Ved Antani, Stoyan Stefanov

Programowanie zorientowane obiektowo w języku Java Script

Wydanie III

Helion

Packt›

Tytuł oryginału: Object Oriented JavaScript - Third Edition

Tłumaczenie: Lech Lachowski

ISBN: 978-83-283-3782-4

First published in the English language under the title 'Object-Oriented JavaScript - Third Edition - (9781785880568)'

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/przojs.zip

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

Printed in Poland.

Spis treści

4

ROZDZIAŁ

Obiekty

Skoro znasz już na wylot podstawowe typy danych, tablice oraz funkcje, przyszła pora na to, co najciekawsze — obiekty.

JavaScript ma ekscentryczne podejście do klasycznego programowania obiektowego. Obiektowość jest jednym z najpopularniejszych paradygmatów programowania i jest podstawą większości języków programowania, takich jak Java i C++. Klasyczne programowanie obiektowe proponuje dobrze przemyślane koncepcje, które są przyjmowane przez większość języków z tej grupy. Jednak JavaScript ma inne podejście. W tym rozdziale przyjrzymy się temu, w jaki sposób JavaScript obsługuje obiektowość.

W tym rozdziale dowiesz się:

- jak tworzyć obiekty i jak ich używać,
- czym są funkcje nazywane konstruktorami,
- jak korzystać z wbudowanych obiektów JavaScriptu.

Od tablic do obiektów

Jak już wiesz z rozdziału 2., tablica jest listą wartości. Każdej wartości odpowiada indeks, przy czym indeksy kolejnych elementów zaczynają się od zera i są zwiększane o jeden dla każdej kolejnej wartości.

```
> var myarr = ['czerwony', 'niebieski', 'żółty', 'fioletowy'];
> myarr;
["czerwony", "niebieski", "żółty", "fioletowy"]
> myarr[0];
"czerwony"
> myarr[3];
"fioletowy"
```

Jeśli wstawimy indeksy do jednej kolumny tablicy, a wartości do drugiej, otrzymamy następującą tablicę par klucz – wartość:

Klucz	Wartość
0	czerwony
1	niebieski
2	żółty
3	fioletowy

Obiekty różnią się od tablic między innymi tym, że programista samodzielnie definiuje klucze. Nie musisz ograniczać się do liczbowych indeksów. Możesz korzystać z bardziej przyjaznych nazw, takich jak `nazwisko`, `data_urodzenia` czy `wiek`.

Przeanalizujmy więc nasz pierwszy, prosty obiekt:

```
var hero = {
  breed: 'Żółw',
  occupation: 'Ninja'
};
```

Możesz zauważyć, że:

- Zmienna, która przechowuje obiekt, nazywa się `hero` (bohater).
- Inaczej niż w przypadku tablic, do definiowania obiektów używa się nawiasów klamrowych `{ i }`, a nie kwadratowych `[ i ]`.
- Elementy obiektu (nazywane właściwościami lub polami) oddziela się za pomocą przecinków.
- Pary `klucz – wartość` rozdziela się dwukropkiem.

Klucze (nazwy właściwości) można umieszczać w cudzysłowach. Poniższe instrukcje są równoważne:

```
var hero = {occupation: 1};
var hero = {"occupation": 1};
var hero = {'occupation': 1};
```

Nie zaleca się stosowania cudzysłowów (chociażby ze względu na oszczędność znaków), jednak w niektórych sytuacjach nie da się ich uniknąć:

- jeśli nazwa właściwości jest jednym z zarezerwowanych słów języka JavaScript (pełna lista w dodatku A);
- jeśli nazwa zawiera znaki specjalne (czyli znaki inne niż litery, liczby, podkreślnik i znak dolara $);
- jeśli pierwszym znakiem nazwy jest cyfra.

W skrócie: jeśli zdecydujesz się nadać właściwość nazwę, która nie jest poprawną nazwą zmiennej, to musisz umieścić ją w cudzysłowie.

Pokazany poniżej dziwaczny twór:

```
var o = {
    $omething: 1,
    'yes or no': 'tak',
    '!@#$%^&*': true
  };
```

jest w pełni poprawnym obiektem. W przypadku drugiej i trzeciej właściwości cudzysłów[1] jest obowiązkowy — pominięcie go doprowadzi do błędu.

W dalszej części rozdziału poznasz inne niż [] i {} sposoby definiowania obiektów i tablic. Tablice zdefiniowane za pomocą [] określa się mianem **literałów tablicowych**, a obiekty zdefiniowane za pomocą {} to **literały obiektowe**.

Elementy, właściwości, metody i składowe

Mówimy, że tablice zawierają elementy. Obiekty, dla odmiany, mają właściwości. Dla JavaScriptu to rozróżnienie nie ma znaczenia — jest ono czysto terminologiczne i pochodzi z innych języków programowania.

Właściwość obiektu może wskazywać funkcję, ponieważ funkcje również są danymi. Takie właściwości nazywamy **metodami**. W poniższym przykładzie `talk` jest metodą:

```
var dog = {
  name: 'Burek',
  talk: function(){
    alert('Hau, hau!');
  }
};
```

Możliwe jest także przechowywanie funkcji w tablicy, jednak taki kod jest rzadkością.

```
> var a = [];
> a[0] = function(what) {alert(what);};
> a[0]('Uuu!');
```

Właściwości obiektu nazywane są również składowymi. Najczęściej ma to miejsce wtedy, kiedy nie ma znaczenia, czy dana właściwość jest funkcją, czy nie.

[1] W języku angielskim cudzysłowu i apostrofów można używać zamiennie — tak samo jest w języku JavaScript — *przyp. tłum.*

Tablice asocjacyjne

W niektórych językach programowania istnieje rozróżnienie na:

- zwykłe tablice **indeksowane**, których kluczami są liczby;
- tablice asocjacyjne, zwane również **tablicami mieszającymi** lub **słownikami**; ich kluczami są łańcuchy znaków[2].

W języku JavaScript tablicom indeksowanym odpowiadają tablice, a tablicom asocjacyjnym — obiekty.

Dostęp do właściwości obiektu

Dostęp do właściwości obiektu można uzyskać na dwa sposoby:

- przy użyciu nawiasów kwadratowych, na przykład `hero['occupation']`;
- przy użyciu kropki, na przykład `hero.occupation`.

Notacja z kropką jest wygodniejsza, ale nie zawsze można ją zastosować. Zasady są takie same jak w przypadku umieszczania w cudzysłowie nazw właściwości: jeśli nazwa nie jest poprawną nazwą zmiennej, nie można skorzystać z notacji z kropką.

Weźmy następujący obiekt:

```
var hero = {
  breed: 'Żółw',
  occupation: 'Ninja'
};
```

Dostęp do właściwości obiektu za pomocą notacji z kropką:

```
> hero.breed;
"Żółw"
```

Dostęp do właściwości obiektu za pomocą notacji nawiasowej:

```
> hero['occupation'];
"Ninja"
```

Próba dostępu do nieistniejącej właściwości kończy się zwróceniem wartości `undefined`:

```
> 'Kolor włosów bohatera to ' + hero.hair_kolor;
"Kolor włosów bohatera to undefined"
```

Obiekty mogą zawierać dane, w tym także inne obiekty.

```
var book = {
  name: 'Paragraf 22',
  published: 1961,
```

[2] Lub dowolne obiekty — *przyp. tłum.*

```
  author: {
    firstname: 'Joseph',
    lastname: 'Heller'
  }
};
```

W celu pobrania wartości właściwości `firstname` obiektu będącego wartością właściwości `author` obiektu `book` należy napisać:

```
> book.author.firstname;
"Joseph"
```

lub, przy użyciu składni z nawiasami:

```
> book['author']['lastname'];
"Heller"
```

Można nawet łączyć notacje:

```
> book.author['lastname'];
"Heller"
> book['author'].lastname;
"Heller"
```

Istnieje jeszcze jedna sytuacja, w której konieczne jest użycie notacji nawiasowej. Jeśli nazwa właściwości, do której chcemy sięgnąć, nie jest znana w czasie pisania kodu, można przypisać jej wartość zmiennej:

```
> var key = 'firstname'
> book.author[key];
Joseph
```

Wywoływanie metod obiektu

Skoro metoda jest po prostu właściwością, która jest także funkcją, dostęp do metod odbywa się tak samo jak dostęp do zwykłych właściwości: przy użyciu notacji z kropką lub notacji nawiasowej. Metody wywołuje się jak wszystkie inne funkcje: należy po nazwie metody dodać nawiasy, które wydadzą rozkaz „Wykonać!”.

```
var hero = {
  breed: 'Żółw',
  ocupation: 'Ninja',
  say: function() {
    return 'Moja specjalizacja to ' + hero.occupation;
  }
}
> hero.say();
"Moja specjalizacja to Ninja"
```

Jeśli metoda pobiera parametry, przekazujemy je dokładnie tak samo jak w przypadku zwykłych funkcji:

```
> hero.say('a', 'b', 'c');
```

To, że dostęp do właściwości może odbywać się za pomocą nawiasów kwadratowych, nie zmienia sposobu wywoływania funkcji. Nadal stosujemy w tym celu parę nawiasów:

```
> hero['say']();
```

Nie jest to jednak powszechna praktyka, chyba że nazwa metody nie jest znana w trakcie pisania kodu, lecz jest definiowana w czasie jego wykonywania:

```
var method = 'say';
hero[method]();
```

> Nie stosuj cudzysłowów, chyba że musisz użyć notacji kropkowej w celu uzyskania dostępu do metod i właściwości. W literałach obiektowych nie umieszczaj nazw właściwości w cudzysłowie.

Modyfikacja właściwości i metod

JavaScript pozwala modyfikować składowe (czyli właściwości i metody) istniejących obiektów w dowolnym momencie. Można dodawać nowe składowe i usuwać stare. Można utworzyć pusty obiekt, a właściwości i metody dodać do niego później. Zobaczmy, jak to zrobić.

Pusty obiekt:

```
> var hero = {};
```

> **„Pusty" obiekt**
>
> Ten punkt rozdziału zaczęliśmy od zdefiniowania „pustego" obiektu `var hero = {}`. Napisaliśmy to słowo w cudzysłowie, ponieważ dany obiekt nie jest w rzeczywistości pusty ani bezużyteczny. Chociaż na tym etapie nie ma on jeszcze własnych właściwości, już pewne właściwości odziedziczył. Różnicę między właściwościami własnymi i odziedziczonymi poznasz nieco później. Tak więc obiekt w ES3 nie jest nigdy tak naprawdę pusty. Jednak w ES5 istnieje sposób utworzenia zupełnie pustego obiektu, który niczego nie dziedziczy, ale nie wybiegajmy za bardzo do przodu.

1. Dostęp do nieistniejącej właściwości:

```
> typeof hero.breed;
"undefined"
```

2. Dodanie dwóch właściwości i metody:

```
> hero.breed = 'Żółw';
> hero.name = 'Leonardo';
```

```
> hero.sayName = function() {
    return hero.name;
  };
```

3. Wywołanie metody:

```
> hero.sayName();
"Leonardo"
```

4. Usuwanie właściwości:

```
> delete hero.name;
true
```

5. Próba ponownego wywołania metody zwracającej imię zakończy się niepowodzeniem, ponieważ nie będzie można odnaleźć właściwości `name`:

```
> hero.sayName();
undefined
```

Elastyczne obiekty

Każdy obiekt można zmienić w dowolnym momencie, na przykład dodając i usuwając właściwości lub zmieniając ich wartości. Istnieją jednak wyjątki od tej reguły. Kilku właściwości niektórych wbudowanych obiektów nie można zmieniać (np. `Math.PI`, o czym przekonasz się później). Ponadto specyfikacja ES5 oferuje możliwości zapobiegania zmianom obiektów. Więcej informacji na ten temat znajdziesz w dodatku C.

Wartość this

W poprzednim przykładzie widzieliśmy metodę `sayName()`, która do właściwości `name` obiektu `hero` odwoływała się za pomocą składni `hero.name`. Istnieje jednak inny, bardziej ogólny sposób dostępu z wnętrza metody do aktualnego obiektu (to znaczy do obiektu, do którego należy metoda): poprzez specjalną wartość `this`.

```
> var hero = {
    name: 'Raphael',
    sayName: function() {
      return this.name;
    }
  }
> hero.sayName();
"Raphael"
```

Jak widać, `this` (z ang. „ten") oznacza bieżący obiekt.

Konstruktory

Obiekty można tworzyć także przy użyciu funkcji nazywanych konstruktorami. Przykład:

```
function Hero() {
  this.occupation = 'Ninja';
}
```

Konstruktor wywołujemy przy użyciu operatora `new`:

```
> var hero = new Hero();
> hero.occupation;
"Ninja"
```

Przewagą tego sposobu tworzenia obiektów jest to, że konstruktory mogą przyjmować parametry. Zmieńmy kod konstruktora tak, by pobierał jeden parametr i przypisywał jego wartość do właściwości `name`.

```
function Hero(name) {
  this.name = name;
  this.occupation = 'Ninja';
  this.whoAreYou = function() {
    return "Jestem " +
           this.name +
           ", a moja specjalizacja to " +
           this.occupation;
  };
}
```

Przy użyciu jednego konstruktora można utworzyć wiele różnych obiektów:

```
> var h1 = new Hero('Michelangelo');
> var h2 = new Hero('Donatello');
> h1.whoAreYou();
"Jestem Michelangelo, a moja specjalizacja to Ninja"
> h2.whoAreYou();
"Jestem Donatello, a moja specjalizacja to Ninja"
```

> Konwencja nakazuje zaczynać nazwy konstruktorów wielką literą, dzięki czemu od razu można zorientować się, że nie mamy do czynienia z normalną funkcją.

Wywołanie konstruktora bez operatora `new` nie zostanie uznane za błąd, ale może prowadzić do nieoczekiwanych wyników:

```
> var h = Hero('Leonardo');
> typeof h;
"undefined"
```

Co tu zaszło? Ponieważ nie został użyty operator `new`, nie powstał nowy obiekt. Funkcja została wywołana jako zwykła funkcja, a nie jako konstruktor, zatem `h` zawiera wartość zwracaną przez funkcję. Ponieważ jednak funkcja nie zawiera instrukcji `return`, w rzeczywistości zwraca wartość `undefined`, która zostaje przypisana zmiennej `h`.

Do czego w takim przypadku odnosi się wskaźnik `this`? Otóż odnosi się on do obiektu globalnego.

Obiekt globalny

Omawialiśmy już zmienne globalne (i potrzebę ich unikania). Mówiliśmy także o tym, że programy napisane w języku JavaScript są uruchamiane wewnątrz środowiska (na przykład przeglądarki). Skoro wiesz już o istnieniu obiektów, musisz poznać całą prawdę: środowisko zapewnia obiekt globalny, a wszystkie zmienne globalne są jego właściwościami.

Jeśli uruchamiasz programy w środowisku przeglądarki, Twoim obiektem globalnym jest `window` („okno”). Innym sposobem uzyskania dostępu do obiektu globalnego (dotyczy to również większości innych środowisk) jest użycie słowa kluczowego `this` poza konstruktorem, na przykład w globalnym kodzie programu poza jakąkolwiek funkcją.

