkcja może mieć ich zero lub więcej. Jeśli jest ich więcej niż jeden, parametry rozdziela się przecinkami.
- Blok kodu, nazywany *ciałem* funkcji.
- Instrukcja `return`, która umożliwia zwrócenie obliczonej wartości funkcji. Funkcja zawsze zwraca wartość. Jeśli nie robi tego w sposób jawny, niejawnie zwraca wartość `undefined`.

Zwróć uwagę, że funkcja może zwrócić tylko jedną wartość. Jeśli potrzebne jest zwrócenie większej liczby wartości, należy umieścić je w tablicy i zwrócić tablicę jako wartość funkcji.

Wywoływanie funkcji

Aby skorzystać z funkcji, należy ją wywołać. Funkcję wywołuje się poprzez podanie jej nazwy i argumentów umieszczonych w nawiasie.

Wywołajmy zatem funkcję `sum()`, przekazując jej dwa argumenty i przypisując zwracaną przez nią wartość zmiennej `result`.

```
>>> var result = sum(1, 2);
>>> result;
```

```
3
```

Parametry

Podczas definiowania funkcji można określić oczekiwane parametry. Funkcja nie musi pobierać parametrów, ale jeśli oczekuje, że je otrzyma, a programista podczas wywoływania funkcji zapomni o ich podaniu, JavaScript przypisze im wartość `undefined`. W poniższym przykładzie funkcja zwraca wartość `NaN`, ponieważ próbuje dodać `1` do `undefined`:

```
>>> sum(1)
```

NaN

JavaScript nie wybrzydza podczas pobierania parametrów. Jeśli otrzyma ich więcej, niż jest potrzebne, dodatkowe parametry zostaną zignorowane:

```
>>> sum(1, 2, 3, 4, 5)
```

3

Na dodatek możliwe jest pisanie funkcji, które mogą przyjmować różną liczbę parametrów. Jest to możliwe dzięki tablicy `arguments`, która jest automatycznie tworzona wewnątrz każdej funkcji. Oto funkcja, której działanie polega na zwracaniu wszystkich przekazanych jej argumentów:

```
>>> function args() { return arguments; }
>>> args();
```

[]

```
>>> args( 1, 2, 3, 4, true, 'ninja');
```

[1, 2, 3, 4, true, "ninja"]

Tablica `arguments` pozwoli nam poprawić funkcję `sum()` tak, by przyjmowała ona dowolną liczbę parametrów i dodawała je wszystkie.

```
function sumaNaSterydach() {
  var i, res = 0;
  var liczba_parametrow = arguments.length;
  for (i = 0; i < liczba_parametrow; i++) {
    res += arguments[i];
  }
  return res;
}
```

Jeśli podczas testowania wywołasz tę funkcję z inną niż wcześniej liczbą parametrów (lub nawet bez parametrów), zobaczysz, że działa tak, jak powinna:

```
>>> sumaNaSterydach(1, 1, 1);
```

3

```
>>> sumaNaSterydach(1, 2, 3, 4);
```

10

```
>>> sumaNaSterydach(1, 2, 3, 4, 4, 3, 2, 1);
```

20

```
>>> sumaNaSterydach(5);
```

5

```
>>> sumaNaSterydach();
```

0

Wyrażenie `arguments.length` zwraca liczbę parametrów podanych podczas wywołania funkcji. Jeśli nie rozumiesz jego składni, nie przejmuj się, wrócimy do tego w następnym rozdziale. Wtedy także dowiesz się, że `arguments` w rzeczywistości nie jest tablicą, ale obiektem tablicopodobnym.

Funkcje predefiniowane

Istnieje pewna liczba funkcji, które zostały wbudowane w silnik JavaScriptu i z których można korzystać do woli. Przyjrzyjmy się im. Warto poeksperymentować z tymi funkcjami i przyjrzeć się ich argumentom i wartościom zwracanym, by móc później korzystać z nich w wygodny sposób. Oto lista funkcji wbudowanych:

- `parseInt()`
- `parseFloat()`
- `isNaN()`
- `isFinite()`
- `encodeURI()`
- `decodeURI()`
- `encodeURIComponent()`
- `decodeURIComponent()`
- `eval()`

Zasada czarnej skrzynki

Z reguły podczas korzystania z funkcji Twój program nie musi wiedzieć, jakie czynności są wykonywane wewnątrz danej funkcji. Możesz myśleć o funkcjach jako o czarnych skrzynkach — podajesz im pewne wartości (w postaci parametrów wejściowych) i odbierasz od nich zwracane wyniki. Jest to prawdziwe dla wszystkich funkcji — tych wbudowanych w język JavaScript, tych pisanych przez Ciebie oraz tych stworzonych przez Twoich współpracowników lub nieznanych Ci programistów.

parseInt()

`parseInt()` pobiera argument dowolnego typu (najczęściej łańcuch znaków) i próbuje zamienić go na liczbę całkowitą. Jeśli operacja się nie powiedzie, zwrócona zostanie wartość `NaN`.

```
>>> parseInt('123')
```

123

```
>>> parseInt('abc123')
```

NaN

```
>>> parseInt('1abc23')
```

1

```
>>> parseInt('123abc')
```

123

Funkcja pobiera jeszcze opcjonalny drugi argument, który określa *podstawę*, opisującą typ liczby: dziesiętny, szesnastkowy, binarny itp. Przykładowo: nie ma sensu próba zamiany pobrania liczby dziesiętnej z łańcucha `"FF"`, zatem wynikiem będzie **NaN**, jednak jeśli potraktujemy `"FF"` jako liczbę szesnastkową, otrzymamy wynik **255**.

```
>>> parseInt('FF', 10)
```

NaN

```
>>> parseInt('FF', 16)
```

255

Spróbujmy teraz sparsować liczby o różnych podstawach: `10` (liczba dziesiętna) i `8` (liczba ósemkowa).

```
>>> parseInt('0377', 10)
```

377

```
>>> parseInt('0377', 8)
```

255

Jeśli drugi argument nie zostanie podany, za podstawę uznawana jest liczba `10`, z następującymi wyjątkami:

- Jeśli jako pierwszy argument przekazany zostanie łańcuch zaczynający się od `0x`, drugiemu argumentowi (jeśli nie został podany) przypisana zostanie wartość `16` (liczba zostanie uznana za szesnastkową).
- Jeśli pierwszy parametr zaczyna się od `0`, drugi otrzyma wartość `8`.

```
>>> parseInt('377')
```

