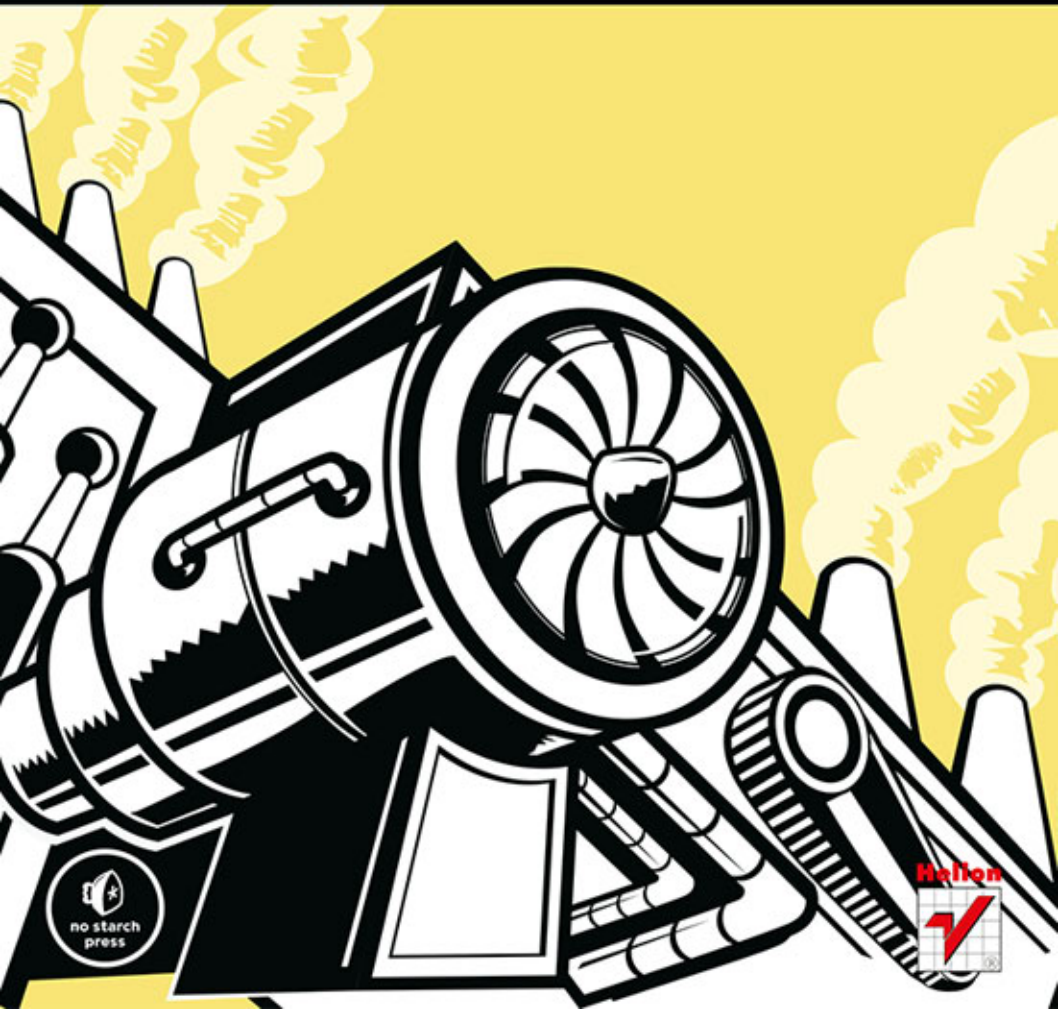
JavaScript

ZASADY PROGRAMOWANIA OBIEKTOWEGO

NICHOLAS C. ZAKAS



Tytuł oryginału: The Principles of Object-Oriented JavaScript

Tłumaczenie: Aleksander Lamża

ISBN: 978-83-246-9592-8

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)

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

Printed in Poland.