Możesz przekonać się o istnieniu obiektu globalnego, deklarując zmienną globalną poza wszelkimi funkcjami:

```
> var a = 1;
```

Dostęp do niej uzyskasz na różne sposoby:

- odwołując się do zmiennej `a`;
- odwołując się do właściwości obiektu globalnego, na przykład `window['a']` lub `window.a`;
- odwołując się do właściwości obiektu globalnego za pomocą `this`:

```
> var a = 1;
> window.a;
1
> this.a;
1
```

Wróćmy do przykładu, w którym definiowaliśmy konstruktor i wywoływaliśmy go bez użycia operatora `new`. Jak już mówiliśmy, w takim wypadku `this` odwołuje się do obiektu globalnego, a wszystkie własności ustawione za pomocą `this` stają się własnościami obiektu globalnego (w przypadku przeglądarki będzie to `window`).

Zadeklarowanie konstruktora i wywołanie go bez `new` zwróci **undefined**.

```
> function Hero(name) {
    this.name = name;
  }
```

```
> var h = Hero('Leonardo');
> typeof h;
"undefined"
> typeof h.name;
TypeError: Cannot read property 'name' of undefined
```

Ponieważ wewnątrz funkcji `Hero` pojawiło się słowo kluczowe `this`, utworzona została zmienna globalna (właściwość obiektu globalnego) o nazwie `name`.

```
> name;
"Leonardo"
> window.name;
"Leonardo"
```

Jeśli konstruktor zostanie wywołany z użyciem `new`, zwrócony zostanie nowy obiekt, do którego będzie się odnosiło słowo `this`.

```
> var h2 = new Hero('Michelangelo');
> typeof h2;
"object"
> h2.name;
"Michelangelo"
```

Także wbudowane funkcje globalne z rozdziału 3. można wywołać jako metody obiektu `window`. Poniższe dwa fragmenty kodu są równoważne:

```
> parseInt('101 dalmatyńczyków');
101
> window.parseInt('101 dalmatyńczyków');
101
```

Właściwość constructor

Gdy obiekt jest tworzony, otrzymuje on specjalną o nazwie `constructor`. Zawiera ona referencję do konstruktora, który został użyty do utworzenia obiektu.

Kontynuując przykład z bohaterami:

```
> h2.constructor;
function Hero(name) {
  this.name = name;
}
```

Jako że właściwość `constructor` zawiera referencję do funkcji, można wywołać tę funkcję w celu utworzenia nowego obiektu. Poniższy kod oznacza mniej więcej: „Nie interesuje mnie, jak powstał obiekt `h2`, ale chcę dostać jeszcze jeden taki sam”.

```
> var h3 = new h2.constructor('Raphael');
> h3.name;
"Raphael"
```

Konstruktorem obiektów literałowych jest wbudowana funkcja `Object()` (więcej na jej temat w dalszej części rozdziału).

```
> var o = {};
> o.constructor;
function Object() { [native code] }
> typeof o.constructor;
"function"
```

Operator instanceof

Przy użyciu operatora `instanceof` można sprawdzić, czy obiekt został utworzony za pomocą określonego konstruktora:

```
> function Hero(){}
> var h = new Hero();
> var o = {};
> h instanceof Hero;
true
> h instanceof Object;
true
> o instanceof Object;
true
```

Zwróć uwagę, że podczas sprawdzania po nazwie konstruktora nie podaje się nawiasów (zatem nie piszemy `h instanceof Hero()`). Jest tak dlatego, że nie wywołujemy funkcji, tylko odwołujemy się do niej za pomocą nazwy, jak do każdej innej zmiennej.

Funkcje zwracające obiekty

Obiekty można tworzyć nie tylko za pomocą konstruktorów i operatora `new`, ale także za pomocą zwykłych funkcji. Możliwe jest napisanie funkcji, która wykona pewne zadania przygotowawcze i na koniec zwróci wartość będącą obiektem.

Poniższy przykład przedstawia prostą funkcję o nazwie `factory()` (fabryka), która produkuje obiekty:

```
function factory(name) {
  return {
    name: name
  };
}
```

Korzysta się z niej w następujący sposób:

```
> var o = factory('jeden');
> o.name
"jeden"
```

```
> o.constructor
function Object() { [native code] }
```

Można także korzystać z konstruktorów i zwracać obiekty inne niż `this`, czyli zmieniać standardowe zachowanie konstruktora. Zobaczmy, jak to zrobić.

Oto najczęstszy scenariusz wykorzystania konstruktora:

```
> function C() {
    this.a = 1;
  }
> var c = new C();
> c.a
1
```

A jednak można zrobić coś takiego:

```
> function C2() {
    this.a = 1;
    return {b: 2};
  }
> var c2 = new C2();
> typeof c2.a;
"undefined"
> c2.b;
2
```

Co się stało? Zamiast obiektu `this`, który posiada pole `a`, konstruktor zwrócił inny obiekt, który posiada właściwość `b`. Jest to możliwe tylko wtedy, gdy zwracana wartość jest obiektem. W przeciwnym wypadku (jeśli zwrócona zostanie dowolna wartość niebędąca obiektem), konstruktor zachowa się zgodnie ze standardowym scenariuszem i zwróci `this`.

Jeśli zastanawiasz się, w jaki sposób tworzone są obiekty wewnątrz konstruktorów, wyobraź sobie, że zmienna o nazwie `this` jest zdefiniowana na początku funkcji, a zwracana na końcu. Rozważmy następujący kod:

```
function C() {
  // var this = {};  // to jest pseudokod; nie można tak robić
  this.a = 1;
  // zwracanie this;
}
```

Przekazywanie obiektów

Podczas przypisywania obiektu do innej zmiennej lub przekazywania go do funkcji w rzeczywistości przekazuje się jedynie referencję do tego obiektu. Modyfikacja tej referencji pociąga za sobą modyfikację oryginalnego obiektu.

W poniższym fragmencie obiekt jest przypisywany do nowej zmiennej, a następnie uzyskana w ten sposób kopia obiektu jest zmieniana. W wyniku tego zmienia się także pierwotny obiekt:

```
> var original = {howmany: 1};
> var mycopy = original;
> mycopy.howmany;
1
> mycopy.howmany = 100;
100
> original.howmany;
100
```

Tak samo ma się sprawa z przekazywaniem obiektów funkcjom:

```
> var original = {howmany: 100};
> var nullify = function(o) {o.howmany = 0;}
> nullify(original);
> original.howmany;
0
```

Porównywanie obiektów

Wynikiem porównania dwóch obiektów będzie `true` tylko wtedy, gdy porównywane będą dwie referencje do tego samego obiektu. Jeśli porównamy dwa oddzielne obiekty, które akurat mają ten sam zestaw właściwości i metod, to mimo wszystko otrzymamy `false`.

Utwórzmy dwa obiekty, które wyglądają tak samo:

```
> var fido  = {breed: 'pies'};
> var benji = {breed: 'pies'};
```

Wynikiem porównania będzie `false`:

```
> benji === fido;
false
> benji == fido;
false
```

Utwórzmy teraz nową zmienną `mydog` i przypiszmy jej jeden z obiektów. W ten sposób otrzymamy dwie zmienne wskazujące ten sam obiekt.

```
> var mydog = benji;
```

Teraz `mydog` i `benji` są referencjami do tego samego obiektu. Zmiana właściwości `mydog` pociągnie za sobą zmianę właściwości zmiennej `benji`. Wynikiem porównania będzie `true`.

```
> mydog === benji;
true
```

Ponieważ `fido` jest innym obiektem, nie zostanie dopasowany do `mydog`:

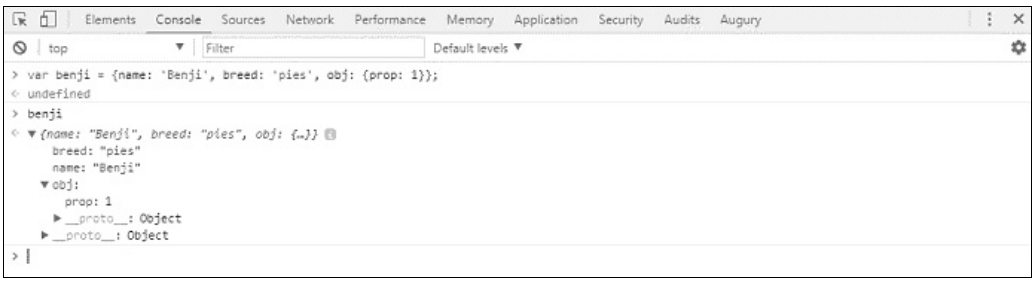
```
> mydog === fido;
false
```

Obiekty w konsoli silnika WebKit

Zanim na poważnie zajmiemy się obiektami wbudowanymi, powiedzmy kilka słów na temat pracy z obiektami w konsoli silnika WebKit.

Prawdopodobnie udało się już, na podstawie przykładów przedstawionych w tym rozdziale, wyciągnąć pewne wnioski na temat sposobu wyświetlania obiektów w konsoli. Jeśli utworzysz obiekt, a następnie wpiszesz jego nazwę, zostanie on przedstawiony w postaci możliwej do rozwinięcia listy elementów (na co wskazuje czarny trójkąt na początku pierwszej linii danych wyjściowych).

Jeśli klikniesz ten obiekt, lista jego właściwości zostanie rozwinięta. Jeśli dana właściwość sama jest obiektem, obok jej nazwy również pojawi się czarny trójkąt, za pomocą którego można wyświetlić szczegóły zagnieżdżonego obiektu. Zobaczmy następujący przykład:

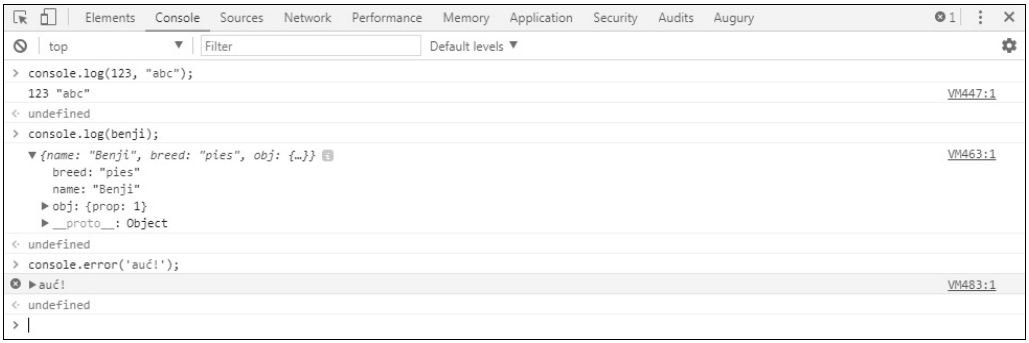


> Na razie możesz zignorować właściwość `__proto__`. Więcej na ten temat dowiesz się w następnym rozdziale.

Wypisywanie w konsoli za pomocą metody console.log

Konsola daje nam dostęp do obiektu o nazwie `console`, którego metody, takie jak `console.log()` i `console.error()`, pozwalają wypisać w konsoli dowolną wartość (patrz rysunek na następnej stronie).

Metoda `console.log()` przydaje się, gdy trzeba szybko coś przetestować lub gdy skrypt ma wypisywać informacje ułatwiające debugowanie. Poniższy przykład pokazuje zastosowanie tej metody w pętli:




```
> for(var i = 0; i < 5; i++) {
    console.log(i);
  }
0
1
2
3
4
```

Literały obiektowe ES6

ES6 wprowadza bardzo zwięzłą składnię do definiowania obiektów za pomocą tzw. literałów. Oferuje skróconą notację dla inicjowania właściwości i definicji funkcji. Ta skrócona składnia ES6 przypomina znaną składnię formatu JSON. Rozważmy następujący fragment kodu:

```
let a = 1
let b = 2
let val = {a: a, b: b}
console.log(val) // {"a":1,"b":2}
```

Jest to typowy sposób przypisywania wartości właściwościom. Jeśli nazwa zmiennej i klucz właściwości są takie same, ES6 umożliwia użycie składni skrótowej. Powyższy kod można napisać następująco:

```
let a = 1
let b = 2
let val = {a, b}
console.log(val) // {"a":1,"b":2}
```

Podobna składnia jest również dostępna dla definicji metod. Jak wspomnieliśmy, metody są po prostu takimi właściwościami obiektu, których wartości są funkcjami. Rozważmy następujący przykład:

```
var obj = {
  prop: 1,
  modifier: function() {
```

```
    console.log(this.prop);
  }
}
```

W ES6 dostępny jest zwięzły sposób definiowania metod. Po prostu pomijamy słowo kluczowe `function` i dwukropek (`:`). Odpowiednik powyższego kodu w ES6 będzie wyglądał tak:

```
var obj = {
  prop: 1,
  modifier () {
    console.log(this.prop);
  }
}
```

ES6 pozwala obliczyć klucz właściwości. Wcześniej można było używać tylko ustalonych nazw właściwości. Oto przykład:

```
var obj = {
  prop: 1,
  modifier: function () {
    console.log(this.prop);
  }
}
obj.prop = 2;
obj.modifier(); // 2
```

Jak widać, w tym przypadku jesteśmy ograniczeni do używania ustalonych nazw kluczy: `prop` i `modifier`. Jedna specyfikacja ES6 umożliwia używanie obliczanych kluczy właściwości. Możliwe jest również tworzenie kluczy właściwości dynamicznie przy użyciu wartości zwracanych przez funkcję:

```
let vehicle = "car"
function vehicleType(){
  return "truck"
}
let car = {
  [vehicle+"_model"]: "Ford"
}
let truck= {
  [vehicleType() + "_model"]: "Mercedes"
}
console.log(car)   // {"car_model":"Ford"}
console.log(truck) // {"truck_model":"Mercedes"}
```

Używamy wartości zmiennej `vehicle`, aby połączyć ją z ustalonym łańcuchem znaków w celu uzyskania klucza właściwości podczas tworzenia obiektu `car`. W drugiej części kodu tworzymy właściwość, łącząc ustalony łańcuch znaków z wartością zwracaną przez funkcję. Ten sposób obliczania kluczy właściwości zapewnia dużą elastyczność podczas tworzenia obiektów, dzięki czemu można wyeliminować znaczne ilości szablonowego i powtarzającego się kodu.

Ta składnia ma również zastosowanie do definicji metod:

```
let object_type = "Vehicle"
let obj = {
  ["get"+object_type]() {
    return "Ford"
  }
}
```

Właściwości i atrybuty obiektów

Każdy obiekt ma kilka właściwości. Każda właściwość ma z kolei klucz i atrybuty. W tych atrybutach jest przechowywany stan właściwości. Wszystkie właściwości mają następujące atrybuty:

- `enumerable` (typu `boolean`): wskazuje, czy dana właściwość obiektu jest wyliczalna. Właściwości systemu są niewyliczalne, podczas gdy właściwości użytkownika są wyliczalne. Jeśli nie ma uzasadnionego powodu, ten atrybut powinien pozostać niezmieniony.
- `configurable` (typu `boolean`): jeśli ten atrybut ma wartość false, nie można usunąć ani edytować danej właściwości (nie można zmienić żadnego z jego atrybutów).