377

```
>>> parseInt('0377')
```

255

```
>>> parseInt('0x377')
```

887

Najbezpieczniejszym rozwiązaniem jest określanie podstawy za każdym razem. Jeśli tego nie zrobisz, kod prawdopodobnie zadziała w 99% przypadków (ponieważ najczęściej parsuje się liczby dziesiętne), jednak jeśli trafisz na liczbę zapisaną w innym systemie, możesz osiwieć, zanim uda Ci się znaleźć przyczynę błędu. Wyobraź sobie na przykład, że parsujesz pola formularza, który reprezentuje kalendarz, i że użytkownik wpisał `08`, mając na myśli sierpień. Jeśli nie podasz podstawy, otrzymasz wynik inny niż oczekiwany.

parseFloat()

`parseFloat()` działa podobnie do `parseInt()`, ale oczekuje ułamków. Pobiera ona tylko jeden parametr.

```
>>> parseFloat('123')
123
>>> parseFloat('1.23')
1.23
>>> parseFloat('1.23abc.00')
1.23
>>> parseFloat('a.bc1.23')
NaN
```

Podobnie jak `parseInt()`, `parseFloat()` podda się po napotkaniu pierwszego znaku, z którym nie będzie umiała sobie poradzić, nawet jeśli pozostała część tekstu zawiera poprawne liczby.

```
>>> parseFloat('a123.34')
NaN
>>> parseFloat('a123.34')
NaN
>>> parseFloat('12a3.34')
12
```

`parseFloat()`, w przeciwieństwie do `parseInt()`, jest w stanie poprawnie zinterpretować zapis wykładniczy.

```
>>> parseFloat('123e-2')
1.23
>>> parseFloat('123e2')
12300
```

```
>>> parseInt('1e10')
```

1

isNaN()

Przy pomocy `isNaN()` można sprawdzić, czy wartość wejściowa jest liczbą, której można bezpiecznie używać w operacjach arytmetycznych. `isNaN()` pozwala w wygodny sposób dowiedzieć się, czy funkcjom `parseInt()` i `parseFloat()` udało się sparsować liczbę.

```
>>> isNaN(NaN)
```

true

```
>>> isNaN(123)
```

false

```
>>> isNaN(1.23)
```

false

```
>>> isNaN(parseInt('abc123'))
```

true

Ta funkcja także stara się zamienić parametr wejściowy na liczbę:

```
>>> isNaN('1.23')
```

false

```
>>> isNaN('a1.23')
```

true

Funkcja `isNaN()` jest potrzebna także dlatego, że liczba `NaN` nie jest równa samej sobie. Wynikiem porównania `NaN === NaN` będzie **false**!

isFinite()

Funkcja `isFinite()` sprawdza, czy wartość parametru wejściowego to liczba różna od `Infinity` i różna od `NaN`.

```
>>> isFinite(Infinity)
```

false

```
>>> isFinite(-Infinity)
```

false

```
>>> isFinite(12)
```

true

```
>>> isFinite(1e308)
```

true

```
>>> isFinite(1e309)
```

false

Jeśli dziwią Cię dwa ostatnie wyniki, przypominam, że zgodnie z tym, co napisałem w poprzednim rozdziale, największą dopuszczalną liczbą w języku JavaScript jest `1.7976931348623157e+308`.

Encode/Decode URIs

W adresach URL (*Uniform Resource Locator*) i URI (*Uniform Resource Identifier*) niektóre znaki mają specjalne znaczenie. Jeśli chcemy mieć pewność, że zostaną one zapisane poprawnie (czyli jeśli chcemy zastosować sekwencję uniku), możemy skorzystać z funkcji `encodeURI()` lub `encodeURIComponent()`. Pierwsza z nich zwróci poprawny adres URL, druga założy, że przekazany jej parametr jest tylko częścią URL (na przykład zawiera parametry żądania), i odpowiednio zakoduje wszystkie nietypowe znaki.

```
>>> var url = 'http://www.packtpub.com/scr ipt.php?q=this and that';
>>> encodeURI(url);
```

"http://www.packtpub.com/scr%20ipt.php?q=this%20and%20that"

```
>>> encodeURIComponent(url);
```

"http%3A%2F%2Fwww.packtpub.com%2Fscr%20ipt.php%3Fq%3Dthis%20and%20that"

Działanie przeciwne do `encodeURI()` i `encodeURIComponent()` mają funkcje `decodeURI()` i `decode↪URIComponent()`. W starszym kodzie można natknąć się na starsze funkcje `escape()` i `unescape()`, jednak są one przestarzałe i nie należy ich stosować.

eval()

Funkcja `eval()` pobiera łańcuch znaków i uruchamia go jako kod w języku JavaScript:

```
>>> eval('var ii = 2;')
>>> ii
```

2

`eval('var ii = 2;')` działa dokładnie tak samo jako `var ii = 2;`

Są sytuacje, w których `eval()` się przydaje, jednak w miarę możliwości należy tej funkcji unikać. Z reguły można zastosować inne rozwiązania, które w większości przypadków są bardziej eleganckie i łatwiejsze w utrzymaniu. Weterani JavaScriptu jak mantrę powtarzają zdanie „`eval` is evil" („`eval` to samo zło"). Można wymienić następujące wady tej funkcji:

- Wydajność: wykonywanie kodu „na żywo" jest wolniejsze od wykonywania kodu zapisanego w skrypcie.
- Bezpieczeństwo: JavaScript ma duże możliwości, co oznacza, że przy jego „pomocy" można coś zepsuć. Jeśli nie możesz ufać źródłu, z którego pochodzi wejście przekazywane do `eval()`, nie wywołuj tej funkcji.

Bonus — funkcja alert()

Spójrzmy jeszcze na bardzo popularną funkcję `alert()`. Nie należy ona do rdzenia języka (nie ma jej w specyfikacji ECMA), ale można z niej korzystać w środowisku przeglądarki. Pozwala ona na wyświetlanie komunikatów w okienku dialogowym. Czasami przydaje się to podczas testowania i debugowania aplikacji, chociaż w tym celu lepiej korzystać z debugera Firebug. Na poniższym rysunku widać efekt wykonania kodu `alert("halo!")`.

Pamiętaj tylko, że okienko dialogowe blokuje wątek przeglądarki, co oznacza, że żaden inny kod nie zostanie wykonany, zanim użytkownik nie kliknie *OK*. Jeśli aplikacja jest często aktualizowaną aplikacją AJAX, to `alert()` nie jest najlepszym pomysłem.