Spis treści

1

Typy proste i referencje

Większość programistów uczy się programowania obiektowego na przykładzie języków bazujących na klasach, takich jak Java lub C#. Przy pierwszym zetknięciu z JavaScriptem można się poczuć zdezorientowanym, ponieważ w tym języku pojęcie klas formalnie nie istnieje. Pracy nad kodem nie rozpoczyna się więc od ich zdefiniowania, tylko od razu tworzy się potrzebne struktury danych. Brak klas pociąga za sobą pewne konsekwencje — skoro nie ma klas, nie ma też pakietów. W językach takich jak Java nazwy pakietów i klas definiują zarówno typy używanych w kodzie obiektów, jak i rozmieszczenie plików oraz folderów w projekcie, natomiast w JavaScripcie nic nie jest narzucone, więc o strukturę plików musimy zadbać sami. Niektórzy programiści starają się stosować rozwiązania z innych języków, natomiast inni korzystają z elastyczności JavaScriptu i podchodzą do problemu w całkiem nowy sposób. Co prawda dla stawiających pierwsze kroki programistów ta swoboda może stać się przyczyną nieporozumień

i problemów, ale kiedy się przywyknie do specyfiki JavaScriptu, okaże się, że to niezwykle efektywny i wygodny język programowania.

Przejście z klasycznych języków obiektowych na JavaScript może ułatwiać to, że on również bazuje na obiektach — przeważnie zapisuje się w nich dane i umieszcza funkcje. Tak naprawdę w JavaScripcie nawet funkcje są obiektami, co czyni je najistotniejszymi elementami języka.

Zrozumienie obiektowości w JavaScripcie i nauczenie się korzystania z obiektów są kluczowe dla zrozumienia JavaScriptu jako języka. Obiekty można tworzyć w dowolnym momencie, niejako z marszu. Dotyczy to również właściwości obiektów — można je dodawać i usuwać, kiedy tylko okaże się to konieczne. Poza tym elastyczność obiektów w JavaScripcie pozwala na stosowanie ciekawych i przydatnych wzorców, których nie da się zrealizować w wielu innych językach.

Ten rozdział jest poświęcony dwóm podstawowym typom danych w JavaScripcie: typom prostym oraz referencjom. Mimo że dostęp do zmiennych obu typów jest realizowany przez obiekty, zachowują się one inaczej, więc konieczne jest zrozumienie dzielących je różnic.

Czym są typy?

Mimo że w JavaScripcie nie stosuje się pojęcia klas, funkcjonują dwa rodzaje **typów** danych: typy proste i referencje. **Typy proste** (ang. *primitives*) służą do zapisywania danych w prostej postaci, natomiast w przypadku **typów referencyjnych** (ang. *references*) dane są zapisywane w obiektach, do których (a właściwie ich lokalizacji w pamięci) prowadzi referencja.

Co ciekawe, JavaScript pozwala traktować typy proste jak referencje, dzięki czemu z punktu widzenia programisty język jest bardziej spójny.

W wielu innych językach istnieje ścisłe rozróżnienie między tymi dwoma typami danych. Zmienna typu prostego jest przechowywana na stosie, a referencja na stercie. W JavaScripcie jest to rozwiązane zupełnie inaczej — zmienna jest zapisywana w **obiekcie zmiennych**. Proste wartości są przechowywane bezpośrednio w tym obiekcie, a w przypadku referencji w obiekcie zmiennej jest zapisany wskaźnik do lokalizacji obiektu w pamięci. Z dalszej lektury tego rozdziału dowiesz się, że mimo że pozornie typy te są do siebie podobne, w wielu sytuacjach zachowują się zupełnie inaczej.

Typy proste

Typy proste reprezentują dane zapisane bezpośrednio w takiej formie, w jakiej występują, jak choćby wartości `true` czy `25`. W JavaScripcie istnieje pięć typów prostych:

`boolean`	wartość logiczna `true` albo `false`
`number`	liczbowa wartość całkowita lub zmiennoprzecinkowa
`string`	znak lub łańcuch znaków ujęty w cudzysłowy lub apostrofy (w JavaScripcie nie ma odrębnego typu dla pojedynczego znaku)
`null`	typ prosty posiadający tylko jedną wartość: `null`
`undefined`	typ prosty posiadający tylko jedną wartość: `undefined` (jest to wartość przypisywana do zmiennej, która nie została jeszcze zainicjalizowana)