Do pobierania właściwości własnych obiektu można użyć metody `Object.getOwnProperty↪Descriptor()`:

```
let obj = {
  age: 25
}
console.log(Object.getOwnPropertyDescriptor(obj, 'age'));
//{"value":25,"writable":true,"enumerable":true,"configurable":true}
```

Natomiast zdefiniować właściwość można przy użyciu metody `Object.defineProperty()`:

```
let obj = {
  age: 25
}
Object.defineProperty(obj, 'age', { configurable: false })
console.log(Object.getOwnPropertyDescriptor(obj, 'age'));
//{"value":25,"writable":true,"enumerable":true,"configurable":false}
```

Chociaż prawdopodobnie nigdy nie będziesz używać tych metod, ważne jest, aby zrozumieć, czym są właściwości i atrybuty obiektu. W następnym podrozdziale omówimy, jak pewne metody obiektu `Object` są używane w kontekście niektórych z tych właściwości.

Metody obiektów w ES6

ES6 wprowadza kilka statycznych metod pomocniczych dla obiektów. `Object.assign` jest metodą pomocniczą, która zastępuje popularne domieszki przy wykonywaniu płytkiej kopii obiektu. Natomiast metoda `Object.is` zapewnia trochę bardziej precyzyjny sposób porównywania wartości.

Kopiowanie właściwości za pomocą Object.assign

Ta metoda jest używana do kopiowania właściwości obiektu źródłowego do obiektu docelowego. Innymi słowy, ta metoda scala obiekt źródłowy z obiektem docelowym i modyfikuje ten ostatni:

```
let a = {}
Object.assign(a, { age: 25 })
console.log(a) // {"age":25}
```

Pierwszym parametrem przekazywanym do `Object.assign` jest cel, do którego skopiowane zostaną właściwości źródłowe. Ten sam obiekt docelowy jest zwracany do podmiotu wywołującego. Właściwości obiektu docelowego, które znajdują się również w obiekcie źródłowym, są nadpisywane, podczas gdy właściwości niebędące częścią obiektu źródłowego są ignorowane:

```
let a = {age : 23, gender: "męska"}
Object.assign(a, { age: 25 }) // age jest nadpisywane, ale gender jest ignorowane
console.log(a) // {"age":25, "gender":"męska"}
```

Metoda `Object.assign` może przyjmować wiele obiektów źródłowych. Możesz użyć składni `Object.assign(cel, źródło1, źródło2, ...)`. Oto przykład:

```
console.log(Object.assign({a:1, b:2}, {a: 2}, {c: 4}, {b: 3}))
//Object {
//"a": 2,
//"b": 3,
//"c": 4
```

W tym fragmencie przypisujemy właściwości z wielu obiektów źródłowych. Zwróć też uwagę, że `Object.assign()` zwraca obiekt docelowy, którego z kolei używamy wewnątrz metody `console.log()`.

Warto zauważyć, że przy użyciu metody `Object.assign()` można skopiować jedynie wyliczalne własności własne (nieodziedziczone). Właściwości pochodzące z łańcucha prototypów (zostaną one omówione w rozdziale 7., gdy będziemy zajmować się dziedziczeniem) nie są brane pod uwagę. Nasza wcześniejsza dyskusja na temat właściwości wyliczalnych pomoże Ci zrozumieć to rozróżnienie.

W poniższym przykładzie utworzymy właściwość niewyliczalną za pomocą metody `define ↪Property()` i sprawdzimy, czy `Object.assign()` zignoruje tę właściwość:

```
let a = {age : 23, gender: "male"}
Object.defineProperty(a, 'superpowers', {enumberable:false, value:'ES6'})
console.log(a) // { age: 23, gender: "male", superpowers: "ES6"  }
let b = {}
Object.assign(b, a)
Console.log(b) // { age: 23, gender: "male" }
```

Właściwość zdefiniowana jako `superpowers` ma ustawiony atrybut enumerable na `false`. Podczas kopiowania właściwości ta właściwość jest ignorowana.

Porównywanie właściwości za pomocą Object.is

ES6 zapewnia nieco dokładniejszy sposób porównywania wartości. Mówiliśmy już na temat ścisłego operatora równości `===`. Jednak dla wartości `NaN`, `-0` i `+0` ścisły operator równości zachowuje się niespójnie. Oto przykład:

```
console.log(NaN===NaN) // false
console.log(-0===+0)   // true
// Metoda Object.is z ES6
console.log(Object.is(NaN,NaN)) // true
console.log(Object.is(-0,+0))   // false
```

Poza tymi dwoma przypadkami metodę `Object.is()` można bezpiecznie zastąpić operatorem `===`.

Destrukturyzacja

Z obiektami i tablicami będziesz pracował cały czas w trakcie kodowania. Notacje obiektowe i tablicowe w języku JavaScript przypominają format JSON. Będziesz definiował obiekty i tablice, a następnie pobierał z nich elementy. ES6 zapewnia wygodną składnię znacznie poprawiającą sposób uzyskiwania dostępu do właściwości (składowych) obiektów i tablic. Rozważmy typowy kod, jaki często zdarza nam się pisać:

```
var config = {
  server: 'localhost',
  port: '8080'
}
var server = config.server;
var port = config.port;
```

Wyodrębniliśmy tutaj wartości serwera i portu z obiektu `config` i przypisaliśmy je do zmiennych lokalnych. Całkiem proste! Jednak gdyby ten obiekt miał wiele właściwości, a niektóre z nich zagnieżdżone, napisanie tej prostej operacji byłoby bardzo żmudne.

Składnia destrukturyzacji ES6 pozwala na zastosowanie literału obiektowego po lewej stronie instrukcji przypisania. W poniższym przykładzie definiujemy obiekt `config` z kilkoma właściwościami. Następnie używamy destrukturyzacji do przypisania wartości właściwości obiektu `config` poszczególnym zmiennym po lewej stronie instrukcji przypisania:

```
let config = {
  server: 'localhost',
  port: '8080',
  timeout: 900,
}
let {server,port} = config
console.log(server, port) // "localhost" "8080"
```

Jak widać, server i port są zmiennymi lokalnymi, które otrzymały właściwości z obiektu `config`, ponieważ nazwy właściwości były takie same jak nazwy zmiennych lokalnych. Podczas przypisywania do zmiennych lokalnych można również wybrać konkretne właściwości. Oto przykład:

```
let {timeout : t} = config
console.log(t) // 900
```

Tutaj z obiektu `config` wybieramy tylko właściwość `timeout` i przypisujemy ją do zmiennej lokalnej `t`.

Można również użyć składni destrukturyzacji w celu przypisania wartości do wcześniej zadeklarowanych zmiennych. W takim przypadku trzeba umieścić instrukcję przypisania w nawiasach:

```
let config = {
  server: 'localhost',
  port: '8080',
  timeout: 900,
}
let server = '127.0.0.1';
let port = '80';
({server,port} = config) // przypisanie umieszczone w ()
console.log(server, port) // "localhost" "8080"
```

Ponieważ wyrażenie destrukturyzacyjne ewaluuje do prawej strony wyrażenia, można go używać wszędzie tam, gdzie oczekiwana jest wartość, na przykład w wywołaniu funkcji, tak jak pokazano tutaj:

```
let config = {
  server: 'localhost',
  port: '8080',
  timeout: 900,
}
let server='127.0.0.1';
let port ='80';
let timeout ='100';
```

```
function startServer(configValue){
  console.log(configValue)
}
startServer({server,port,timeout} = config)
```

Jeśli określisz zmienną lokalną z nazwą właściwości, która nie istnieje w obiekcie, ta zmienna otrzyma wartość `undefined`. Jednak podczas używania zmiennych w przypisaniu destrukturyzacyjnym można opcjonalnie określić wartości domyślne:

```
let config = {
  server: 'localhost',
  port: '8080'
}
let {server,port,timeout=0} = config
console.log(timeout)
```

W tym przykładzie podaliśmy wartość domyślną dla nieistniejącej właściwości `timeout`, aby do zmiennych lokalnych nie zostały przypisane wartości `undefined`.

Destrukturyzacja działa również z tablicami, a składnia jest bardzo podobna do składni dla obiektów. Trzeba tylko zastąpić literały obiektowe literałami tablicowymi:

```
const arr = ['a','b']
const [x,y] = arr
console.log (x,y) / "a" "b"
```

Jak widać, jest to dokładnie taka sama składnia jak wcześniej. Zdefiniowaliśmy tablicę `arr`, a potem użyliśmy składni destrukturyzacji, aby przypisać elementy tej tablicy do dwóch zmiennych lokalnych, `x` i `y`. Tutaj przypisanie odbywa się na podstawie kolejności elementów w tablicy. Ponieważ istotna jest tylko pozycja elementów, możemy niektóre pominąć, jeśli chcemy. Oto przykład:

```
const days = ['Czwartek','Piątek','Sobota','Niedziela']
const [,,sat,sun] = days
console.log (sat,sun) // "Sobota" "Niedziela"
```

W tym przypadku wiemy, że potrzebujemy elementów z pozycji 2 i 3 (indeks tablicy zaczyna się od 0), a więc ignorujemy elementy z pozycji 0 i 1. Destrukturyzacja tablicy może wyeliminować użycie zmiennej `temp` przy zamianie wartości dwóch zmiennych. Rozważmy następujący kod:

```
let a=1, b=2;
[b,a] = [a,b]
console.log(a,b) // 2 1
```

Możemy użyć operatora reszty (`...`), aby wyodrębnić pozostałe elementy i przypisać je do tablicy. Przy destrukturyzacji operator reszty może być używany tylko jako ostatni operator:

```
const [x, ...y] = ['a', 'b', 'c']; // x='a'; y=['b', 'c']
```

Obiekty wbudowane

Wcześniej w tym rozdziale zetknęliśmy się już z konstruktorem `Object()`. Jest on zwracany przez obiekty literałowe, gdy sięgnie się do ich pola `constructor`. Funkcja ta jest jednym z konstruktorów wbudowanych. Takich konstruktorów jest więcej — poznasz je wszystkie w dalszej części rozdziału.

Obiekty wbudowane można podzielić na trzy kategorie:

- **Obiekty opakowujące**: `Object`, `Array`, `Function`, `Boolean`, `Number` i `String`. Odpowiadają one różnym typom danych JavaScriptu. Zasadniczo każda wartość zwracana przez operator `typeof` (omówiony w rozdziale 2.) posiada swój obiekt opakowujący. Wyjątkami są `"undefined"` i `"null"`.
- **Obiekty użytkowe**. Są to `Math`, `Date` i `RegExp` — warto jest je poznać, ponieważ często okazują się przydatne.
- **Obiekty błędów**, czyli obiekt `Error` oraz inne, bardziej szczegółowe obiekty, za pomocą których można przywrócić działanie programu po wystąpieniu nieoczekiwanej sytuacji.

W rozdziale omawiamy jedynie wybrane metody obiektów wbudowanych. Pełna lista znajduje się w dodatku C.

Jeśli nie widzisz różnicy pomiędzy wbudowanym obiektem a wbudowanym konstruktorem — nie martw się, w zasadzie są tym samym. Za chwilę wytłumaczymy, że funkcje, wśród nich również konstruktory, także są obiektami.

Object

`Object` jest rodzicem wszystkich obiektów w języku JavaScript — wszystkie inne obiekty z niego dziedziczą. W celu utworzenia nowego obiektu możesz skorzystać z notacji literałowej albo z konstruktora `Object()`. Następujące dwie linie są równoważne:

```
> var o = {};
> var o = new Object();
```

Jak wspomnieliśmy wcześniej, pusty obiekt nie jest zupełnie bezużyteczny, ponieważ już na starcie jest wyposażony w kilka odziedziczonych właściwości i metod:

- Właściwość `o.constructor` zwraca referencję do konstruktora.
- `o.toString()` to metoda, która zwraca tekstową reprezentację obiektu.
- `o.valueOf()` zwraca jednowartościową reprezentację obiektu, najczęściej sam obiekt.

Zobaczmy te metody w akcji. Najpierw utwórzmy obiekt:

```
> var o = new Object();
```

Wywołanie `toString()` zwróci tekstową reprezentację obiektu:

```
> o.toString();
"[object Object]"
```

Metoda `toString()` zostanie wewnętrznie wywołana przez JavaScript, jeśli obiekt zostanie użyty w kontekście łańcucha znaków. Przykładowo `alert()` działa jedynie na łańcuchach, dlatego jeśli zostanie jej przekazany obiekt, w tle zostanie wywołana metoda `toString()`. Poniższe dwie linie przyniosą ten sam efekt:

```
> alert(o);
> alert(o.toString());
```

Innym typem kontekstu tekstowego jest konkatenacja (złączanie) łańcuchów znaków. Jeśli podjęta zostanie próba połączenia obiektu z łańcuchem, najpierw wywołana będzie metoda `toString()` tego obiektu:

```
> "Obiekt: " + o
"Obiekt: [object Object]"
```

`valueOf()` to kolejna metoda, w którą wyposażone są wszystkie obiekty. W przypadku obiektów prostych, których konstruktorem jest `Object()`, `valueOf()` zwróci dany obiekt:

```
> o.valueOf() === o;
true
```

Podsumujmy:

- Obiekty można tworzyć za pomocą `var o = {};` (preferowana notacja literałowa) lub za pomocą `var o = new Object();`.
- Każdy, nawet najbardziej złożony obiekt dziedziczy z obiektu `Object` i dzięki temu posiada metody takie jak `toString()` i właściwości takie jak `constructor`.

Array

`Array()` to funkcja wbudowana, której można używać jako konstruktora do tworzenia tablic:

```
> var a = new Array();
```

Powyższy fragment kodu odpowiada następującemu zapisowi literałowemu:

```
> var a = [];
```

Niezależnie od tego, w jaki sposób została utworzona tablica, można dodawać do niej elementy w ten sam, znany nam sposób:

```
> a[0] = 1;

> a[1] = 2;

> a;
[1, 2]
```

Konstruktorowi `Array()` można przekazać wartości, które zostaną wstawione do tablicy jako jej elementy.

```
> var a = new Array(1, 2, 3, 'cztery');
> a;
[1, 2, 3, "cztery"]
```

Wyjątkiem jest zachowanie konstruktora, gdy jako argument przekażemy pojedynczą liczbę. Wówczas zostanie ona uznana za długość tablicy.

```
> var a2 = new Array(5);
> a2;
[undefined x 5]
```

Skoro tablice można tworzyć przy użyciu konstruktora, czy są one obiektami? Tak — można upewnić się za pomocą operatora `typeof`:

```
> typeof [1, 2, 3];
"object"
```

Ponieważ tablice są obiektami, dziedziczą właściwości i metody pochodzące od obiektu nadrzędnego:

```
> var a = [1, 2, 3, 'cztery'];
> a.toString();
"1,2,3,cztery"
> a.valueOf();
[1, 2, 3, "cztery"]
> a.constructor;
function Array() { [native code] }
```

Tablice są obiektami obdarzonymi pewnymi wyjątkowymi cechami:

- Ich właściwości są nazywane automatycznie za pomocą liczb od zera w górę.
- Posiadają właściwość `length`, która zawiera liczbę elementów tablicy.
- Poza metodami odziedziczonymi z `Object` posiadają własne metody wbudowane.