Zasięg zmiennych

Warto zwrócić uwagę, zwłaszcza, jeśli jest się osobą, która wcześniej programowała w innym języku, że zmienne w języku JavaScript nie są definiowane w obrębie bloku, tylko funkcji. Oznacza to, że jeśli zmienna została zdefiniowana wewnątrz funkcji, nie jest widoczna poza nią. Natomiast zmienna zdefiniowana wewnątrz bloku `if` lub `for` jest widoczna poza blokiem. *Zmienne globalne* to zmienne używane poza funkcjami, natomiast *zmienne lokalne* to zmienne używane wewnątrz funkcji. Kod wewnątrz funkcji ma dostęp zarówno do zmiennych globalnych, jak i do swoich zmiennych lokalnych.

W poniższym przykładzie:

- funkcja `f()` ma dostęp do zmiennej `global`,
- poza funkcją `f()` zmienna `local` nie istnieje.

```
var global = 1;
function f() {
  var local = 2;
  global++;
  return global;
}
>>> f();
```

2

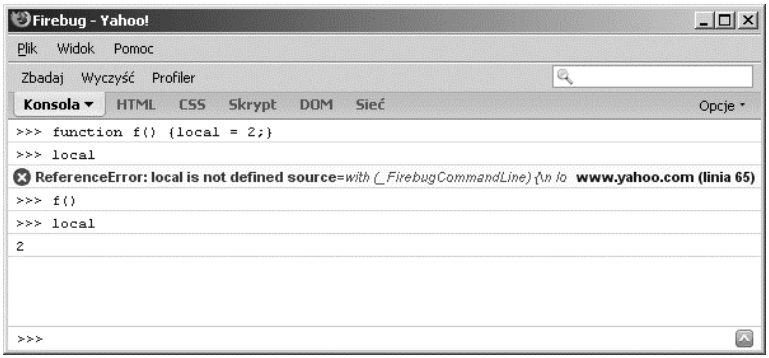
```
>>> f();
```

3

```
>>> local
```

local is not defined

Ponadto należy mieć na uwadze, że jeśli do deklaracji zmiennej nie zostanie użyta instrukcja `var`, zmienna będzie miała zasięg globalny. Spójrzmy na przykład:



Co się stało? Funkcja `f()` zawiera zmienną `local`. Przed wywołaniem funkcji zmienna nie istnieje. Jednak podczas pierwszego wywołania funkcji zmienna jest tworzona i ma zasięg globalny. Dlatego jeśli wówczas spróbujemy sięgnąć do zmiennej `local`, okaże się ona dostępna.

Dobre rady

- Staraj się ograniczać liczbę zmiennych globalnych. Wyobraź sobie dwie osoby pracujące nad dwiema różnymi funkcjami w tym samym skrypcie, które przypadkowo postanawiają nadać tę samą nazwę zmiennej globalnej. Może to doprowadzić do nieoczekiwanych wyników i trudnych do wykrycia błędów.
- Zawsze deklaruj zmienne za pomocą instrukcji `var`.

Poniższy przykład ilustruje ważny aspekt podziału na zmienne lokalne i globalne.

```
var a = 123;
function f() {
  alert(a);
  var a = 1;
  alert(a);
}
f();
```

Być może spodziewasz się, że pierwszy `alert()` wyświetli `123` (wartość globalnej zmiennej `a`), a drugi wyświetli `1` (wartość lokalnej zmiennej `a`). Jednak stanie się inaczej. Pierwszy `alert()` pokaże `"undefined"`. Stanie się tak dlatego, że wewnątrz funkcji zasięg lokalny jest ważniejszy od globalnego. Zmienna lokalna nadpisuje zmienną globalną o tej samej nazwie. Podczas wykonywania pierwszego `alert()`, `a` nie było jeszcze zdefiniowane (stąd wartość `undefined`), ale już istniało w lokalnej przestrzeni nazw.

Funkcje są danymi

Zrozumienie tego punktu widzenia będzie na późniejszym etapie bardzo ważne — funkcje tak naprawdę są danymi. Oznacza to, że następujące dwie metody definiowania funkcji są równoważne:

```
function f(){return 1;}
var f = function(){return 1;}
```

Drugi z pokazanych sposobów definiowania funkcji określa się mianem **zapisu literałowego funkcji**. Jeśli na zmiennej, której została przypisana wartość będąca funkcją, wywołamy operator `typeof`, zwróci on łańcuch znaków `"function"`.

```
>>> function f(){return 1;}
>>> typeof f
```

"function"

Zatem: funkcje w języku JavaScript są specjalnym typem danych. Posiadają dwie istotne cechy:

- zawierają kod,
- są wykonywalne (mogą być wywoływane).

Wiesz już, że funkcje wywołuje się poprzez podanie nawiasu po ich nazwie. Następny przykład pokazuje, że ta metoda zadziała niezależnie od sposobu definicji funkcji. Widać w nim także, że funkcja jest traktowana jak normalna wartość, którą można przypisać nowej zmiennej lub nawet wykasować.

```
>>> var sum = function(a, b) {return a + b;}
>>> var add = sum;
>>> delete sum
```

true

```
>>> typeof sum;
"undefined"
>>> typeof add;
"function"
>>> add(1, 2);
3
```

Ponieważ funkcje to dane przypisane do zmiennych, stosujemy tę samą konwencję nazw co przy nazywaniu zmiennych — nazwa funkcji nie może zaczynać się liczbą i może zawierać dowolną kombinację liter, cyfr oraz znaku podkreślnika.

Funkcje anonimowe

JavaScript pozwala na rozrzucanie fragmentów danych po całym programie. Wyobraź sobie, że Twój program zawiera następujący fragment kodu:

```
>>> "test"; [1,2,3]; undefined; null; 1;
```

Kod wygląda dość dziwnie, ponieważ nie robi nic pożytecznego, jednak jest poprawny i nie spowoduje błędu. Można powiedzieć, że zawiera dane *anonimowe*, czyli nieprzypisane do żadnej zmiennej i nieposiadające nazwy.

Wiesz już, że funkcje można traktować jak wszystkie inne dane. W związku z tym ich także można używać bez podania nazwy:

```
>>> function(a){return a;}
```

Anonimowe fragmenty danych w kodzie nie mogą być zbyt przydatne, chyba że są funkcjami. W takim wypadku istnieją dwa bardzo eleganckie zastosowania tych danych:

- Funkcję anonimową można przekazać jako parametr do innej funkcji. Funkcja odbierająca ten parametr może przeprowadzić operacje na otrzymanej funkcji.
- Funkcje anonimowe możne definiować i od razu uruchamiać.