Pierwsze trzy typy (`boolean`, `number` i `string`) zachowują się podobnie, natomiast dwa ostatnie (`null` i `undefined`) są traktowane inaczej, co zostanie szczegółowo opisane w dalszej części rozdziału. Wartości wszystkich typów prostych mają reprezentację w postaci literałów. **Literały** to te wartości, które nie znajdują się w zmiennych, czyli na przykład zapisane w kodzie imię lub cena. Poniżej kilka przykładów użycia literałów omawianych typów prostych:

```
// string
var name = "Nicholas";
var selection = "a";
// number
var count = 25;
var cost = 1.51;
// boolean
var found = true;
// null
var object = null;
// undefined
var flag = undefined;
var ref; // wartość undefined zostanie przypisana automatycznie
```

W JavaScripcie, podobnie jak w wielu innych językach, zmienna typu prostego przechowuje wartość (a nie wskaźnik do obiektu). Przypisanie

wartości do takiej zmiennej polega więc na utworzeniu kopii wartości i zapisaniu jej w zmiennej. Stąd wniosek, że jeśli do zmiennej przypisze się inną zmienną, każda z nich będzie przechowywała własną kopię tej samej wartości. Ilustruje to poniższy przykład:

```
var color1 = "czerwony";
var color2 = color1;
```

Najpierw zmiennej `color1` jest przypisywana wartość `"czerwony"`. Następnie zmiennej `color2` jest przypisywana wartość zmiennej `color1`, czyli `"czerwony"`. Mimo że obie zmienne mają tę samą wartość, nie są w żaden sposób ze sobą powiązane, więc jeśli zmieni się wartość pierwszej zmiennej, nie wpłynie to na drugą (i odwrotnie). Wynika to z tego, że wartości tych zmiennych są przechowywane w innych miejscach, co ilustruje rysunek 1.1.

Rysunek 1.1. Obiekt zmiennej

Podsumowując, zmienne typów prostych zajmują odrębne miejsca w pamięci, więc zmiany wartości jednej zmiennej nie wpływają na inne zmienne. Oto jeszcze jeden przykład:

```
var color1 = "czerwony";
var color2 = color1;

console.log(color1); // "czerwony"
console.log(color2); // "czerwony"

color1 = "niebieski";

console.log(color1); // "niebieski"
console.log(color2); // "czerwony"
```

W powyższym kodzie zmieniamy wartość zmiennej `color1` na `"niebieski"`, co nie wpływa na oryginalną wartość (`"czerwony"`) zmiennej `color2`.

Identyfikowanie typów prostych

Najlepszym sposobem określenia typu jest użycie operatora `typeof`. Działa on ze wszystkimi zmiennymi i zwraca tekstową reprezentację typu danych. W przypadku łańcuchów tekstowych, liczb, wartości logicznych i typu `undefined` operator działa w przewidywalny sposób, co widać w poniższym przykładzie:

```
console.log(typeof "Nicholas");     // "string"
console.log(typeof 10);             // "number"
console.log(typeof 5.1);            // "number"
console.log(typeof true);           // "boolean"
console.log(typeof undefined);      // "undefined"
```

Jak można się spodziewać, operator `typeof` zwraca `"string"`, jeśli wartość jest łańcuchem tekstowym, `"number"` w przypadku liczb (bez względu na to, czy to liczba całkowita, czy zmiennoprzecinkowa), `"boolean"` dla wartości logicznych oraz `"undefined"` — w sytuacji gdy wartość nie jest zdefiniowana.

Sprawy się komplikują w przypadku typu `null`.

Niejednego programistę zdziwi wynik wykonania poniższego kodu:

```
console.log(typeof null);           // "object"
```

Operator `typeof` dla wartości `null` zwraca `"object"`. Dlaczego obiekt, skoro to typ `null`? Sprawa jest dyskusyjna (TC39, czyli komitet odpowiedzialny za zaprojektowanie i utrzymywanie JavaScriptu, przyznał, że to błąd), ale jeśli przyjmiemy, że `null` jest pustym wskaźnikiem do obiektu, rezultat `"object"` można uznać za uzasadniony.