Przyjrzymy się różnicom pomiędzy tablicą a obiektem. Na początek utwórzmy pusty obiekt `o` i pustą tablicę `a`:

```
> var a = [], o = {};
```

Tablice zawsze posiadają właściwość `length` określającą ich długość, podczas gdy zwykłe obiekty nie:

```
> a.length
0
> typeof o.length;
"undefined"
```

Zarówno do tablic, jak i do obiektów można dodawać właściwości liczbowe i nieliczbowe:

```
> a[0] = 1;
> o[0] = 1;
> a.prop = 2;
> o.prop = 2;
```

Właściwość `length` zawsze przechowuje liczbę właściwości numerycznych, ignorując pozostałe.

```
> a.length;
1
```

Wartość właściwości `length` można zmieniać. Zwiększenie jej wartości powoduje dodanie do tablicy pustych elementów (o wartości `undefined`).

```
> a.length = 5;
5
> a;
[1, undefined x 4]
```

Zmniejszenie wartości właściwości `length` spowoduje usunięcie końcowych elementów.

```
> a.length = 2;
2
> a;
[1, undefined x 1]
```

Niektóre metody obiektu Array

Poza metodami odziedziczonymi z `Object` obiekty tablicowe posiadają własne przydatne metody, takie jak `sort()`, `join()` i `slice()` (pełna lista w dodatku C).

Poeksperymentujmy sobie z metodami tablic:

```
> var a = [3, 5, 1, 7, 'test'];
```

Metoda `push()` dodaje element na koniec tablicy, zaś `pop()` usuwa ostatni element. Wywołanie `a.push('new')` zadziała tak samo jak `a[a.length] = 'new'`, natomiast `a.pop()` odpowiada `a.length--`.

Metoda `push()` zwraca długość zmienionej tablicy, podczas gdy `pop()` zwraca usunięty element:

```
> a.push('new');
6
> a;
[3, 5, 1, 7, "test", "new"]
> a.pop();
"new"
> a;
[3, 5, 1, 7, "test"]
```

Metoda `sort()` sortuje elementy tablicy, a także zwraca wynik sortowania. W poniższym przykładzie po wywołaniu `sort()` zmienne `a` i `b` wskazują tę samą tablicę:

```
> var b = a.sort();
> b;
[1, 3, 5, 7, "test"]
> a === b;
true
```

Metoda `join()` zwraca łańcuch składający się z wartości elementów tablicy rozdzielonych łańcuchem przekazanym jako parametr:

```
> a.join(' to nie ');
"1 to nie 3 to nie 5 to nie 7 to nie test"
```

`slice()` zwraca wycinek tablicy bez wprowadzania modyfikacji do oryginalnego obiektu. Pierwszym parametrem jest indeks początkowy (indeks pierwszego elementu, który ma zostać zwrócony), drugim indeks końcowy — oba liczone od zera.

```
> b = a.slice(1, 3);
[3, 5]
> b = a.slice(0, 1);
[1]
> b = a.slice(0, 2);
[1, 3]
```

Oryginalna tablica nie uległa zmianie:

```
> a;
[1, 3, 5, 7, "test"]
```

Metoda `splice()` dla odmiany zmienia tablicę, na której jest wywoływana. Usuwa ona wycinek tablicy, zwraca go oraz opcjonalnie wypełnia powstałą lukę nowymi elementami. Pierwsze dwa parametry to indeksy początkowy i końcowy, pozostałe to nowe wartości.

```
> b = a.splice(1, 2, 100, 101, 102);
[3, 5]
> a;
[1, 100, 101, 102, 7, "test"]
```

Wypełnianie luki nowymi elementami nie jest obowiązkowe — można z niego zrezygnować:

```
> a.splice(1, 3);
[100, 101, 102]
> a;
[1, 7, "test"]
```

Metody tablic w ES6

W ES6 tablice otrzymują wiele użytecznych metod. Wcześniej to biblioteki takie jak **lodash** i **underscore** dostarczały funkcjonalności, których do tej pory brakowało w JavaScripcie. Dzięki nowym metodom pomocniczym tworzenie tablic i manipulowanie nimi jest znacznie wygodniejsze i łatwiejsze.

Array.From

Konwersja wartości tablicopodobnych na tablice zawsze stanowiła wyzwanie w JavaScripcie. Programiści do efektywnej obsługi tablic wykorzystywali różne sztuczki i pisali biblioteki.

ES6 wprowadza bardzo przydatną metodę przekształcania obiektów tablicopodobnych i wartości iterowalnych na tablice. Wartościami tablicopodobnymi są obiekty, które mają właściwość `length` i elementy indeksowane. Każda funkcja ma domyślną zmienną `arguments`, która zawiera listę wszystkich argumentów przekazanych do funkcji. Ta zmienna jest obiektem tablicopodobnym. Przed wprowadzeniem specyfikacji ES6 jedynym sposobem przekonwertowania obiektu `arguments` na tablicę było przejście przez jego elementy i skopiowanie wartości do nowej tablicy:

```
function toArray(args) {
    var result = [];
    for (var i = 0, len = args.length; i < len; i++) {
        result.push(args[i]);
    }
    return result;
}
function doSomething() {
    var args = toArray(arguments);
    console.log(args)
}
doSomething("witaj", "świecie")
// Array [
//   "witaj",
//   "świecie"
// ]
```

Tworzymy tutaj nową tablicę, aby skopiować do niej wszystkie elementy obiektu `arguments`. Jest to marnotrawstwo i wymaga dużo niepotrzebnego kodowania. `Array.from()` jest zwięzłą metodą konwersji obiektów tablicopodobnych na tablice. Możemy zmienić ten przykład w bardziej zwięzły przy użyciu `Array.from()`:

```
function doSomething() {
    console.log(Array.from(arguments))
}
doSomething("witaj", "świecie")
// Array [
//   "witaj",
//   "świecie"
// ]
```

Podczas wywoływania `Array.from()` możesz podać własny schemat mapowania, przekazując odpowiednią funkcję. Ta funkcja jest wywoływana na wszystkich elementach obiektu i konwertuje je. Jest to przydatna konstrukcja dla wielu typowych przypadków użycia, na przykład:

```
function doSomething() {
    console.log(Array.from(arguments, function(elem)
  { return elem + " zmapowany"; }));
}
```

W tym przykładzie dekonstruujemy obiekt `arguments` za pomocą `Array.from()` i dla każdego elementu w tym obiekcie wywołujemy określoną funkcję.

Tworzenie tablic za pomocą Array.of

Tworzenie tablicy za pomocą konstruktora `Array()` powoduje trochę problemów. Ten konstruktor zachowuje się odmiennie w zależności od liczby i typu argumentów. Kiedy do konstruktora `Array()` przekazywana jest pojedyncza wartość liczbowa, tworzona jest tablica niezdefiniowanych elementów, a jej długości przypisywana jest wartość argumentu:

```
let arr = new Array(2)
console.log(arr) // [undefined x 2]
console.log(arr.length) // 2
```

Z drugiej strony, jeśli przekażesz tylko jedną wartość nieliczbową, stanie się ona jedynym elementem w tablicy:

```
let arr = new Array("2")
console.log(arr) // ["2"]
console.log(arr.length) // 1
```

To nie wszystko. Jeśli przekażesz wiele wartości, stają się one elementami tablicy:

```
let arr = new Array(1,"2",{obj: "3"})
console.log(arr.length) // 3
```

Oczywiste jest, że musi istnieć lepszy sposób tworzenia tablic, aby uniknąć takiego zamieszania. ES6 wprowadza metodę `Array.of`, która działa jak konstruktor `Array()`, ale gwarantuje jedno standardowe zachowanie. `Array.of` tworzy tablicę z przekazanych jej argumentów, niezależnie od ich liczby i typu:

```
let arr = Array.of(1,"2",{obj: "3"})
console.log(arr.length) // 3
```

Metody Array.prototype

ES6 wprowadza kilka ciekawych metod obiektu tablicy. Te metody są pomocne przy iterowaniu przez tablicę i wyszukiwaniu elementów w tablicy — obie te operacje są bardzo powszechne i użyteczne.

Oto metody używane do iteracji przez tablice:

- `Array.prototype.entries()`,
- `Array.prototype.values()`,
- `Array.prorotype.keys()`.

Wszystkie trzy metody zwracają iterator. Ten iterator może być używany do tworzenia tablic za pomocą `Array.from()` i może być wykorzystywany w pętlach do iteracji:

```
let arr = ['a','b','c']
for (const index of arr.keys()){
  console.log(index) // 0 1 2
}
for (const value of arr.values()){
  console.log(value) // a b c
}
for (const [index,value] of arr.entries()){
  console.log(index,value)
}
// 0 "a"
// 1 "b"
// 2 "c"
```

Jak już wspomnieliśmy, istnieją też nowe metody do przeszukiwania tablic. Do tej pory szukanie elementu w tablicy zwykle wymagało iteracji przez całą listę i porównywania jej elementów z wartością — nie było do tego żadnych wbudowanych metod. Chociaż `indexOf()` i `lastIndexOf()` pomagały znaleźć pojedynczą wartość, nie było sposobu na znalezienie elementów na podstawie złożonych warunków. Poniższe wbudowane metody ES6 pozwalają na bardziej zawansowane przeszukiwanie tablic.

- `Array.prototype.find`,
- `Array.prototype.findIndex`.

Obie te metody przyjmują dwa argumenty. Pierwszym jest funkcja wywołania zwrotnego (która zawiera warunek predykatu), a drugim jest opcjonalny obiekt `thisArg`, który jest używany jako `this` podczas wykonywania funkcji wywołania zwrotnego. Funkcja wywołania zwrotnego przyjmuje trzy argumenty: element tablicy, indeks tego elementu i tablicę. Zwraca `true`, jeśli element pasuje do predykatu:

```
let numbers = [1,2,3,4,5,6,7,8,9,10];
console.log(numbers.find(n => n > 5)); // 6
console.log(numbers.findIndex(n => n > 5)); // 5
```

Function

Wiesz już, że funkcje są pewnym specjalnym typem danych. Okazuje się jednak, że są czymś więcej — są obiektami. Istnieje wbudowany konstruktor `Function()`, który pozwala tworzyć funkcje w odmienny od pokazanego wcześniej (aczkolwiek niezalecany) sposób.

Poniższe trzy sposoby definiowania funkcji są równoważne:

```
> function sum(a, b) { // deklaracja funkcji
    return a + b;
  };
> sum(1, 2);
3
> var sum = function(a, b) { // wyrażenie funkcyjne
    return a + b;
  };
> sum(1, 2);
3
> var sum = new Function('a', 'b', 'return a + b;');
> suma(1, 2);
3
```

Konstruktorowi `Function()` przekazuje się najpierw nazwy parametrów, a potem kod źródłowy ciała funkcji (wszystko jako łańcuch znaków). Do utworzenia funkcji konieczne jest przetworzenie kodu podanego w postaci tekstowej. Rozwiązanie to posiada wszystkie wady funkcji `eval()`, dlatego należy ograniczać stosowanie konstruktora `Function()`.

Jeśli funkcja tworzona za pomocą `Function()` ma wiele argumentów, można zapisać je wewnątrz jednego łańcucha znaków, oddzielając poszczególne parametry przecinkami. Następujące definicje są równoważne:

```
> var first = new Function(
    'a, b, c, d',
    'return arguments;'
  );
> first(1, 2, 3, 4);
[1, 2, 3, 4]
> var second = new Function(
    'a, b, c',
    'd',
    'return arguments;'
  );
> second(1, 2, 3, 4);
[1, 2, 3, 4]
> var third = new Function(
    'a',
    'b',
    'c',
    'd',
    'return arguments;'
  );
> third(1, 2, 3, 4);
[1, 2, 3, 4]
```

Nie używaj konstruktora `Function()`. Należy unikać wszelkich funkcji, które jako argument pobierają kod w postaci łańcucha znaków. Do tej samej grupy należą funkcje `setTimeout()` (która jeszcze nie pojawiła się w tej książce) oraz `eval()`.

Właściwości obiektu Function

Jak wszystkie inne obiekty, funkcje posiadają właściwość `constructor`, która zawiera referencję do konstruktora `Function()`. Dotyczy to wszystkich funkcji, bez względu na to, jaka składnia została użyta do ich utworzenia:

```
> function myfunc(a) {
    return a;
  }
> myfunc.constructor;
function Function() { [native code] }
```

Funkcje posiadają także właściwość `length`, która określa liczbę parametrów przyjmowanych przez funkcję.

```
> function myfunc(a, b, c) {
    return true;
  }
> myfunc.length;
3
```

Korzystanie z właściwości prototype

Jedną z najczęściej wykorzystywanych właściwości funkcji jest `prototype`. Omówimy ją dokładnie w następnym rozdziale, chwilowo wystarczy następujący zestaw faktów:

- Właściwość `prototype` obiektu `function` wskazuje inny obiekt.
- Ma ona znaczenie tylko, jeśli funkcja jest wywoływana jako konstruktor.
- Wszystkie obiekty utworzone za pomocą funkcji przechowują referencję do właściwości `prototype` i mogą korzystać z jej właściwości jak z własnych.

Krótka demonstracja właściwości `prototype`. Zacznijmy od prostego obiektu, który posiada właściwość `name` i metodę `say()`.

```
var ninja = {
  name: 'Ninja',
  say: function(){
    return 'Jestem ' + this.name;
  }
};
```

Możesz sprawdzić, że nawet pusta funkcja posiada właściwość `prototype`, która wskazuje nowy obiekt.

```
> function F(){}
> typeof F.prototype;
"object"
```

Jeśli zmienisz właściwość `prototype`, zacznie robić się ciekawie. Można dodawać do niej właściwości lub domyślny obiekt zamienić na dowolny inny obiekt. Przypiszmy zatem obiekt `ninja` do `prototype`.

```
> F.prototype = ninja;
```

Po tej zmianie, przy użyciu funkcji `F()` w roli konstruktora możesz utworzyć nowy obiekt `baby_ninja`, który będzie miał dostęp do właściwości `F.prototype` (wskazującej na `ninja`) jak do własnych.

```
> var baby_ninja = new F();
> baby_ninja.name;
"Ninja"
> baby_ninja.say();
"Jestem Ninja"
```

Więcej o właściwości `prototype` dowiesz się z następnego rozdziału.

Metody obiektu Function

Obiekty będące funkcjami również są potomkami obiektu `Object`, dlatego posiadają metody domyślne, takie jak `toString()`. `toString()` wywołana na obiekcie będącym funkcją zwróci jej kod źródłowy.

```
> function myfunc(a, b, c) {
    return a + b + c;
  }
> myfunc.toString();
"function myfunc(a, b, c) {
  return a + b + c;
}"
```

Jeśli spróbujesz zajrzeć do kodu źródłowego funkcji wbudowanych, otrzymasz zamiast ciała funkcji niezbyt przydatny łańcuch znaków `[native code]`:

```
> parseInt.toString();
"function parseInt() { [native code] }"
```

Jak widać, funkcji `toString()` można użyć do odróżnienia metod natywnych od tych, które zostały zdefiniowane przez programistów.