Przyjrzyjmy się uważniej obu zastosowaniom funkcji anonimowych.

Wywołania zwrotne

Skoro funkcje to dane, które można przypisać zmiennym, to można je definiować, kasować, kopiować… Dlaczego zatem nie miałoby być możliwe *przekazywanie ich jako parametrów* do innych funkcji?

Oto przykład funkcji, która pobiera dwie funkcje jako parametry, wywołuje je, po czym zwraca wynik będący sumą zwróconych przez nie wartości:

```
function wywolaj_i_dodaj(a, b){
  return a() + b();
}
```

Zdefiniujmy teraz dwie pomocnicze funkcje, które będą zwracały ustalone wartości:

```
function jeden() {
  return 1;
}
function dwa() {
  return 2;
}
```

Możemy przekazać je oryginalnej funkcji i obejrzeć wynik:

```
>>> wywolaj_i_dodaj(jeden, dwa);
```

```
3
```

Jako parametry można także przekazywać funkcje anonimowe. Wówczas zamiast definiowania `jeden()` i `dwa()` wystarczyłoby napisać:

```
wywolaj_i_dodaj(function(){return 1;}, function(){return 2;})
```

Jeśli funkcja A zostaje przekazana funkcji B i B wywołuje A, często mówi się, że A jest *wywołaniem zwrotnym* (ang. *callback function*). Jeśli A nie ma nazwy, to jest anonimowym wywołaniem zwrotnym.

Jakie zastosowania mają takie funkcje? Spójrzmy na przykłady, które ilustrują następujące zalety wywołań zwrotnych:

- Można przekazywać funkcje bez konieczności ich nazywania, co oznacza, że potrzebnych jest mniej zmiennych globalnych.
- Jeśli przeniesiemy obowiązek wywołania funkcji na inną funkcję, nasz kod będzie krótszy.
- Wywołania zwrotne mogą korzystnie wpłynąć na wydajność aplikacji.

Przykłady wywołań zwrotnych

Przeanalizujmy częsty scenariusz: mamy funkcję, która zwraca wartość, przekazywaną następnie kolejnej funkcji. W naszym przykładzie pierwsza funkcja, `pomnozRazyDwa()`, przyjmuje trzy parametry, przechodzi przez nie w pętli oraz zwraca tablicę zawierającą wynik. Druga funkcja, `dodajJeden()`, pobiera wartość, dodaje do niej jeden, po czym zwraca wynik.

```
function pomnozRazyDwa(a, b, c) {
  var i, ar = [];
  for(i = 0; i < 3; i++) {
    ar[i] = arguments[i] * 2;
  }
  return ar;
}
function dodajJeden(a) {
  return a + 1;
}
```

Przetestujmy te funkcje:

```
>>> pomnozRazyDwa(1, 2, 3);
```

[2, 4, 6]

```
>>> dodajJeden(100)
```

101

Załóżmy teraz, że chcemy, by tablica `myarr` zawierała trzy elementy, z których każdy przejdzie przez obie funkcje. Zacznijmy od `pomnozRazyDwa()`.

```
>>> var myarr = [];
>>> myarr = pomnozRazyDwa(10, 20, 30);
```

[20, 40, 60]

Możemy teraz wywoływać funkcję `dodajJeden()` w pętli, raz dla każdego elementu tablicy:

```
>>> for (var i = 0; i < 3; i++) {myarr[i] = addOne(myarr[i]);}
>>> myarr
```

[21, 41, 61]

Wszystko zadziała, ale jest tu pole do poprawek. Po pierwsze, przykład uruchamia dwie pętle, które mogą być kosztowne, jeśli powtórzeń jest wiele. Żądany wynik można otrzymać przy użyciu jednej tylko pętli. Oto, jak zmienić funkcję `pomnozRazyDwa()` tak, by jako parametr przyjmowała funkcję i wywoływała ją przy każdej iteracji:

```
function pomnozRazyDwa(a, b, c, callback) {
  var i, ar = [];
  for(i = 0; i < 3; i++) {
    ar[i] = callback(arguments[i] * 2);
  }
return ar;
}
```

Zmieniona wersja funkcji pozwala na wykonanie tej samej pracy przy pomocy jednego wywołania. Przekazuje się do niego wartości początkowe oraz funkcję, która ma zostać wywołana na każdej z tych wartości.

```
>>> myarr = pomnozRazyDwa(1, 2, 3, dodajJeden);
```

[3, 5, 7]

Zamiast definiowania funkcji `dodajJeden()` można skorzystać z funkcji anonimowej, dzięki czemu zdefiniowana zostanie jedna zmienna globalna mniej.

```
>>> myarr = pomnozRazyDwa(1, 2, 3, function(a){return a + 1});
```

[3, 5, 7]

Oczywiście tej samej funkcji można jako parametr przekazać różne funkcje anonimowe:

```
>>> myarr = multiplyByTwo(1, 2, 3, function(a){return a + 2});
```

[4, 6, 8]

Funkcje samowywołujące się

Omówiliśmy już funkcje anonimowe i wywołania zwrotne. Przejdźmy teraz do innego zastosowania funkcji anonimowych — wywoływania funkcji zaraz po ich zdefiniowaniu. Oto przykład:

```
(
  function(){
    alert('uuu!');
  }
)()
```

Początkowo może to wyglądać groźnie, ale tak naprawdę to proste — funkcję anonimową umieszcza się w nawiasie, po którym następuje inny nawias (w przykładzie jest pusty). Drugi nawias oznacza „uruchom teraz". To w nim umieszcza się ewentualne parametry funkcji.

```
(
  function(imie){
    alert('Cześć ' + imie + '!');
  }
)('stary')
```

Jedną z zalet samowywołujacych się funkcji anonimowych jest to, że kod zostanie wykonany bez tworzenia nadmiaru zmiennych. Minus jest taki, że tej samej funkcji nie da się uruchomić dwukrotnie (chyba że znajdzie się wewnątrz pętli lub innej funkcji). Dlatego anonimowe funkcje samowywołujące najlepiej nadają się do wykonywania jednokrotnych zadań inicjalizacyjnych.