Najlepszym sposobem na sprawdzenie, czy wartość to `null`, jest bezpośrednie porównanie:

```
console.log(value === null);        // true albo false
```

PORÓWNYWANIE BEZ ZMIANY TYPU

Zwróć uwagę, że w powyższym przykładzie został zastosowany operator potrójnego znaku równości (`===`), a nie standardowy (`==`). Wynika to z tego, że potrójny operator nie wymusza zmiany typu porównywanych zmiennych. Poniższy przykład pomoże zrozumieć, dlaczego to takie ważne:

```
console.log("5" == 5);              // true
console.log("5" === 5);             // false
console.log(undefined == null);     // true
console.log(undefined === null);    // false
```

W przypadku zastosowania podwójnego znaku równości łańcuch `"5"` i liczba `5` są równe, ponieważ przed wykonaniem porównania łańcuch zostaje przekonwertowany na liczbę. Operator potrójnego znaku równości nie uzna tych dwóch wartości za równe, ponieważ są innego typu. Podobnie sprawa wygląda w przypadku porównywania `undefined` i `null`. Wniosek stąd prosty — jeśli chcemy sprawdzić, czy jakaś wartość jest `null`, musimy zastosować potrójny znak równości, tak by był brany pod uwagę również typ.

Metody typów prostych

Mimo że łańcuchy, liczby i wartości logiczne są typami prostymi, mają metody (wyjątkiem są jednak typy `null` i `undefined`). Pod względem liczby przydatnych metod wyróżnia się typ `string`. Kilka z nich zostało przedstawionych na poniższym listingu:

```
var name = "Nicholas";
var lowercaseName = name.toLowerCase();  // zamienia na małe znaki
var firstLetter = name.charAt(0);        // zwraca pierwszy znak
var middleOfName = name.substring(2, 5); // zwraca sekwencję znaków 2 - 4

var count = 10;
var fixedCount = count.toFixed(2);       // konwertuje na łańcuch "10.00"
var hexCount = count.toString(16);       // konwertuje na łańcuch "a"

var flag = true;
var stringFlag = flag.toString();        // konwertuje na łańcuch "true"
```

UWAGA *Na wartościach typów prostych — mimo że nie są obiektami — można wywoływać metody. Odpowiadają za to mechanizmy JavaScriptu, dzięki którym wartości te wydają się obiektami, a to podnosi spójność języka.*

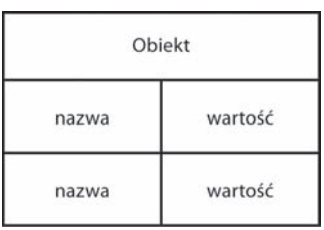
Typy referencyjne

Typy referencyjne reprezentują w JavaScripcie obiekty. Referencje to **instancje** typów referencyjnych, więc można je uznać za synonim *obiektów* (w dalszej części rozdziału terminy te będą stosowane zamiennie). Obiekt jest nieuporządkowaną listą **właściwości** złożonych z nazwy (będącej zawsze łańcuchami) i wartości. W przypadku gdy wartością właściwości jest funkcja, nazywa się ją **metodą**. Funkcje są tak naprawdę referencjami, więc istnieje drobna różnica między właściwością przechowującą — powiedzmy — tablicę a taką, która zawiera funkcję (oczywiście poza tym, że funkcję można wywołać, a tablicę nie).

Zanim zacznie się korzystać z obiektu, trzeba go utworzyć.

Tworzenie obiektów

Obiekty w JavaScripcie dobrze jest sobie wyobrazić jako tablice asocjacyjne (patrz rysunek 1.2).