> Zachowanie funkcji `toString()` zależy od środowiska i w kwestiach odstępów i nowych linii jest odmienne w różnych przeglądarkach.

Metody call i apply

Ważnymi metodami obiektów funkcyjnych są `call()` i `apply()`. Można ich używać do wywoływania funkcji i przekazywania do nich dowolnych argumentów.

Dzięki nim obiekty mogą wypożyczać metody od innych obiektów i wywoływać je jak własne. Jest to prosty i skuteczny sposób wielokrotnego wykorzystywania kodu.

Załóżmy, że mamy obiekt `some_obj`, który posiada metodę `say()`:

```
var some_obj = {
  name: 'Ninja',
  say: function(who){
    return 'Siema ' + who + ', jestem ' + this.name;
  }
};
```

Możesz wywołać metodę `say()`, która sięga do `this.name` w celu pobrania wartości własnej właściwości.

```
> some_obj.say('stary');
"Siema stary, jestem Ninja"
```

Utwórzmy teraz prosty obiekt `my_obj`, który posiada jedynie właściwość `name`:

```
> var my_obj = {name: 'Guru programowania'};
```

Powiedzmy, że `my_obj` tak bardzo lubi metodę `say()` obiektu some_obj, że chce wywołać ją jako swoją własną. Jest to możliwe przy użyciu metody `call()` obiektu funkcyjnego `say()`:

```
> some_obj.say.call(my_obj, 'stary');
"Siema stary, jestem Guru programowania"
```

Działa! Co dokładnie tutaj zaszło? Wywołaliśmy metodę `call()` obiektu `say()`, przekazując jej dwa parametry: obiekt `my_obj` oraz łańcuch znaków `'stary'`. W wyniku tego podczas wywołania `say()` wszystkie referencje do wartości `this` wskazywały `my_obj`. Dzięki temu `this.imie` nie zwróciło `'Ninja'`, tylko `'Guru programowania'`.

Jeśli dana funkcja pobiera więcej argumentów, po prostu wymieniamy je podczas wywołania `call()`:

```
some_obj.someMethod.call(my_obj, 'a', 'b', 'c');
```

Jeśli jako pierwszy parametr `call()` nie zostanie przekazany obiekt lub jeśli przekaże się `null`, funkcja zostanie wywołana na rzecz obiektu globalnego.

Metoda `apply()` działa tak jak `call()`, z tą różnicą, że wszystkie parametry przekazywane metodzie innego obiektu umieszcza się w tablicy. Poniższe dwie linie są równoważne:

```
some_obj.someMethod.apply(my_obj, ['a', 'b', 'c']);
some_obj.someMethod.call(my_obj, 'a', 'b', 'c');
```

Kontynuując powyższy przykład, możesz spróbować wykonać następujący kod:

```
> some_obj.someMethod.apply(my_obj, ['stary']);
"Siema stary, jestem Guru programowania"
```

Nowe spojrzenie na obiekt arguments

W poprzednim rozdziale pokazaliśmy, jak z wnętrza funkcji uzyskać dostęp do zmiennej o nazwie `arguments`, która przechowuje wartości wszystkich parametrów użytych podczas wywołania funkcji:

```
> function f() {
    return arguments;
  }
> f(1, 2, 3);
[1, 2, 3]
```

`arguments` wygląda jak tablica, jednak w rzeczywistości jest to obiekt tablicopodobny. Przypomina tablicę, ponieważ posiada indeksowane elementy oraz właściwość `length`. Na tym jednak podobieństwa się kończą — arguments nie posiada metod tablicowych, takich jak `sort()` czy `slice()`.

Można jednak przekonwertować `arguments` na tablicę i korzystać z wszystkich jej dobrodziejstw. Można do tego celu wykorzystać nowo poznaną metodę `call()`:

```
> function f() {
    var args = [].slice.call(arguments);
    return args.reverse();
  }
> f(1, 2, 3, 4);
[4, 3, 2, 1]
```

Jak widać, można wypożyczyć metodę `slice()` za pomocą notacji `[].slice` lub używając bardziej rozwlekłej składni `Array.prototype.slice`.

Leksykalne this w funkcjach strzałkowych

W poprzednim rozdziale omówiliśmy szczegółowo funkcje strzałkowe ES6 i ich składnię. Istotnym aspektem funkcji strzałkowych jest to, że zachowują się inaczej niż zwykłe funkcje. Różnica jest subtelna, ale ważna. Funkcje strzałkowe nie mają własnej wartości `this`. Wartość `this` w funkcji strzałkowej jest dziedziczona z zakresu otaczającego (leksykalnego).

Funkcje mają specjalną zmienną this, która odnosi się do obiektu, za pośrednictwem którego metoda została wywołana. Ponieważ wartość `this` jest nadawana dynamicznie na podstawie wywołania funkcji, mówimy czasem o dynamicznym `this`. Funkcja jest wykonywana w dwóch zakresach — leksykalnym i dynamicznym. Zakres leksykalny jest zakresem otaczającym zakres funkcji, a zakresem dynamicznym jest zakres, który wywołał funkcję (zazwyczaj obiekt).

W języku JavaScript tradycyjne funkcje odgrywają kilka ról. Wśród nich możemy wyróżnić funkcje niebędące metodami (czyli podprogramy lub po prostu funkcje), metody (składowe obiektu) i konstruktory. Kiedy funkcje pełnią rolę podprogramu, istnieje niewielki problem spowodowany dynamicznym `this`. Ponieważ podprogramy nie są wywoływane na obiekcie, w trybie ścisłym wartość `this` nie jest zdefiniowana i ustawiana jest na zakres globalny. To utrudnia pisanie wywołań zwrotnych. Rozważmy następujący przykład:

```
var greeter = {
  default: "Witaj, ",
  greet: function (names){
    names.forEach(function(name) {
      console.log(this.default + name); // nie można odczytać właściwości 'default'
                                        // wartości undefined
    })
  }
}
console.log(greeter.greet(['świecie', 'niebo']))
```

Przekazujemy podprogram do funkcji `forEach()` wywoływanej na tablicy `names`. Ten podprogram ma niezdefiniowaną wartość `this` i niestety nie ma dostępu do wartości `this` zewnętrznej metody `greet`. Oczywiste jest, że ten podprogram potrzebuje leksykalnego `this`, pochodzącego z otaczającego zakresu metody `greet`. Tradycyjnie, aby usunąć to ograniczenie, przypisujemy leksykalne `this` do zmiennej, która jest dostępna dla podprogramu poprzez domknięcie.

Możemy poprawić poprzedni przykład w następujący sposób:

```
var greeter = {
  default: "Witaj, ",
  greet: function (names){
    let that = this
    names.forEach(function(name) {
      console.log(that.default + name);
    })
  }
}
console.log(greeter.greet(['świecie', 'niebo']))
```

Jest to rozsądna sztuczka, która pozwala zasymulować leksykalne `this`. Jednak problem z takimi sztuczkami polega na tym, że generują za dużo „szumów" dla osoby piszącej lub przeglądającej ten kod. Trzeba dobrze zrozumieć dziwaczne zachowanie wartości `this`. Nawet jeśli Ci się to uda, będziesz musiał stale śledzić takie sztuczki w swoim kodzie.

Funkcje strzałkowe mają leksykalne `this` i nie wymagają takich sztuczek. Z tego względu lepiej sprawdzają się jako podprogramy. Możemy przekształcić poprzedni przykład, aby użyć leksykalnego `this` oraz funkcji strzałkowej:

```
var greeter = {
  default: "Witaj, ",
  greet: function (names){
```

```
    names.forEach(name=> {
      console.log(this.default + name); // leksykalne 'this' dostępne dla podprogramu
    })
  }
}
console.log(greeter.greet(['świecie', 'niebo']))
```

Inferencja typów obiektów

Załóżmy, że mamy tablicopodobny obiekt `arguments`, który wygląda zupełnie jak obiekt tablicowy. Jak można wiarygodnie rozróżnić te dwa obiekty? Ponadto operator `typeof` informuje, że typ to `"object"`, gdy jest używany z tablicami. W związku z tym jak możemy odróżnić obiekt od tablicy?

Panaceum jest metoda `toString()` obiektu `Object`. Zwraca ona nazwę wewnętrznej klasy użytej do utworzenia danego obiektu:

```
> Object.prototype.toString.call({});
"[object Object]"
> Object.prototype.toString.call([]);
"[object Array]"
```

Musimy wywołać oryginalną metodę `toString()` zdefiniowaną w prototypie konstruktora `Object`. W przeciwnym razie, jeśli wywołamy `toString()` funkcji `Array`, otrzymamy inny wynik, ponieważ zostanie on nadpisany dla konkretnych celów obiektów tablicowych:

```
> [1, 2, 3].toString();
"1,2,3"
```

Poprzedni kod jest równoważny z tym:

```
> Array.prototype.toString.call([1, 2, 3]);
"1,2,3"
```

Pobawmy się jeszcze trochę z `toString()`. Zdefiniujmy przydatną referencję, aby potem zaoszczędzić sobie pisania:

```
> var toStr = Object.prototype.toString;
```

Poniższy przykład pokazuje, jak możemy odróżnić tablicę od tablicopodobnego obiektu `arguments`:

```
> (function () {
    return toStr.call(arguments);
  }());
"[object Arguments]"
```

Możemy nawet sprawdzać elementy DOM:

```
> toStr.call(document.body);
"[object HTMLBodyElement]"
```

Boolean

Kontynuując naszą podróż przez wbudowane obiekty JavaScriptu, dochodzimy do mało skomplikowanej grupy, w której skład wchodzą obiekty opakowujące proste typy danych (typ logiczny boolean, liczba, łańcuch znaków).

Typ boolean po raz pierwszy pojawił się w rozdziale 2. Teraz przyszła pora na spotkanie z konstruktorem `Boolean()`:

```
> var b = new Boolean();
```

Zmiennej `b` zostanie przypisany nowy obiekt, a nie prosta wartość typu boolean. Do właściwej wartości przechowywanej wewnątrz obiektu można dostać się za pomocą metody `valueOf()`, odziedziczonej z `Object`.

```
> var b = new Boolean();
> typeof b;
"object"
> typeof b.valueOf();
"boolean"
> b.valueOf();
false
```

Obiekty utworzone za pomocą konstruktora `Boolean()` nie są specjalnie przydatne, jako że obiekt ten zawiera jedynie odziedziczone metody.

Inaczej ma się sprawa z `Boolean()` wywoływanym jako normalna funkcja, a nie konstruktor — czyli bez użycia `new`. W ten sposób można zamienić na typ logiczny wartość należącą do innego typu danych (odpowiada to zastosowaniu na zmiennej podwójnej negacji, jak w przypadku `!!wartość`).

```
> Boolean("test");
true
> Boolean("");
false
> Boolean({});
true
```

Poza sześcioma fałszywymi wartościami wszystko, w tym obiekty puste, zostanie uznane za prawdziwe. Oznacza to także, że wszystkie obiekty utworzone za pomocą konstruktora `Boolean()` zwrócą wartość **true**, ponieważ są obiektami:

```
> Boolean(new Boolean(false));
true
```

Może to być dezorientujące, a skoro obiekty Boolean nie oferują żadnych specjalnych metod, najlepiej trzymać się zwykłych typów prostych `boolean`.

Number

Funkcji `Number()` używa się podobnie jak `Boolean()`:

- jako konstruktora, za pomocą którego (i przy użyciu operatora `new`) tworzymy nowe obiekty;
- jako normalnej funkcji, zamieniającej dowolną wartość na liczbę (podobnej do `parseInt()` i `parseFloat()`):

```
> var n = Number('12.12');
> n;
12.12
> typeof n;
"number"
> var n = new Number('12.12');
> typeof n;
"object"
```

Skoro funkcje są obiektami, mogą posiadać właściwości. Funkcja `Number()` ma kilka stałych właściwości wbudowanych (których wartości nie można zmieniać):

```
> Number.MAX_VALUE;
1.7976931348623157e+308
> Number.MIN_VALUE;
5e-324
> Number.POSITIVE_INFINITY;
Infinity
> Number.NEGATIVE_INFINITY;
-Infinity
> Number.NaN;
NaN
```

Obiekt liczbowy posiada trzy metody: `toFixed()`, `toPrecision()` i `toExponential()` (szczegóły w dodatku C).

```
> var n = new Number(123.456);
> n.toFixed(1)
"123.5"
```

Warto wiedzieć, że do korzystania z tych metod nie jest konieczne jawne utworzenie obiektu liczbowego. W takim przypadku obiekt `Number` jest tworzony automatycznie i niszczony po wykonaniu obliczeń:

```
> (12345).toExponential();
"1.2345e+4"
```

Jak wszystkie inne obiekty, obiekty liczbowe również posiadają metodę `toString()`. W ich wypadku można jednak podać opcjonalny drugi parametr, określający podstawę (domyślnie 10).

```
> var n = new Number(255);
> n.toString();
"255"
> n.toString(10);
"255"
> n.toString(16);
"ff"
> (3).toString(2);
"11"
> (3).toString(10);
"3"
```

String

Konstruktor `String()` służy do tworzenia obiektów przechowujących łańcuchy znaków. Oferują one szereg metod ułatwiających przetwarzanie tekstu.

Poniższy przykład ilustruje różnicę pomiędzy obiektem a prostym typem danych przechowującym tekst.

```
> var primitive = 'Witaj!';
> typeof primitive;
"string"
> var obj = new String('świecie');
> typeof obj;
"object"
```

Obiekt przechowujący łańcuch jest bardzo podobny do tablicy znaków: umożliwia odwoływanie się do poszczególnych pól za pomocą indeksów (zostało to wprowadzone w ES5, ale było już od dawna obsługiwane przez wiele przeglądarek poza starszymi wersjami IE) i posiada właściwość `length`, określającą długość łańcucha:

```
> obj[0];
"ś"
> obj[4];
"c"
> obj.length;
7
```

Łańcuch znaków w postaci prostego typu danych można wydobyć z obiektu `String` za pomocą metod `valueOf()` lub `toString()`, odziedziczonych z `Object`. Prawdopodobnie nigdy nie skorzystasz z tego sposobu, ponieważ metoda `toString()` jest automatycznie wywoływana za każdym razem, gdy obiekt zostanie użyty w kontekście tekstowym.

```
> obj.valueOf();
"świecie"
> obj.toString();
"świecie"
> obj + "";
"świecie"
```

Sam łańcuch znaków nie jest obiektem i nie posiada żadnych właściwości ani metod. JavaScript pozwala jednak traktować proste łańcuchy jak obiekty (widziałeś to już w przypadku prostych typów liczbowych).