Funkcje wewnętrzne (prywatne)

Skoro funkcje są zwykłymi wartościami, nic nie stoi na przeszkodzie, by zdefiniować funkcję wewnątrz innej funkcji.

```
function a(param) {
  function b(theinput) {
    return theinput * 2;
  };
  return 'Wynik wynosi ' + b(param);
};
```

Stosując drugą notację definiowania funkcji, możemy napisać:

```
var a = function(param) {
  var b = function(theinput) {
    return theinput * 2;
  };
return 'Wynik wynosi ' + b(param);
};
```

Kiedy globalna funkcja `a()` zostanie wywołana, wywoła także lokalną funkcję `b()`. Jako że `b()` jest lokalna, nie jest dostępna spoza `a()`, dlatego nazywamy ją funkcją *prywatną*.

```
>>> a(2);
```

"The result is 4"

```
>>> a(8);
```

"The result is 16"

```
>>> b(2);
```

b is not defined

Ze stosowania funkcji prywatnych płyną następujące korzyści:

- Nie dochodzi do zaśmiecenia globalnej przestrzeni nazw (zmniejszone ryzyko kolizji nazw).
- Prywatność: na zewnątrz widoczne są tylko te funkcje, które programista chce udostępnić. Funkcjonalności nieprzeznaczone dla reszty aplikacji są ukryte.

Funkcje, które zwracają funkcje

Wspominałem już, że funkcja zawsze zwraca wartość, a jeśli nie robi tego w sposób jawny, to niejawnie zwracana jest wartość `undefined`. Funkcja zwraca dokładnie jedną wartość, która z powodzeniem może być inną funkcją.

```
function a() {
  alert('A!');
  return function(){
    alert('B!');
  };
}
```

Widoczna powyżej funkcja `a()` wykonuje swoją pracę (mówi `'A!'`) i zwraca inną funkcję, która robi coś innego (mówi `'B!'`). Wynik można przypisać jakiejś zmiennej i używać jej jako normalnej funkcji.

```
>>> var newFunc = a();
>>> newFunc();
```

Pierwsza linia powyższego kodu spowoduje wyświetlenie okienka z wiadomością `'A!'`, a druga — okienka z wiadomością `'B!'`.

Jeśli funkcja zwracana przez inną funkcję ma zostać wykonana natychmiast, bez potrzeby przypisywania jej do nowej zmiennej, wystarczy dodać jeszcze jeden nawias. Wynik końcowy będzie taki sam jak wcześniej.

```
>>> a()();
```

Funkcjo, przepiszże się!

Ponieważ funkcje potrafią zwracać funkcje, możliwe jest zastąpienie oryginalnej funkcji tą zwracaną. Wróćmy do poprzedniego przykładu. Wartość zwróconą przez wywołanie `a()` można przypisać zmiennej `a`, nadpisując w ten sposób istniejącą funkcję:

```
>>> a = a();
```

Powyższa linia przy pierwszym uruchomieniu spowoduje wyświetlenie `'A!'`, jednak jej drugie uruchomienie wyświetli `'B!'`.

Opisany mechanizm jest przydatny, jeśli funkcja wykonuje pewne jednorazowe zadanie. Po zakończeniu zadania zmiennej przechowującej funkcję przypisywana jest nowa wartość, dzięki czemu operacje nie muszą być powtarzane za każdym razem, gdy ktoś wywoła funkcję. W ostatnim przykładzie funkcja została przedefiniowana z zewnątrz — pobraliśmy zwróconą wartość i przypisaliśmy ją funkcji. Jednakże możliwe jest również przepisanie funkcji od środka.

```
function a() {
  alert('A!');
  a = function(){
    alert('B!');
  };
}
```

Przy pierwszym wywołaniu funkcja:

- Wyświetli `'A!'` (załóżmy, że to właśnie jest nasze jednorazowe zadanie inicjalizacyjne).
- Zmieni definicję globalnej zmiennej `a`, przypisując jej nową funkcję.

Każde kolejne wywołanie będzie powodowało wyświetlenie `'B!'`.

Oto inny przykład, który łączy kilka technik omówionych na ostatnich kilku stronach:

```
var a = function() {
  function inicjalizacja(){
    var setup = 'już';
  }
  function normalnaPraca() {
    alert('praca wre!');
  }
  inicjalizacja();
  return normalnaPraca;
}();
```

W przykładzie:

- Mamy funkcje prywatne: `inicjalizacja()` i `normalnaPraca()`.
- Mamy funkcję samowywołującą się: funkcja `a()` jest wywoływana dzięki nawiasowi po jej definicji.
- Pierwsze wywołanie `a()` polega na wywołaniu funkcji `inicjalizacja()` i zwróceniu referencji do zmiennej `normalnaPraca`, która jest funkcją. Zwróć uwagę na brak nawiasów przy zwracanej wartości — nie ma ich dlatego, że zwracamy do funkcji referencję, a nie wynik wywołania tejże funkcji.
- Jako że kod zaczyna się od `var a =`, wartość zwrócona przez samowywołującą się funkcję zostanie przypisana zmiennej `a`.

Jeśli chcesz sprawdzić, czy poprawnie rozumiesz omówiony zakres materiału, spróbuj odpowiedzieć na poniższe pytania. Jakie będzie zachowanie napisanego przed chwilą programu, gdy:

- zostanie wgrany po raz pierwszy?
- po wgraniu zostanie wywołane `a()`?

Przedstawione mechanizmy okazują się bardzo przydatne w środowisku przeglądarki. Różne przeglądarki mogą realizować konkretne zadania na różne sposoby. Przy założeniu, że właściwości przeglądarki nie zmienią się pomiędzy wywołaniami funkcji, możemy stworzyć funkcję, która wybierze sposób działania najlepiej dopasowany do danej przeglądarki, po czym w odpowiedni sposób zmieni swoją definicję, dzięki czemu tylko raz będzie musiała wykrywać typ przeglądarki. Konkretne przykłady zastosowania tego scenariusza będzie można zobaczyć na dalszych stronach książki.

Domknięcia

Pozostała część tego rozdziału jest poświęcona domknięciom (czyż istnieje lepszy sposób na zamknięcie rozdziału?). Domknięcia początkowo mogą wydawać się trudne do zrozumienia, dlatego nie zniechęcaj się, jeśli nie pojmiesz wszystkiego od razu. Postaraj się doczytać rozdział

do końca i poeksperymentować z przykładami, a jeśli niektóre zagadnienia nadal nie będą jasne, możesz do nich wrócić później, kiedy inne mechanizmy omówione w tym rozdziale nie będą już sprawiały Ci żadnego kłopotu.