Rysunek 1.2. Struktura obiektu

Istnieje kilka sposobów tworzenia obiektów (*instancji*). Pierwszym z nich jest użycie operatora `new` w połączeniu z **konstruktorem** obiektu. Konstruktor to funkcja, która jest używana przez operator `new` (może to być dowolna funkcja). W JavaScripcie przyjęto konwencję rozpoczynania nazw konstruktorów wielką literą, by odróżnić je od zwykłych funkcji. Poniższy kod tworzy instancję ogólnego obiektu i zapisuje referencję do niego w zmiennej `object`:

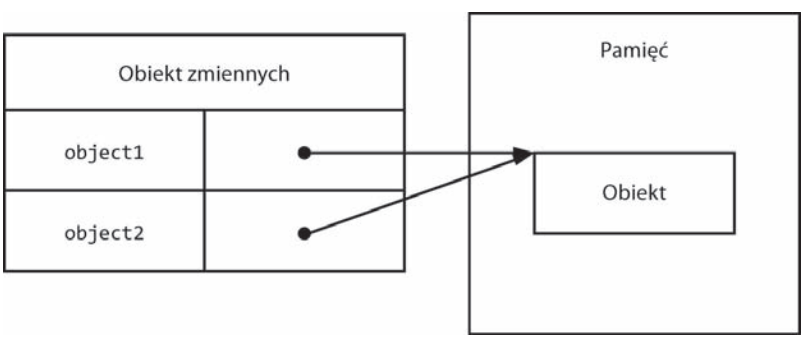
```
var object = new Object();
```

W przypadku typów referencyjnych obiekty nie są zapisywane bezpośrednio w zmiennej, do której są przypisywane, więc zmienna `object` z powyższego przykładu nie zawiera utworzonego obiektu. Przechowywany jest za to wskaźnik (czyli referencja) do lokalizacji w pamięci, gdzie znajduje się obiekt. Jest to podstawowa różnica między obiektami i wartościami typów prostych, które są przechowywane bezpośrednio w zmiennej.

Po przypisaniu obiektu do zmiennej zostaje w niej tak naprawdę zapisany wskaźnik. Gdybyśmy tę zmienną przypisali innej zmiennej, obie przechowywałyby kopię wskaźnika, który prowadzi do tego samego obiektu:

```
var object1 = new Object();
var object2 = object1;
```

W tym kodzie najpierw jest tworzony obiekt (za pomocą operatora `new`), a referencja do niego jest zapisywana w zmiennej `object1`. Następnie zmienna ta jest przypisywana zmiennej `object2`. W tej chwili istnieje nadal tylko jedna instancja (utworzona w pierwszym wierszu kodu), ale referencja do niej jest przechowywana w dwóch zmiennych, co widać na rysunku 1.3.



Rysunek 1.3. Dwie zmienne wskazujące ten sam obiekt

Dereferencja obiektów

W języku JavaScript jest stosowany mechanizm odśmiecania pamięci (ang. *garbage collection*), więc podczas stosowania referencji nie ma potrzeby zbytniego przejmowania się zwalnianiem alokowanej pamięci.

Dobrym zwyczajem jest jednak **dereferencja** obiektów, które nie będą już używane, tak aby odśmiecacz mógł zwolnić zajmowaną przez nie pamięć. Aby to zrobić, wystarczy zmiennej referencyjnej przypisać `null`:

```
var object1 = new Object();

// jakieś operacje na obiekcie

object1 = null; // dereferencja
```

Obiekt `object1` jest tworzony, następnie używany, a na końcu zmiennej jest przypisywany `null`. Jeśli nie istnieją żadne inne referencje do tego obiektu, odśmiecacz może zwolnić zajmowaną przez niego pamięć. Świadome wykorzystanie tego mechanizmu jest szczególnie istotne w dużych aplikacjach, w których jest tworzonych bardzo dużo obiektów.

Dodawanie i usuwanie właściwości

Bardzo ciekawą cechą obiektów w JavaScripcie jest możliwość dodawania i usuwania właściwości w dowolnym momencie. Oto przykład:

```
var object1 = new Object();
var object2 = object1;

object1.myCustomProperty = "Super!";
console.log(object2.myCustomProperty); // "Super!"
```

Do obiektu `object1` zostaje dodana właściwość `myCustomProperty` o wartości `"Super!"`. Do właściwości tej można się odwołać również przez obiekt `object2`, ponieważ obie zmienne (`object1` i `object2`) wskazują tę samą instancję.