W poniższym przykładzie obiekt `String` jest automatycznie tworzony (a następnie niszczony) za każdym razem, gdy traktujemy prosty łańcuch znaków jak obiekt:

```
> "ziemniak".length;
8
> "pomidor"[0];
"p"
> "ziemniaki"["ziemniaki".length - 1];
"i"
```

Ostatni przykład prezentujący różnicę pomiędzy obiektem przechowującym łańcuch znaków a łańcuchem będącym prostym typem danych: zamieńmy oba na `boolean`. Pusty łańcuch jest wartością fałszywą, natomiast wszystkie obiekty zostaną zamienione na `true`.

```
> Boolean("");
false
> Boolean(new String(""));
true
```

Podobnie jak w przypadku `Number()` i `Boolean()`, funkcja `String()` użyta bez operatora `new` zamieni parametr na typ prosty.

```
> String(1);
"1"
```

Jeśli wartość wejściowa będzie obiektem, zostanie wywołana funkcja `toString()`.

```
> String({p: 1});
"[object Object]"
> String([1,2,3]);
"1,2,3"
> String([1, 2, 3]) === [1, 2, 3].toString();
true
```

Niektóre metody obiektu String

Poeksperymentujmy sobie z metodami, które można wywołać na obiektach reprezentujących łańcuchy znaków (pełna lista tych metod znajduje się w dodatku C).

Zacznijmy od utworzenia obiektu:

```
> var s = new String("Guru programowania");
```

Metody `toUpperCase()` i `toLowerCase()` pozwalają zmienić wielkość liter:

```
> s.toUpperCase();
"GURU PROGRAMOWANIA"
> s.toLowerCase();
"guru programowania"
```

Metoda `charAt()` zwraca znak znajdujący się na określonej pozycji (ten sam efekt da użycie nawiasu kwadratowego, ponieważ obiekt przechowujący łańcuch znaków przypomina tablicę znaków).

```
> s.charAt(0);
"G"
> s[0];
"G"
```

Odwołanie się do nieistniejącej pozycji zwróci pusty łańcuch:

```
> s.charAt(101);
""
```

Metoda `indexof()` pozwala na przeszukiwanie łańcuchów. Jeśli istnieje chociaż jedno dopasowanie, zwracana jest pozycja pierwszego z nich. Pozycje znaków liczone są od 0. Trzecim znakiem w wyrazie `"Guru"` (czyli znakiem na pozycji 2.) jest `"r"`.

```
> s.indexOf('r');
2
```

Można również określić pozycję, od której ma się rozpocząć wyszukiwanie. Poniższy fragment kodu znajdzie drugie wystąpienie litery `"r"`, ponieważ rozpocznie sprawdzanie łańcucha od pozycji 5.:

```
> s.indexOf('r', 5);
6
```

Metoda `lastIndexOf()` rozpoczyna wyszukiwanie od końca łańcucha (co nie zmienia faktu, że pozycja dopasowania jest liczona od początku łańcucha):

```
> s.lastIndexOf('r');
9
```

Podczas wyszukiwania rozróżniane są wielkie i małe litery. Możliwe jest wyszukiwanie całych łańcuchów, a nie tylko znaków:

```
> s.indexOf('Guru');
0
```

Jeśli fragment nie zostanie znaleziony, zwracana jest wartość `-1`:

```
> s.indexOf('guru');
-1
```

Jeśli chcesz pominąć kwestię wielkości liter, możesz przed rozpoczęciem wyszukiwania zamienić wszystkie litery na małe:

```
> s.toLowerCase().indexOf('guru');
0
```

Wartość 0 oznacza, że wynik wyszukiwania znajduje się na początku łańcucha. Może to prowadzić do nieporozumień podczas korzystania z if, ponieważ wartości 0 odpowiada wartość logiczna false. Prowadzi to do zachowań nieco sprzecznych z logiką:

```
if (s.indexOf('Guru')) {...}
```

Bezpiecznym sposobem sprawdzenia, czy tekst zawiera inny tekst, jest porównanie wyniku zwracanego przez indexOf() z liczbą -1.

```
if (s.indexOf('Guru') !== -1) {...}
```

Metody slice() i substring() zwracają fragment łańcucha ograniczony za pomocą argumentów rozumianych jako pozycje początkowa i końcowa.

```
 > s.slice(5, 12);
"program"
> s.substring(5, 12);
"program"
```

Należy pamiętać, że drugi parametr to pozycja końcowa, a nie długość fragment łańcucha. Pokazane powyżej dwie metody różnią się sposobem interpretacji argumentów ujemnych. substring() potraktuje je jak zera, podczas gdy slice() doda je do długości łańcucha. Zatem przekazanie wartości (1, -1) zostanie zrozumiane jako substring(1, 0) i slice(1, s.length-1):

```
> s.slice(1, -1);
"uru programowani"
> s.substring(1, -1);
"G"
```

Istnieje jeszcze nieznormalizowana metoda substr(), ale należy jej unikać i stosować raczej substring().

Metoda split() zamienia obiekt na tablicę, traktując parametr jako wartość rozdzielającą:

```
> s.split(" ");
["Guru", "programowania"]
```

Przeciwieństwem split() jest metoda join(), która zamienia tablicę w obiekt typu String:

```
> s.split(' ').join(' ');
"Guru programowania"
```

Metoda concat() skleja łańcuchy, podobnie jak operator + dla typu prostego:

```
> s.concat(" w języku JavaScript");
"Guru programowania w języku JavaScript"
```

Zwróć uwagę na to, że wszystkie omawiane powyżej metody związane z łańcuchami zwracają wartości proste i nie modyfikują obiektu źródłowego. Po wywołaniu wszystkich tych metod łańcuch znaków przechowywany przez obiekt nie uległ zmianie:

```
> s.valueOf();
"Guru programowania"
```

Do przeszukiwania łańcuchów używaliśmy `indexOf()` oraz `lastIndexOf()`. Istnieją bardziej zaawansowane metody (`search()`, `match()` i `replace()`), które jako parametry pobierają wyrażenia regularne. Opowiemy o nich później, gdy dojdziemy do konstruktora `RegExp()`.

To już wszystkie obiekty opakowujące dane. Zajmiemy się teraz użytkowymi obiektami `Math`, `Date` i `RegExp`.

Math

`Math` różni się nieco od innych wbudowanych obiektów globalnych. Nie jest funkcją i w związku z tym nie może być używany do tworzenia nowych obiektów. `Math` to wbudowany obiekt globalny, który dostarcza szeregu metod i właściwości ułatwiających wykonywanie operacji matematycznych.

Właściwości i metody `Math` są stałymi — nie można zmieniać ich wartości. Ich nazwy pisane są wielkimi literami w celu odróżnienia ich od zwykłych właściwości będących zmiennymi (podobnie jak właściwości stałe konstruktora `Number()`). Przyjrzyjmy się niektórym spośród tych stałych:

Liczba π:

```
> Math.PI;
3.141592653589793
```

Pierwiastek kwadratowy z 2:

```
> Math.SQRT2;
1.4142135623730951
```

Liczba Eulera e:

```
> Math.E;
2.718281828459045
```

Logarytm naturalny z 2:

```
> Math.LN2;
0.6931471805599453
```

Logarytm naturalny z 10:

```
> Math.LN10;
2.302585092994046
```

Nareszcie wiesz, jak zaimponować przyjaciołom, gdy któryś z nich (nieważne z jakiego dziwnego powodu) zacznie się zastanawiać, jaka jest wartość liczby *e* — wystarczy, że wpiszesz w konsoli `Math.E`, i od razu otrzymasz odpowiedź.

Popatrzmy teraz na metody obiektu `Math` (ich pełna lista znajduje się w dodatku C).

Losowanie liczb:

```
> Math.random();
0.3649461670235814
```

Metoda `random()` zwraca liczbę pomiędzy 0 a 1. Jeśli potrzebna jest Ci liczba z przedziału od 0 do 100, możesz wykonać następującą operację:

```
> 100 * Math.random();
```

W celu otrzymania liczb z przedziału od wartości min do wartości max najlepiej skorzystać z formuły `((max - min) * Math.random()) + min`. Przykładowo liczbę z przedziału od 2 do 10 losujemy w następujący sposób:

```
> 8 * Math.random() + 2;
9.175650496668485
```

Jeśli potrzebna jest Ci liczba całkowita, możesz skorzystać z jednej z metod zaokrąglających:

- `floor()` zaokrągla w dół,
- `ceil()` zaokrągla w górę,
- `round()` zaokrągla do najbliższej wartości.

Jeśli wynikiem ma być 0 albo 1, stosujemy:

```
> Math.round(Math.random());
```

Najwyższą lub najniższą liczbę ze zbioru wyznaczamy za pomocą metod `min()` i `max()`. Jeśli na stronie internetowej umieścimy formularz, w którym użytkownik powinien podać liczbę odpowiadającą miesiącowi, możemy w następujący sposób upewnić się, że wartość jest poprawna:

```
> Math.min(Math.max(1, input), 12);
```

Obiekt `Math` umożliwia wykonywanie działań matematycznych, które nie posiadają własnych operatorów. Liczby można podnosić do dowolnej potęgi za pomocą metody `pow()`, do wyznaczania pierwiastka kwadratowego służy metoda `sqrt()`, wartości trygonometryczne wyznaczają `sin()`, `cos()`, `atan()` itp.

2 do potęgi 8:

```
> Math.pow(2, 8);
256
```

Pierwiastek kwadratowy z 9:

```
> Math.sqrt(9);
3
```

Date

Konstruktor `Date()` tworzy obiekty reprezentujące daty. Jako argument funkcja ta może pobierać:

- nic (domyślnie zostanie ustawiona aktualna data);
- tekst, który da się przetłumaczyć na datę;
- osobne wartości określające dzień, miesiąc, godzinę itd.;
- znacznik czasu.

Obiekt, któremu przypisana zostanie aktualna data i godzina (zgodnie z ustawieniami strefy czasowej przeglądarki):

```
> new Date();
Sat Aug 12 2017 18:45:51 GMT+0200 (Środkowoeuropejski czas letni)
```

Jak zwykle w przypadku obiektów konsola wyświetla wynik wywołania metody `toString()` na obiekcie `Date`, dlatego otrzymujemy ten długi łańcuch znaków jako reprezentację obiektu daty.

Poniżej przedstawiamy kilka przykładów użycia łańcuchów znaków do inicjowania obiektu przechowującego datę. Można wybierać spomiędzy wielu różnych formatów:

```
> new Date('2015 11 12');
Thu Nov 12 2015 00:00:00 GMT+0100 (Środkowoeuropejski czas stand.)
> new Date('1 1 2016');
Fri Jan 01 2016 00:00:00 GMT+0100 (Środkowoeuropejski czas stand.)
> new Date('1 mar 2016 5:30');
Tue Mar 01 2016 05:30:00 GMT+0100 (Środkowoeuropejski czas stand.)
```

JavaScript potrafi odczytać datę z łańcuchów znaków w różnym formacie, jednak nie jest to najskuteczniejszy sposób precyzyjnego definiowania daty. Lepiej jest przekazać konstruktorowi wartości liczbowe określające następujące parametry:

- rok;
- miesiąc: od 0 (styczeń) do 11 (grudzień);
- dzień: od 1 do 31;
- godzina: od 0 do 23;
- minuty: od 0 do 59;
- sekundy: od 0 do 59;
- milisekundy: od 0 do 999.

Spójrzmy na przykłady.

Podanie wszystkich wspomnianych parametrów:

```
> new Date(2015, 0, 1, 17, 05, 03, 120);
Thu Jan 01 2015 17:05:03 GMT+0100 (Środkowoeuropejski czas stand.)
```

Podanie tylko daty i godziny:

```
> new Date(2015, 0, 1, 17);
Thu Jan 01 2015 17:00:00 GMT+0100 (Środkowoeuropejski czas stand.)
```

Postaraj się zapamiętać, że miesiące liczone są od 0, zatem 1 oznacza luty:

```
> new Date(2016, 1, 28);
Sun Feb 28 2016 00:00:00 GMT+0100 (Środkowoeuropejski czas stand.)
```

Jeśli podasz zbyt dużą wartość, zostanie ona przetłumaczona na odpowiednią datę w przyszłości. Przykładowo ponieważ w roku 2016 nie było dnia 30 lutego, taka wartość zostanie przetłumaczona na 1 marca (2016 był rokiem przestępnym).

```
> new Date(2016, 1, 29);
Mon Feb 29 2016 00:00:00 GMT+0100 (Środkowoeuropejski czas stand.)
> new Date(2016, 1, 30);
Tue Mar 01 2016 00:00:00 GMT+0100 (Środkowoeuropejski czas stand.)
```

W analogiczny sposób 32 grudnia zostanie zamieniony na 1 stycznia następnego roku:

```
> new Date(2012, 11, 31);
Mon Dec 31 2012 00:00:00 GMT+0100 (Środkowoeuropejski czas stand.)
> new Date(2012, 11, 32);
Tue Jan 01 2013 00:00:00 GMT+0100 (Środkowoeuropejski czas stand.)
```

Obiekt reprezentujący datę można jeszcze zainicjować za pomocą znacznika czasu, czyli liczby milisekund od początku ery Uniksa (gdzie 0 milisekund oznacza 1 stycznia 1970).

```
> new Date(1357027200000);
Tue Jan 01 2013 09:00:00 GMT+0100 (Środkowoeuropejski czas stand.)
```

Jeśli funkcja `Date()` zostanie wywołana bez operatora `new`, zwróci łańcuch znaków reprezentujący bieżącą datę, niezależnie od tego, czy podano jakiekolwiek parametry. Poniższe wywołania zwracają aktualną (w chwili uruchomienia kodu) datę.

```
> Date();
"Sat Aug 12 2017 19:02:34 GMT+0200 (Środkowoeuropejski czas letni)"
> Date(1, 2, 3, "bez znaczenia");
"Sat Aug 12 2017 19:03:02 GMT+0200 (Środkowoeuropejski czas letni)"
> typeof Date();
"string"
> typeof new Date();
"object"
```

Metody działające na obiektach Date

Istnieje wiele metod, które można wywołać na obiektach `Date`. Większość z nich to metody dostępowe `set*()` (ustawiające wartość atrybutu) lub `get*()` (pobierające wartość). Mamy na przykład `getMonth()` (pobierz miesiąc), `setMonth()` (ustaw miesiąc), `getHours()` (pobierz godzinę), `setHours()` (ustaw godzinę) itd.

Utwórzmy obiekt:

```
> var d = new Date(2015, 1, 1);
> d.toString();
"Sun Feb 01 2015 00:00:00 GMT+0100 (Środkowoeuropejski czas stand.)"
```

Ustawienie miesiąca na marzec (miesiące liczymy od 0):

```
> d.setMonth(2);
1425164400000
> d.toString();
"Sun Mar 01 2015 00:00:00 GMT+0100 (Środkowoeuropejski czas stand.)"
```

Pobranie miesiąca:

```
> d.getMonth();
2
```

Poza metodami należącymi do instancji obiektu `Date` istnieją jeszcze dwie (plus jeszcze jedna dodana w ES5) metody będące właściwościami funkcji (obiektu) `Date()`. Nie potrzebują one obiektu `date` — działają tak jak metody obiektu `Math`. W językach, w których istnieje pojęcie klasy, tego typu metody nazywa się statycznymi, ponieważ nie wymagają instancji.