Zanim zajmiemy się domknięciami, powtórzmy i rozszerzmy trochę pojęcia zakresu w języku JavaScript.

Łańcuch zakresów

Jak już Ci wiadomo, JavaScript nie wyróżnia żadnych zakresów ograniczonych nawiasami klamrowymi, ale istnieje zakres funkcji. Zmienna zdefiniowana wewnątrz funkcji nie jest widoczna poza tą funkcją, natomiast zmienna zdefiniowana wewnątrz bloku kodu (np. po `if` lub w pętli `for`) jest dostępna poza blokiem.

```
>>> var a = 1; function f(){var b = 1; return a;}
>>> f();
```

1

```
>>> b
```

b is not defined

Zmienna `a` należy do globalnej przestrzeni nazw, podczas gdy zmienna `b` tylko do zakresu funkcji `f()`. Dlatego:

- Wewnątrz `f()` widoczne są zarówno `a` i `b`.
- Wewnątrz `f()` widoczna jest zmienna `a`, ale nie zmienna `b`.

Jeśli zdefiniujesz funkcję `n()` osadzoną w `f()`, `n()` będzie miała dostęp do zmiennych ze swojego zakresu, a także do zmiennych swoich „rodziców". W takim wypadku mówimy o *łańcuchu zakresów*, który może być dowolnie długi (głęboki).

```
var a = 1;
function f(){
  var b = 1;
  function n() {
    var c = 3;
  }
}
```

Zasięg leksykalny

Funkcje w języku JavaScript mają zasięg leksykalny. Oznacza to, że funkcje tworzą swoje własne środowisko (zakres) podczas definicji, a nie podczas wywołania. Spójrzmy na przykład:

```
>>> function f1(){var a = 1; f2();}
>>> function f2(){return a;}
>>> f1();
```

a is not defined

Wewnątrz funkcji `f1()` wywołujemy funkcję `f2()`. Ponieważ zmienna lokalna `a` znajduje się także wewnątrz `f1()`, ktoś mógłby się spodziewać, że `f2()` będzie miała dostęp do `a`, jednak tak nie jest. W momencie *definicji* `f2()` (a nie w momencie *wywołania*) nigdzie nie było śladu `a`. `f2()`, podobnie jak `f1()`, ma dostęp jedynie do własnego zakresu oraz do zakresu globalnego. `f1()` i `f2()` nie współdzielą zakresów lokalnych.

Podczas definiowania funkcja zapamiętuje swoje środowisko, to znaczy swój łańcuch zakresów. Nie znaczy to wcale, że funkcja pamięta każdą konkretną zmienną, która pojawiła się w tym zakresie. Wręcz przeciwnie — zmienne można dodawać, usuwać i uaktualniać, a funkcja zawsze będzie widziała najnowszy, aktualny stan zmiennych. Jeśli rozszerzymy przykład o deklarację globalnej zmiennej `a`, stanie się ona widoczna dla `f2()`, ponieważ `f2()` zna ścieżkę do zmiennych globalnych i ma dostęp do całości tego środowiska. Zwróć uwagę na to, że `f1()` zawiera wywołanie `f2()`, które działa — mimo że `f2()` nie została jeszcze zdefiniowana. `f1()` musi tylko posiadać wiedzę o własnym zakresie, by wszystko, co się w nim pojawi, stawało się automatycznie dostępne dla `f1()`.

```
>>> function f1(){var a = 1; f2();}
>>> function f2(){return a;}
>>> f1();
```

a is not defined

```
>>> var a = 5;
>>> f1();
```

5

```
>>> a = 55;
>>> f1();
```

55

```
>>> delete a;
```

true

```
>>> f1();
```

a is not defined

Przedstawiony mechanizm sprawia, że JavaScript jest bardzo elastyczny — można dodawać zmienne, usuwać je, a potem dodawać je ponownie. Możesz poeksperymentować, kasując funkcję `f2()`, a potem definiując ją ponownie, ale z innym ciałem. Funkcja `f1()` nadal będzie działać, ponieważ musi znać jedynie sposób dostępu do swojego zakresu — nie jest jej potrzebna wiedza o tym, co kiedyś do tego zakresu należało. Ciąg dalszy przykładu:

```
true
>>> f1()
```

f2 is not defined

```
>>> var f2 = function(){return a * 2;}
>>> var a = 5;
```

5

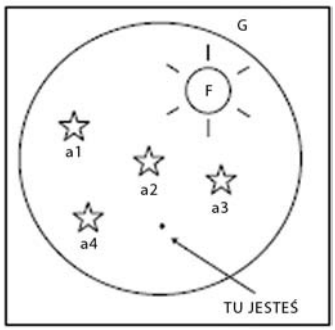
```
>>> f1();
```

10

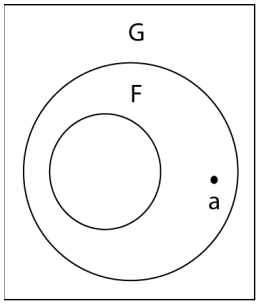
Przerwanie łańcucha za pomocą domknięcia

Zaczniemy od ilustracji.

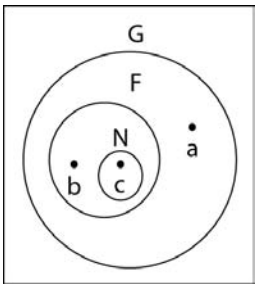
Poniżej widzisz zakres globalny. Wyobraź go sobie jako wszechświat.



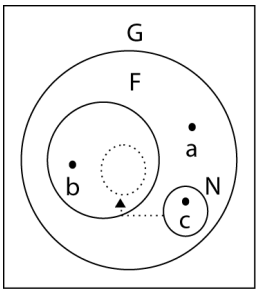
Może on zawierać zmienne, takie jak `a`, i funkcje, jak `F`.



Funkcje posiadają własną przestrzeń, którą mogą wykorzystywać do przechowywania innych zmiennych (i funkcji). W pewnym momencie rysunek będzie wyglądał mniej więcej tak:



Jeśli jesteś w punkcie `a`, jesteś w przestrzeni globalnej. Jeśli w punkcie `b`, który należy do przestrzeni funkcji `F`, masz dostęp do przestrzeni globalnej oraz do przestrzeni `F`. Jeśli znalazłeś się w punkcie `c`, który należy do funkcji `N`, możesz sięgnąć do przestrzeni globalnej, przestrzeni `F` oraz `N`. Nie da się sięgnąć z `a` do `b`, ponieważ punkt `b` nie jest widoczny poza `F`. Możesz natomiast uzyskać dostęp z `c` do `b` lub z `N` do `b`. Ciekawe rzeczy (domknięcie) zaczynają się dziać, gdy jakimś sposobem `N` wydostaje się z `F` i trafia do przestrzeni globalnej.