UWAGA *Ten przykład ilustruje jedną z ciekawszych cech języka JavaScript — możliwość modyfikowania obiektów w dowolnym momencie, nawet jeśli wcześniej nie zostały w ogóle zdefiniowane. W sytuacjach gdy taka możliwość jest niepożądana, można temu zapobiegać na kilka sposobów (poznasz je w dalszej części książki).*

Poza podstawowym, ogólnym typem referencyjnym JavaScript oferuje kilka innych, bardzo przydatnych, wbudowanych typów.

Tworzenie instancji wbudowanych typów

Wcześniej omówiliśmy sposób tworzenia ogólnych obiektów za pomocą operatora `new` i konstruktora: `new Object()`. Typ `Object` jest tylko jednym z kilku przydatnych wbudowanych typów referencyjnych dostępnych w języku JavaScript. Pozostałe są bardziej wyspecjalizowane.

Poniżej znajduje się krótki opis wbudowanych typów:

`Array`	tablica indeksowana numerycznie
`Date`	data i czas
`Error`	błąd w czasie wykonania (istnieją również bardziej konkretne podtypy błędów)
`Function`	funkcja
`Object`	ogólny obiekt
`RegExp`	wyrażenie regularne

Instancje wbudowanych typów można tworzyć za pomocą operatora `new`, co widać w poniższym przykładzie:

```
var items = new Array();
var now = new Date();
var error = new Error("Stało się coś złego.");
var func = new Function("console.log('Cześć');");
var object = new Object();
var re = new RegExp("\\d+");
```

Literały

Niektóre wbudowane typy są dostępne w formie literałów. **Literał** to składnia umożliwiająca zdefiniowanie typu referencyjnego bez potrzeby jawnego tworzenia obiektu za pomocą operatora `new` i konstruktora. We wcześniejszych przykładach z tego rozdziału były stosowane literały wartości prostych, czyli literały łańcuchów, liczb, wartości logicznych, a także literały `null` i `undefined`.

Literały obiektów i tablic

Aby utworzyć obiekt za pomocą **literału obiektu**, wystarczy zdefiniować właściwości wewnątrz nawiasów klamrowych. Każda właściwość jest zbudowana z nazwy (łańcucha), dwukropka i wartości. Poszczególne właściwości rozdziela się przecinkami:

```
var book = {
   name: "JavaScript. Programowanie obiektowe",
   year: 2014
};
```

Nazwa właściwości może być również literałem łańcucha, co bywa przydatne w przypadku nazw zawierających spacje lub inne znaki specjalne:

```
var book = {
   "name": "JavaScript. Programowanie obiektowe",
   "year": 2014
};
```

Pomijając różnicę w składni, ten przykład jest równoważny poprzedniemu. Ten sam efekt można uzyskać w jeszcze inny sposób:

```
var book = new Object();
book.name = "JavaScript. Programowanie obiektowe";
book.year = 2014;
```

Wynik działania wszystkich trzech przedstawionych fragmentów kodu jest taki sam — powstaje obiekt z dwiema właściwościami. Można więc korzystać z dowolnego zapisu, ponieważ funkcjonalnie są one dokładnie takie same.

UWAGA *Użycie literału obiektu nie wiąże się tak naprawdę z wywołaniem konstruktora* `new Object()`*. Silnik JavaScriptu wykonuje jednak te same operacje co w przypadku tego wywołania. Odnosi się to do wszystkich literałów typów referencyjnych.*

W podobny sposób stosuje się **literał tablicy**: dowolną liczbę wartości rozdzielonych przecinkami umieszcza się w nawiasach kwadratowych:

```
var colors = [ "czerwony", "niebieski", "zielony" ];
console.log(colors[0]);          // "czerwony"
```

Taki zapis daje ten sam efekt co poniższy kod:

```
var colors = new Array("czerwony", "niebieski", "zielony");
console.log(colors[0]);          // "czerwony"
```

Literały funkcji

W większości przypadków funkcje definiuje się za pomocą literałów. Tak naprawdę konstruktor `Function` jest stosowany w specyficznych sytuacjach — gdy ciało funkcji jest zapisane jako łańcuch. Taki kod jest trudny w utrzymaniu, analizowaniu i debugowaniu, więc raczej nie spotkasz się z nim w praktyce.