Metoda `Date.parse()` przyjmuje łańcuch znaków i zwraca znacznik czasu:

```
> Date.parse('Jan 11, 2018');
1515625200000
```

`Date.UTC()` pobiera parametry określające rok, miesiąc, dzień itd. i zwraca znacznik **czasu uniwersalnego** (ang. *Universal Time* — UT):

```
> Date.UTC(2018, 0, 11);
1515628800000
```

Skoro konstruktor `new Date()` przyjmuje znaczniki czasu, można przekazać mu wynik wywołania `Date.UTC`. Poniższy przykład dowodzi, że `UTC()` podaje czas uniwersalny, a `Date()` czas lokalny:

```
> new Date(Date.UTC(2018, 0, 11));
Thu Jan 11 2018 01:00:00 GMT+0100 (Środkowoeuropejski czas stand.)
> new Date(2018, 0, 11);
Thu Jan 11 2018 00:00:00 GMT+0100 (Środkowoeuropejski czas stand.)
```

W ES5 do konstruktora `Date()` dodana została metoda `now()`, która zwraca bieżący znacznik czasu. Zapewnia ona wygodniejszy sposób pobierania znacznika czasu bez konieczności użycia metody `getTime()` na obiekcie `Date`, tak jak zrobilibyśmy to w ES3:

```
> Date.now();
1362038353044
> Date.now() === new Date().getTime();
true
```

Wewnętrzną reprezentację daty możemy potraktować jako znacznik czasu w postaci liczby całkowitej, a wszystkie pozostałe metody jako lukier składniowy. Dlatego ma sens, że `valueOf()` zwraca znacznik czasu:

```
> new Date().valueOf();
1362418306432
```

Możemy również przekonwertować datę na liczbę całkowitą za pomocą operatora `+`:

```
> +new Date();
1362418318311
```

Obliczanie dnia urodzin

Spójrzmy jeszcze na ostatni przykład działań na obiekcie typu `Date`. Ciekawiło mnie, w jaki dzień tygodnia wypadną moje urodziny w roku 2016:

```
> var d = new Date(2016, 5, 20);
> d.getDay();
1
```

Dni tygodnia liczymy od 0 (niedziela), zatem 1 powinno oznaczać poniedziałek. Sprawdźmy:

```
> d.toDateString();
"Mon Jun 20 2016"
```

Zgadza się, będzie to poniedziałek (ang. *Monday*) — nie jest to najlepszy dzień na imprezę. W takim razie napiszę sobie pętlę, która obliczy, ile razy na przestrzeni lat od 2016 do 3016 dzień 20 czerwca wypadnie w piątek. Albo, jeszcze lepiej, sprawdzę, jak rozkładają się dni tygodnia. Zakładając obecne tempo rozwoju medycyny, w roku 3016 będziemy mogli wspólnie napić się szampana.

Najpierw zainicjujmy tablicę z siedmioma elementami, po jednym dla każdego dnia tygodnia. Wykorzystamy elementy jako liczniki, które będziemy zwiększać w miarę zbliżania się pętli do roku 3016.

```
var stats = [0,0,0,0,0,0,0];
```

Pętla:

```
for (var i = 2016; i < 3016; i++) {
  stats[new Date(i, 5, 20).getDay()]++;
}
```

Wynik:

```
> stats;
[140, 146, 140, 145, 142, 142, 145]
```

142 piątki i 145 sobót. Tak jest!

RegExp

Wyrażenia regularne (ang. *regular expressions*) są niezwykle potężnym mechanizmem przeszukiwania i edycji tekstu. Różne języki stosują różne implementacje wyrażeń regularnych (możesz o nich myśleć jako o dialektach). JavaScript stosuje składnię odpowiadającą językowi Perl 5.

Wyrażenia regularne w skrócie określa się mianem „regex" lub „regexp".

Wyrażenie regularne składa się z:

- wzorca, do którego ma zostać dopasowany tekst;
- nieobowiązkowych modyfikatorów (nazywanych także flagami), które wpływają na sposób stosowania wzorca.

Wzorzec może być zwykłym fragmentem tekstu, który ma zostać dokładnie dopasowany, ale takie zastosowania wyrażeń regularnych spotyka się rzadko, tym bardziej że do osiągnięcia tego celu wystarczy zastosować `indexOf()`. W większości przypadków wzorzec jest dość złożony i czasem trudny do zrozumienia. Opanowanie wzorców wyrażeń regularnych nie jest proste, dlatego nie będziemy omawiać ich szczegółowo. Zamiast tego pokażemy, jak składnia, obiekty i metody JavaScriptu ułatwiają korzystanie z wyrażeń regularnych. Dodatkowe informacje na temat wzorców można znaleźć w dodatku D.

Konstruktor `RegExp()` pozwala tworzyć obiekty reprezentujące wyrażenia regularne.

```
> var re = new RegExp("j.*t");
```

Obiekty te można w nieco wygodniejszy sposób tworzyć za pomocą literałów:

```
> var re = /j.*t/;
```

`j.*t` w powyższym przykładzie jest wzorcem wyrażenia regularnego. Oznacza „znajdź takie łańcuchy, które zaczynają się od `j` i kończą się na `t`, a pomiędzy nimi występuje 0 lub więcej dowolnych znaków". Kropka (`.`) oznacza dowolny znak, gwiazdka (`*`) oznacza „zero lub więcej wystąpień znaków poprzedzających gwiazdkę". Jeśli wzorzec jest przekazywany konstruktorowi `RegExp()`, należy umieścić go w cudzysłowie.

Właściwości obiektów RegExp

Obiekty reprezentujące wyrażenia regularne posiadają następujące właściwości:

- `global`: jeśli ta właściwość ma wartość `false` (domyślną), w wyniku wyszukiwania zwrócony zostanie tylko pierwszy odnaleziony wynik. Jeśli chcesz otrzymać wszystkie dopasowania, zmień wartość właściwości na `true`.
- `ignoreCase`: `true` oznacza, że nie są rozróżniane wielkie i małe litery. Wartość domyślna tej właściwości to `false`.
- `multiline`: wartość `true` pozwala na wyszukiwanie dopasowań, które zajmują więcej niż jedną linię. Wartością domyślną jest `false`.

- `lastIndex`: pozycja, od której ma się rozpocząć wyszukiwanie — domyślnie 0.
- `source`: przechowuje wzorzec wyrażenia regularnego.

Właściwość `lastIndex` jest jedyną, której wartość można zmieniać po utworzeniu obiektu.

Pierwsze trzy parametry to modyfikatory wyrażenia regularnego. Podczas tworzenia obiektu wyrażenia za pomocą konstruktora można jako drugi parametr przekazać dowolną kombinację poniższych znaków:

- `"g"` dla `global`,
- `"i"` dla `ignoreCase`,
- `"m"` dla `multiline`.

Kolejność liter nie ma znaczenia. Przekazanie danej litery powoduje ustawienie wartości powiązanego z nią modyfikatora na `true`. W poniższym przykładzie wszystkie modyfikatory otrzymują wartość `true`:

```
> var re = new RegExp('j.*t', 'gmi');
```

Sprawdźmy:

```
> re.global;
true
```

Po utworzeniu obiektu nie można już zmienić wartości modyfikatora:

```
> re.global = false;
> re.global;
true
```

Jeśli obiekt jest tworzony za pomocą literału, modyfikatory podaje się po końcowym ukośniku.

```
> var re = /j.*t/ig;
> re.global;
true
```

Metody obiektów RegExp

Obiekty `RegExp` oferują dwie metody służące do znajdowania fragmentów tekstu: `test()` i `exec()`. Obie przyjmują parametr tekstowy. `test()` zwraca wartość logiczną (`true`, jeśli znaleziono dopasowanie, i `false` w przeciwnym razie), natomiast `exec()` — tablicę dopasowanych łańcuchów znaków. Oczywiście `exec()` wykonuje bardziej skomplikowane obliczenia, dlatego o ile nie są Ci potrzebne konkretne dopasowania, korzystaj z `test()`. Najczęstsze zastosowanie wyrażeń regularnych to walidacja formularzy — do tego `test()` w zupełności wystarczy.

Brak dopasowania z powodu różnicy w wielkości liter:

```
> /j.*t/.test("Javascript");
false
```

Ustawienie wartości ignoreCase na true powoduje, że uda się odnaleźć pasujący tekst:

```
> /j.*t/i.test("Javascript");
true
```

To samo zapytanie zadane za pomocą exec() zwraca tablicę. Sprawdźmy wartość jej pierwszego elementu:

```
> /j.*t/i.exec("Javascript")[0];
"Javascript"
```

Metody obiektu String, których parametram i mogą być wyrażenia regularne

Wcześniej w tym rozdziale opowiadaliśmy o obiekcie String i o wykorzystaniu jego metod indexOf() i lastIndexOf() do przeszukiwania tekstu. Przy ich użyciu można odnajdować w tekście fragmenty przekazane w postaci parametrów. Więcej możliwości daje przeszukiwanie tekstu przy użyciu wyrażeń regularnych. Obiekty String umożliwiają również to.

Obiekty tekstowe posiadają następujące metody przyjmujące jako parametry wyrażenia regularne:

- match() zwraca tablicę dopasowań.
- search() zwraca pozycję pierwszego dopasowania.
- replace() pozwala zamienić dopasowany tekst na inny.
- split() potrafi dzielić tekst na tablicę elementów, także na podstawie wyrażenia regularnego.

search() i match()

Poeksperymentujmy trochę z metodami search() i match(). Zacznijmy od utworzenia nowego obiektu tekstowego.

```
> var s = new String('HelloJavaScriptWorld');
```

match() zwróci tablicę zawierającą tylko pierwsze dopasowanie:

```
> s.match(/a/);
["a"]
```

Jeśli skorzystamy z modyfikatora g, wyszukiwanie będzie miało zasięg globalny i otrzymamy tablicę zawierającą dwa elementy:

```
> s.match(/a/g);
["a", "a"]
```

Pominięcie różnic w wielkości liter:

```
> s.match(/j.*a/i);
["Java"]
```

Metoda `search()` zwraca pozycję dopasowanego łańcucha:

```
> s.search(/j.*a/i);
5
```

replace()

`replace()` umożliwia zamianę dopasowanego tekstu na inny. W poniższym przykładzie przy użyciu tej metody usuwamy z tekstu wszystkie wielkie litery (zamieniając je na pusty łańcuch):

```
> s.replace(/[A-Z]/g, '');
"elloavacriptorld"
```

Jeśli nie zostanie użyty modyfikator `g`, zmieni się tylko pierwszy znaleziony fragment:

```
> s.replace(/[A-Z]/, '');
"elloJavaScriptWorld"
```

Jeśli chcesz wykorzystać odnaleziony tekst jako fragment podstawienia, dostęp do niego uzyskasz za pomocą sekwencji `$&`. Poniższy fragment kodu poprzedza dopasowany tekst znakiem podkreślnika:

```
> s.replace(/[A-Z]/g, "_$&");
"_Hello_Java_Script_World"
```

Jeśli wyrażenie regularne zawiera grupy (oznaczone nawiasami), do poszczególnych grup można się dostać dzięki sekwencjom: `$1` dla pierwszej grupy, `$2` dla drugiej itd.

```
> s.replace(/([A-Z])/g, "_$1");
"_Hello_Java_Script_World"
```

Wyobraź sobie, że na Twojej stronie znajduje się formularz, w który użytkownik powinien wpisać swój adres e-mail, nazwę użytkownika i hasło. Gdy tylko wpisze e-mail, nasz skrypt zasugeruje nazwę użytkownika na podstawie tego adresu:

```
> var email = "stoyan@phpied.com";
> var username = email.replace(/(.*)@.*/, "$1");
> username;
"stoyan"
```

Wywołania zwrotne replace

Podczas zamiany fragmentów tekstu na inne można zamiast konkretnego tekstu podać funkcję zwracającą łańcuch, która w odpowiedni sposób przetworzy odnaleziony tekst.

```
> function replaceCallback(match) {
    return "_" + match.toLowerCase();
  }
> s.replace(/[A-Z]/g, replaceCallback);
"_hello_java_script_world"
```

W rzeczywistości funkcja otrzymuje kilka parametrów (w powyższym przykładzie zignorowaliśmy wszystkie oprócz pierwszego):

- Pierwszy parametr to dopasowywany tekst.
- Drugi to przeszukiwany łańcuch znaków.
- Przedostatni informuje o pozycji dopasowania.
- Pozostałe parametry (jeśli jest ich więcej niż jeden, część z nich pojawi się w tablicy przed informacją o pozycji dopasowania) zawierają fragmenty dopasowane do poszczególnych grup z wzorca.

Przetestujmy to. Utwórzmy zmienną, która będzie przechowywała tablicę argumentów przekazanych podczas wywołania funkcji:

```
> var glob;
```

Następnie zdefiniujmy wyrażenie regularne z trzema grupami, które ma pasować do adresów e-mail postaci `cos@cos.cos`:

```
> var re = /(.*)@(.*)\.(.*)/;
```

Teraz napiszmy funkcję, która przechowa argumenty w zmiennej `glob` i zwróci podstawienie:

```
var callback = function(){
  glob = arguments;
  return arguments[1] + ' małpa ' +
    arguments[2] + ' kropka ' + arguments[3];
}
```

Działanie funkcji jest następujące:

```
> "stoyan@phpied.com".replace(re, callback);
"stoyan małpa phpied kropka com"
```

Oto argumenty, które odebrała funkcja:

```
> glob;
["stoyan@phpied.com", "stoyan", "phpied", "com", 0, "stoyan@phpied.com"]
```

split()

Zapoznaliśmy Cię już z metodą `split()`, która tworzy tablicę na podstawie tekstu wejściowego i łańcucha znaków pełniącego funkcję separatora. Podzielmy łańcuch składający się z wartości oddzielonych przecinkami:

```
> var csv = 'raz, dwa,trzy ,cztery';
> csv.split(',');
["raz", " dwa", "trzy ", "cztery"]
```

Ponieważ w wejściowym tekście spacje nie są stosowane konsekwentnie, w wyniku podziału otrzymaliśmy tablicę, która także zawiera spacje. Można to naprawić, stosując wyrażenie `\s*`, które oznacza „zero lub więcej spacji”:

```
> csv.split(/\s*,\s*/);
["raz", "dwa", "trzy", "cztery"]
```

Przekazanie zwykłego tekstu zamiast wyrażenia regularnego

Warto zapamiętać, że omówione przed chwilą cztery metody (`split()`, `match()`, `search()` i `replace()`) mogą zamiast wyrażeń regularnych pobierać zwykły tekst. W takim wypadku argument zostanie wykorzystany do utworzenia nowego obiektu wyrażenia regularnego, tak jakby został przekazany do konstruktora `new RegExp()`.