Co się wtedy dzieje? `N` jest w tej samej przestrzeni globalnej co `a`. Jako że funkcje pamiętają środowisko, w którym zostały zdefiniowane, `N` nadal ma dostęp do przestrzeni `F`, a co za tym idzie dostęp do `b`. Jest to ciekawe dlatego, że `N` znajduje się tam gdzie `a`, a jednak `N` ma dostęp do `b`, zaś `a` nie.

Jak `N` udaje się przerwać łańcuch? Istnieją dwa sposoby: `N` może zostać zmienną globalną (pominięcie `var`) lub może zostać zwrócona przez `F` do przestrzeni globalnej. Zobaczmy, jak to wygląda w praktyce.

Domknięcie 1.

Przyjrzyj się uważnie tej funkcji:

```
function f(){
  var b = "b";
  return function(){
    return b;
  }
}
```

Funkcja zawiera lokalną zmienną `b`, która nie jest dostępna z przestrzeni globalnej:

```
>>> b
```

b is not defined

Zwróć uwagę na wartość zwracaną przez `f()`: jest ona inną funkcją. Możesz o niej myśleć jako o `N` z przedstawionych powyżej rysunków. Nowa funkcja ma dostęp do swojej przestrzeni prywatnej, do przestrzeni funkcji `f()` oraz do przestrzeni globalnej. Widzi zatem również `b`. Ponieważ `f()` można wywołać w przestrzeni globalnej (jest funkcją globalną), możesz ją wywołać i przypisać zwracaną przez nią wartość innej zmiennej globalnej. Wynikiem będzie nowa funkcja globalna, która ma dostęp do prywatnej przestrzeni `f()`.

```
>>> var n = f();
>>> n();
```

"b"

Domknięcie 2.

Przykład, który nastąpi za chwilę, pozwala uzyskać ten sam wynik co przykład wcześniejszy, jednak z zastosowaniem nieco innych metod. Funkcja `f()` nie będzie zwracała funkcji, a zamiast tego utworzy nową, globalną funkcję `n()` wewnątrz swojego ciała.

Zacznijmy od deklaracji zmiennej, do której później przypiszemy nową funkcję. Nie jest to obowiązkowe, ale zawsze warto deklarować zmienne. Definicja funkcji `f()` może wyglądać tak:

```
var n;
function f(){
  var b = "b";
  n = function(){
    return b;
  }
}
```

Co się stanie po wywołaniu `f()`?

```
>>> f();
```

Wewnątrz przestrzeni `f()` definiowana jest nowa funkcja. Ponieważ nie została użyta instrukcja `var`, funkcja jest globalna. W czasie definicji funkcja `n()` znajdowała się wewnątrz `f()`, zatem ma dostęp do zakresu zmiennych `f()`. `n()` zachowa prawo dostępu nawet wtedy, gdy stanie się częścią przestrzeni globalnej.

```
>>> n();
```

"b"

Domknięcie 3. i jedna definicja

W oparciu o to, co zostało powiedziane do tej pory, możemy powiedzieć, że domknięcie jest tworzone, gdy funkcja zachowuje dostęp do zakresu rodzica po tym, jak rodzic zwrócił ją do globalnej przestrzeni nazw.

Argument przekazany funkcji wewnątrz niej jest dostępny jako zmienna globalna. Możesz stworzyć funkcję zwracającą inną funkcję, która z kolei zwraca argument przekazany rodzicowi.

```
function f(arg) {
  var n = function(){
    return arg;
  };
  arg++;
  return n;
}
```

Funkcję można wywołać w następujący sposób:

```
>>> var m = f(123);
>>> m();
```

124

Zauważ, że zmienna `arg` została zwiększona już po definicji funkcji, a pomimo tego `m()` zwróciła aktualną wartość. Jest to kolejny dowód na to, że funkcje są związane ze swoimi zakresami, a nie z przechowywanymi tam w danym momencie zmiennymi i ich wartościami.

Domknięcia w pętli

Pokażę teraz coś, co często prowadzi do bardzo trudnych do wykrycia błędów, ponieważ na pierwszy rzut oka wydaje się, że nie ma tam miejsca na pomyłkę.

Napiszmy pętlę o trzech iteracjach, która za każdym przebiegiem zwraca numer pętli. Funkcje zostaną dodane do tablicy, która na koniec zostanie zwrócona. Oto nasza funkcja:

```
function f() {
  var a = [];
  var i;
  for(i = 0; i < 3; i++) {
    a[i] = function(){
      return i;
    }
  }
return a;
}
```

Wywołajmy ją teraz, przypisując wynikową tablicę zmiennej a.

```
>>> var a = f();
```

Mamy zatem tablicę z trzema funkcjami. Wywołajmy je, podając nawiasy po każdym elemencie tablicy. Oczekiwane zachowanie to wypisanie numerów iteracji: 0, 1 i 2. Spróbujmy:

```
>>> a[0]()
```

3

```
>>> a[1]()
```

3

```
>>> a[2]()
```

3

Hm, niezupełnie to mieliśmy na myśli. Co się stało? Utworzyliśmy trzy domknięcia, które wskazują na tę samą lokalną zmienną i. Domknięcia nie pamiętają wartości, tylko przechowują referencję do zmiennej i — dlatego zwracają jej aktualną wartość. Po wyjściu z pętli wartością zmiennej i jest **3**. Wszystkie funkcje wskazują na tę samą wartość.

(Dla lepszego zrozumienia pętli zastanów się, dlaczego wartością i jest 3, a nie 2).

Jak zatem zaimplementować poprawne zachowanie? Potrzebne nam są trzy różne zmienne. Eleganckie rozwiązanie polega na wykorzystaniu kolejnego domknięcia:

```
function f() {
  var a = [];
  var i;
  for(i = 0; i < 3; i++) {
    a[i] = (function(x){
      return function(){
        return x;
      }
    })(i);
  }
return a;
}
```

Uzyskamy oczekiwany wynik:

```
>>> var a = f();
>>> a[0]();
```

0

```
>>> a[1]();
```

1

```
>>> a[2]();
2
```

W tej wersji nie tworzymy funkcji zwracającej `i`, tylko przekazujemy `i` innej, samowywołującej się funkcji. W tej funkcji `i` staje się lokalną zmienną `x` i za każdym razem ma inną wartość.