Przykład:

```
> "test".replace('e', 'o');
"tost"
```

Powyższe wywołanie jest równoważne:

```
> "test".replace(new RegExp('e'), 'o');
"tost"
```

Jeśli parametr jest łańcuchem znaków, nie można ustawić wartości modyfikatorów, tak jak w przypadku normalnego konstruktora lub literału wyrażenia regularnego. Używanie dla podstawiania tekstu łańcucha znaków zamiast obiektu wyrażenia regularnego jest powszechnym źródłem błędów, ponieważ modyfikator `g` jest domyślnie ustawiony na `false`. Z tego powodu podstawiany jest tylko pierwszy łańcuch znaków, co jest niespójne z większością innych języków i może być mylące. Oto przykład:

```
> "pool".replace('o', '*');
"p*ol"
```

Możemy się domyślić, że chodziło o podstawienie wszystkich wystąpień:

```
> "pool".replace(/o/g, '*');
"p**l"
```

Obiekty Error

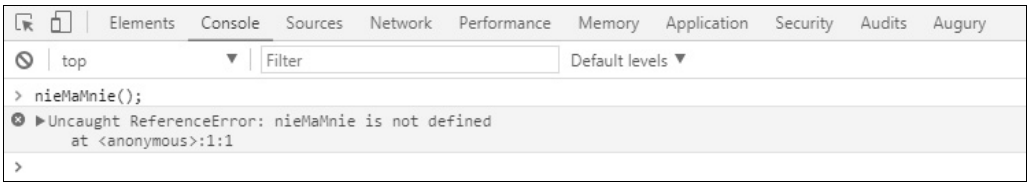
Nie da się całkowicie wyeliminować błędów, dlatego potrzebny jest mechanizm ich wykrywania, dzięki któremu program będzie mógł stwierdzić, że coś poszło nie tak, i w elegancki sposób odzyskać sprawność. Do obsługi błędów w języku JavaScript służą instrukcje `try`, `catch` i `finally`. Kiedy pojawia się błąd, rzucany jest obiekt błędu. Obiekty te tworzy się za pomocą wbudowanych konstruktorów: `EvalError`, `RangeError`, `ReferenceError`, `SyntaxError`, `TypeError` i `URIError`[3]. Wszystkie konstruktory dziedziczą z obiektu `Error`.

[3] Odpowiednio: błąd wykonania, błąd zakresu, błąd referencji, błąd składniowy, błąd typu, błąd adresu URI — *przyp. tłum.*

Spowodujmy błąd i zobaczmy, co się stanie. Nasz przykład będzie próbował wywołać nieistniejącą funkcję. Wpisz w konsoli następujący kod:

```
> nieMaMnie();
```

Wynik będzie mniej więcej taki:



Sposób wyświetlania informacji o błędach jest odmienny w różnych przeglądarkach. W zależności od konfiguracji przeglądarki użytkownik może nawet nie zauważyć, że wystąpił błąd. Nigdy jednak nie będziesz mieć pewności, że wszyscy użytkownicy wyłączyli informowanie o błędach. Uwolnienie ich od konieczności oglądania komunikatów o błędach na Twojej stronie należy tylko do Ciebie. Błąd z naszego przykładu został wyświetlony użytkownikowi, ponieważ kod nie próbował go „przechwycić" i nie był przygotowany na jego obsługę. Na szczęście łapanie błędów jest naprawdę proste. Potrzeba do tego instrukcji `try` (spróbuj), po której nastąpi instrukcja `catch` (przechwyć).

Poniższy kod ukrywa błąd przed użytkownikiem:

```
try {
  nieMaMnie();
} catch (e){
  // nic nie rób
}
```

Mamy tu następujące elementy:

- Instrukcję `try`, po której następuje blok kodu.
- Instrukcję `catch`, po której następuje nazwa zmiennej w nawiasie i kolejny blok kodu.

Istnieje jeszcze nieobowiązkowa instrukcja `finally`. Towarzyszący jej blok kodu jest wykonywany niezależnie od tego, czy wystąpił błąd.

W powyższym przykładzie w żaden sposób nie naprawiamy błędu. Blok następujący po `catch` jest miejscem, w którym możemy wprowadzić konieczne poprawki lub poinformować użytkownika, że zaszły nieoczekiwane okoliczności.

Zmienna `e` w nawiasie po słowie `catch` przechowuje obiekt błędu. Jak wszystkie inne obiekty, zawiera on pewne przydatne właściwości i metody. Niestety różne przeglądarki implementują je na różne sposoby, ale istnieją dwie właściwości, które występują w każdej wersji. Są to `e.name` (nazwa) i `e.message` (komunikat).

Uruchom teraz następujący kod:

```
try {
  nieMaMnie();
} catch (e){
  alert(e.name + ': ' + e.message);
} finally {
  alert('Wreszcie!');
}
```

Pojawi się okienko `alert()` pokazujące nazwę błędu i komunikat, a potem drugie okienko o treści *Wreszcie!*.

W przeglądarkach Firefox i Chrome pierwsze okienko wyświetli tekst *ReferenceError: nieMaMnie is not defined*. W Internet Explorerze będzie to *TypeError: Oczekiwano obiektu*. Na tej podstawie możemy wywnioskować dwa fakty:

- `e.name` przechowuje nazwę konstruktora, który został użyty podczas tworzenia obiektu błędu.
- Skoro w różnych przeglądarkach ten sam błąd w kodzie jest wiązany z różnymi obiektami błędów, nie jest dobrym pomysłem podejmowanie decyzji na temat zachowania kodu na podstawie typu błędu (tzn. wartości `e.name`).

Nowe obiekty błędów można tworzyć samodzielnie za pomocą konstruktora `new Error()` lub dowolnego z dziedziczących z niego konstruktorów. Wystąpienie nowego błędu w kodzie sygnalizujemy za pomocą instrukcji `throw` (rzuć).

Niech nasz kod wywołuje funkcję `mozeIstnieje()`, a następnie wykonuje pewne obliczenia. Chcemy w konsekwentny sposób przechwycić wszystkie błędy, niezależnie od tego, czy zostały one spowodowane tym, że nie istnieje funkcja `mozeIstnieje()`, czy niedozwoloną operacją podczas obliczeń. Oto kod:

```
try {
  var total = mozeIstnieje();
  if (total === 0) {
    throw new Error('Dzielenie przez zero!');
  } else {
    alert(50 / total);
  }
} catch (e){
  alert(e.name + ': ' + e.message);
} finally {
  alert('Wreszcie!');
}
```

Kod zachowa się inaczej w zależności od tego, czy istnieje funkcja `mozeIstnieje()`, i od zwracanych przez nią wartości:

- Jeśli `mozeIstnieje()` nie istnieje, wyświetlony zostanie komunikat *ReferenceError: mozeIstnieje is not defined* (w Firefoksie) lub *TypeError: Oczekiwano obiektu* (w IE).
- Jeśli `mozeIstnieje()` zwraca 0, wystąpi błąd *Dzielenie przez zero!*.
- Jeśli `mozeIstnieje()` zwraca 2, w okienku pojawi się tekst 25.

Niezależnie od istnienia funkcji `mozeIstnieje()` i od zwracanej przez nią wartości, na końcu pojawi się okno dialogowe z komunikatem *Wreszcie!*.

Zamiast rzucania ogólnego błędu `throw new Error('Dzielenie przez zero! ')` możesz zdecydować się na większą drobiazgowość i rzucić na przykład błąd zakresu `throw new RangeError('Dzielenie przez zero! ')`. Możesz także zrezygnować z konstruktora i rzucić zwykły obiekt:

```
throw {
  name: "MójBłąd",
  message: "O rany! Stało się coś strasznego"
}
```

To daje Ci kontrolę nad nazwą błędu w różnych przeglądarkach.

Ćwiczenia

Wykonaj następujące ćwiczenia.

1. Do którego z obiektów (globalnego czy obiektu `o`) odnosi się wartość `this` w poniższym kodzie?

```
function F() {
  function C() {
     return this;
  }
  return C();
}
var o = new F();
```

2. Jaki będzie wynik wykonania poniższego fragmentu kodu?

```
function C(){
  this.a = 1;
  return false;
}
console.log(typeof new C());
```

3. A jaki będzie wynik wykonania tego fragmentu?

```
> c = [1,. 2, [1, 2]];
> c.sort();
> c.join('--');
> console.log(c);
```

4. Wyobraź sobie, że nie istnieje konstruktor `String()`. Utwórz konstruktor `MyString()`, którego działanie będzie tak bliskie działaniu `String()`, jak to tylko możliwe. Nie wolno Ci używać wbudowanych pól i metod obiektu `String` i pamiętaj, że nie istnieje `String()`. Sprawdź działanie kodu przy użyciu następującego testu:

```
> var s = new MyString('witaj');
> s.length;
5
> s[0];
"w"
> s.toString();
"witaj"
> s.valueOf();
"witaj"
> s.charAt(1);
"i"
> s.charAt('2');
"t"
> s.charAt('e');
"w"
> s.concat('świecie!');
"witaj świecie!"
> s.slice(1,3);
"it"
> s.slice(0,-1);
"wita"
> s.split('i');
["w", "aj"]
> s.split('t');
["wi", "aj"]
```

Możesz przejść przez wszystkie znaki łańcucha wejściowego za pomocą pętli `for...in`, traktując go jak tablicę.

5. Dodaj do konstruktora `MyString()` metodę `reverse()` (odwróć).

Wykorzystaj fakt, że tablice posiadają metodę `reverse()`.

6. Wyobraź sobie, że nie istnieje ani `Array()`, ani literałowy sposób tworzenia tablic. Napisz konstruktor `MyArray()`, który zachowuje się w prawie taki sam sposób jak `Array()`. Przeprowadź następujące testy:

```
> var a = new MyArray(1,2,3,"test");
> a.toString();
"1,2,3,test"
> a.length;
4
```

```
> a[a.length - 1];
"test"
> a.push('boo');
5
> a.toString();
"1,2,3,test,boo"
> a.pop();
"boo"
> a.toString();
"1,2,3,test"
> a.join(',')
"1,2,3,test"
> a.join(' nie jest ')
"1 nie jest 2 nie jest 3 nie jest test"
```

Jeśli podoba Ci się to ćwiczenie, nie poprzestawaj na `join()`, ale zaimplementuj również inne metody.

7. Wyobraź sobie, że nie istnieje `Math`. Utwórz obiekt `MyMath`, który posiada następujące metody:
   - `MojMath.rand(min, max, wlacznie)` — losuje liczbę z przedziału od `min` do `max`, włącznie, jeśli `wlacznie` ma wartość `true`.
   - `MojMath.min(tablica)` — zwraca najmniejszy element tablicy.
   - `MojMath.max(tablica)` — zwraca największy element tablicy.

Podsumowanie

W rozdziale 2. przedstawiliśmy pięć prostych typów danych (liczba, łańcuch znaków, boolean, null i undefined). Napisaliśmy także, że wszystko, co nie należy do typu prostego, jest obiektem. Teraz wiesz również, że:

- Obiekty są podobne do tablic, ale sami określamy klucze.
- Obiekty posiadają właściwości.
- Niektóre spośród właściwości są funkcjami (`funkcje to dane, var f = function(){};`). Takie właściwości nazywa się metodami.
- Tablice to obiekty, które posiadają predefiniowane właściwości o nazwach będących liczbami oraz właściwość `length`.
- Obiekty tablicowe posiadają wiele użytecznych metod, takich jak `sort()` czy `slice()`.
- Funkcje także są obiektami posiadającymi właściwości (takie jak `length` i `prototype`) i metody (takie jak `call()` i `apply()`).

Spośród pięciu prostych typów danych wszystkim oprócz undefined (który reprezentuje wartość pustą) i null (który także jest obiektem) odpowiadają konstruktory: `Number()`, `String()` i `Boolean()`.

Przy ich użyciu tworzy się tak zwane obiekty opakowujące, posiadające dodatkowe funkcje ułatwiające pracę z prostymi typami danych.

`Number()`, `String()` i `Boolean()` można wywołać:

- z operatorem `new` — w celu utworzenia nowego obiektu;
- bez `new` — w celu przekształcenia dowolnej wartości na odpowiadającą jej wartość prostą.

Inne omówione w tym rozdziale wbudowane konstruktory to: `Object()`, `Array()`, `Function()`, `Date()`, `RegExp()`i `Error()`. Opisaliśmy także globalny obiekt `Math`, który jednak nie jest konstruktorem.

Wiesz już, że obiekty odgrywają w języku JavaScript podstawową rolę. Prawie wszystko albo jest obiektem, albo może zostać opakowane w obiekt.

Podsumujmy jeszcze sposoby tworzenia obiektów za pomocą notacji literałowej:

Nazwa	Literał	Konstruktor	Przykład
`Object`	`{}`	`new Object()`	`{prop:1}`
`Array`	`[]`	`new Array()`	`[1,2,3,'test']`
wyrażenie regularne	`/wzorzec/modyfikatory`	`new RegExp('wzorzec', 'modyfikatory')`	`/java.*/img`

Skorowidz

G

H

I

J

K

L

R

S

Ś

T

U

W

X

Z

Ż

JavaScript — najlepszy warsztat profesjonalisty!

Dziś JavaScript jest dojrzałym, wszechstronnym i potężnym językiem programowania, który znakomicie się nadaje do programowania zorientowanego obiektowo. Pozwala przy tym na pisanie solidnego i efektywnego kodu. Stał się świetnym narzędziem do tworzenia bardzo złożonych, skalowalnych i łatwych w utrzymaniu aplikacji. Można zaobserwować, że JavaScript powoli nadaje kształt następnej generacji platform internetowych i serwerowych. Najnowsza specyfikacja tego języka, ES6, wprowadza ważne konstrukcje, takie jak obietnice, klasy, funkcje strzałkowe.

Ta książka jest znakomitym podręcznikiem programowania obiektowego w JavaScripcie. Przedstawiono tu solidne podstawy języka oraz programowania obiektowego, co ułatwia zrozumienie zaawansowanych, nowoczesnych funkcjonalności ES6: iteratorów i generatorów. Wyjaśniono koncepcję prototypów i zasady dziedziczenia, a także zasady programowania asynchronicznego. Nie zabrakło kilku niezwykle przydatnych dodatków: listy słów zastrzeżonych, funkcji i obiektów wbudowanych w język, a także wprowadzenia do wyrażeń regularnych. Ponadto każdy rozdział zakończono zestawem przydatnych ćwiczeń do samodzielnego wykonania.

Niektóre zagadnienia omówione w książce:

- Konfiguracja środowiska programistycznego
- Środowisko przeglądarki, model BOM i DOM
- Technika AJAX
- Wzorce kodowania i wzorce projektowe
- Framework Jasmine i projektowanie oparte na testach

Ved Antani od kilkunastu lat używa JavaScriptu, Go i Javy do tworzenia skalowalnych serwerów oraz mobilnych platform. Jest zapalonym czytelnikiem i autorem książek. Studiował informatykę. Mieszka w Bangalore w Indiach. Jest miłośnikiem muzyki klasycznej i uwielbia spędzać czas z synem.
Stoyan Stefanov pracuje w Facebooku, jest autorem książek i często zabiera głos na konferencjach poświęconych WWW. Obecnie mieszka w Los Angeles w Kalifornii. W wolnych chwilach gra na gitarze, lata lub po prostu leniuchuje wraz z rodziną na którejś z plaż Santa Monica.

Helion
księgarnia internetowa
http://helion.pl
zamówienia telefoniczne
0 801 339900
0 601 339900
Informatyka w najlepszym wydaniu

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

sięgnij po WIĘCEJ
KOD KORZYŚCI

ISBN 978-83-283-3782-4
9 788328 337824
cena: 79,00 zł

Packt