Ten sam wynik można uzyskać przy użyciu „normalnej" (czyli niesamowywołującej się) funkcji wewnętrznej. Kluczem do sukcesu jest wykorzystanie środkowej funkcji do ustalenia wartości `i` podczas danej iteracji.

```
function f() {
  function makeClosure(x) {
    return function(){
      return x;
    }
  }
  var a = [];
  var i;
  for(i = 0; i < 3; i++) {
    a[i] = makeClosure(i);
  }
  return a;
}
```

Funkcje dostępowe

Chcę opowiedzieć o jeszcze dwóch sposobach wykorzystania domknięć. Pierwszy z nich polega na utworzeniu funkcji dostępowych *get* (pobranie wartości) i *set* (ustawienie wartości). Załóżmy, że posiadasz zmienną, która może przyjmować wartości tylko ze ściśle określonego zbioru. Nie chcesz odkrywać tej zmiennej, ponieważ chcesz zabezpieczyć się przed sytuacją, w której pewien fragment kodu nada jej niedozwoloną wartość. Rozwiązaniem jest utworzenie schronienia dla tej zmiennej wewnątrz pewnej funkcji i stworzenie dwóch dodatkowych funkcji, które będą odczytywały i ustawiały jej wartość. Funkcja ustawiająca wartość może zawierać pewną logikę, która nie pozwoli na nadanie zmiennej wartości spoza dozwolonego zbioru (jednak dla uproszczenia przykładu pomińmy walidację).

Funkcje dostępowe powinny znaleźć się wewnątrz tej samej funkcji, która zawiera tajną zmienną, tak by dzieliły ten sam zakres:

```
var getValue, setValue;
(function() {
  var secret = 0;
  getValue = function(){
    return secret;
  };
```

```
  setValue = function(v){
    secret = v;
  };
})()
```

Funkcja, która opakowuje zmienną i dwie funkcje dostępowe, jest tutaj samowywołującą się funkcją anonimową. Definiuje ona `setValue()` i `getValue()` jako funkcje globalne, podczas gdy zmienna `secret` pozostaje lokalna i nie jest dostępna bezpośrednio.

```
>>> getValue()
```

0

```
>>> setValue(123)
>>> getValue()
```

123

Iterator

Ostatni przykład domknięcia (a zarazem ostatni przykład w tym rozdziale) pokazuje wykorzystanie domknięć w celu osiągnięcia funkcjonalności *iteratora*.

Wiesz już, jak wykorzystać pętlę do przejścia przez wszystkie elementy zwykłej tablicy. Możesz jednak napotkać bardziej złożoną strukturę danych, w której kolejność elementów jest określana przez bardziej złożony zestaw reguł. Wówczas skomplikowaną logikę rozwiązującą problem „kto następny?" umieszczasz w wygodnej w użyciu funkcji `next()`. Następnie wywołujesz `next()` za każdym razem, gdy chcesz pobrać kolejną wartość. Na potrzeby przykładu wykorzystamy jednak zwykłą tablicę, a nie złożoną strukturę danych.

Oto funkcja inicjalizacyjna, która pobiera tablicę, a także definiuje prywatny wskaźnik `i`, zawsze wskazujący następny element w tablicy:

```
function setup(x) {
  var i = 0;
  return function(){
    return x[i++];
  };
}
```

Wywołanie funkcji `setup()` z parametrem będącym tablicą danych spowoduje automatyczne utworzenie funkcji `next()`.

```
>>> var next = setup(['a', 'b', 'c']);
```

Dalej czekają nas sam przyjemności: wywołując wciąż tę samą funkcję, przejdziemy przez wszystkie elementy tablicy.

```
>>> next();
"a"
>>> next();
"b"
>>> next();
"c"
```

Podsumowanie

Właśnie skończyliśmy podstawowy kurs pojęć związanych z funkcjami. Przejście do konceptów programowania obiektowego oraz do wzorców wykorzystywanych w nowoczesnym programowaniu w języku JavaScript powinno być dla Ciebie proste. Do tej pory unikaliśmy funkcjonalności obiektowych, ale od tej chwili nie będziemy już tego robić. Powtórzmy materiał przedstawiony w tym rozdziale. Omówione zostały następujące kwestie:

- Definiowanie i wywoływanie funkcji.
- Parametry funkcji i ich elastyczność.
- Funkcje wbudowane: `parseInt()`, `parseFloat()`, `isNaN()`, `isFinite()`, `eval()`, a także cztery funkcje do kodowania i dekodowania adresów URL.
- Zakres zmiennych: nie ma zakresu związanego z nawiasami klamrowymi, istnieje zakres funkcji, funkcje mają zakres leksykalny, obowiązuje zasada łańcucha zakresów.
- Funkcje to dane — funkcję można przypisać zmiennej, z czego wynika szereg ciekawych zastosowań, wśród których można wymienić:
  - prywatne funkcje i zmienne,
  - funkcje anonimowe,
  - wywołania zwrotne,
  - samowywołujące się funkcje,
  - funkcje zmieniające swoją definicję.
- Domknięcia.

Ćwiczenia

1. Napisz funkcję, która przekształca szesnastkową definicję koloru (np. niebieski to `"0000FF"`) na reprezentację RGB (np. "**rgb(0, 0, 255)**"). Nazwij funkcję `getRGB()` i przetestuj ją za pomocą następującego kodu:

```
>>> var a = getRGB("#00FF00");
>>> a;
```

"**rgb(0, 255, 0)**"

2. Co pojawi się w konsoli po uruchomieniu każdej z poniższych linii kodu?

```
>>> parseInt(1e1)
>>> parseInt('1e1')
>>> parseFloat('1e1')
>>> isFinite(0/10)
>>> isFinite(20/0)
>>> isNaN(parseInt(NaN));
```

3. Co pojawi się w okienku `alert()` po wykonaniu następującego kodu?

```
var a = 1;
function f() {
  var a = 2;
  function n() {
    alert(a);
  }
  n();
}
f();
```

4. Wszystkie poniższe przykłady spowodują wyświetlenie `"Uuu!"`. Czy potrafisz powiedzieć dlaczego?

4.1

```
var f = alert;
eval('f("Uuu!")');
```

4.2

```
var e;
var f = alert;
eval('e=f')('Uuu!');
```

4.3

```
(
  function(){
    return alert;
  }
)()('Uuu!');
```