Tworzenie funkcji za pomocą literału jest o wiele prostsze i niesie za sobą mniejsze ryzyko popełnienia błędu. Oto przykład:

```
function reflect(value) {
   return value;
}

// jest równoważne temu:

var reflect = new Function("value", "return value;");
```

W powyższym kodzie jest definiowana funkcja `reflect()`, która zwraca przekazaną do niej wartość. Nawet w przypadku tak prostej funkcji można zauważyć, że postać literału jest o wiele prostsza w czytaniu i zrozumieniu niż wersja z konstruktorem. Trzeba też pamiętać, że nie ma dobrego sposobu na debugowanie funkcji utworzonych za pomocą konstruktora, ponieważ takie funkcje nie są rozpoznawane przez debugery JavaScriptu. W aplikacji są więc czarnymi skrzynkami — wiemy tylko, co do nich jest przekazywane i co zwracają.

Literały wyrażeń regularnych

JavaScript oferuje również **literały wyrażeń regularnych**, dzięki którym można definiować wyrażenia regularne bez konieczności użycia konstruktora `RegExp`. Są one zapisywane podobnie jak w języku Perl: wzorzec

jest zapisywany między prawymi ukośnikami, a ewentualne opcje umieszcza się za ukośnikiem zamykającym:

```
var numbers = /\d+/g;

// jest równoważne temu:

var numbers = new RegExp("\\d+", "g");
```

Z literałów wyrażeń regularnych korzysta się trochę prościej niż z konstruktora, ponieważ nie trzeba stosować znaków ucieczki (tzw. *escaping*). Kiedy używa się konstruktora, wyrażenie regularne przekazuje się jako łańcuch, więc wszystkie znaki lewych ukośników trzeba poprzedzić dodatkowym lewym ukośnikiem (właśnie dlatego w powyższym przykładzie w wersji z literałem jest `\d`, a w wersji z konstruktorem — `\\d`). Z tego względu lepiej stosować literały, z wyjątkiem sytuacji gdy wyrażenie regularne jest konstruowane dynamicznie.

Podsumowując, poza przypadkiem konstruktora `Function` tak naprawdę nie ma większego znaczenia sposób tworzenia instancji wbudowanych typów. Niektórzy programiści preferują literały, a inni konstruktory. Stosuj więc takie rozwiązanie, które w danej sytuacji wydaje się wygodniejsze i lepsze.

Dostęp do właściwości

Jak już zostało wspomniane, właściwości to pary nazwa-wartość zapisane w obiekcie. W JavaScripcie, podobnie jak w wielu innych językach, w celu uzyskania dostępu do właściwości najczęściej stosuje się notację z kropką. Można jednak zastosować zapis z nawiasami klamrowymi, między którymi umieszcza się nazwę właściwości.

Jako przykład posłuży prosty kod, w którym został zastosowany zapis z kropką:

```
var array = [];
array.push(12345);
```

Można to zapisać inaczej, umieszczając nazwę metody (jako łańcuch) w nawiasach kwadratowych:

```
var array = [];
array["push"](12345);
```

Taki zapis przydaje się w sytuacjach, gdy musimy dynamicznie decydować, do której właściwości chcemy uzyskać dostęp. Zamiast korzystać z literału łańcucha, nazwę pożądanej właściwości można zapisać w zmiennej i przekazać ją w nawiasach klamrowych:

```
var array = [];
var method = "push";
array[method](12345);
```

Zmienna `method` ma wartość `"push"`, więc na tablicy (zmiennej `array`) jest wywoływana metoda `push()`. Ta przydatna możliwość zostanie wykorzystana jeszcze kilkukrotnie w dalszej części książki. Jeśli pominiemy składnię czy wydajność, zauważymy, że jedyna różnica między zapisem z kropką i nawiasami polega na tym, że drugi sposób daje możliwość zastosowania w nazwach właściwości znaków specjalnych, co czasem może się przydać. Programiści preferują zapis z kropką, ponieważ taki kod łatwiej się czyta i analizuje.

Identyfikowanie typów referencyjnych

Najprostszym do zidentyfikowania typem referencyjnym jest funkcja, ponieważ operator `typeof` zwraca łańcuch `"function"`:

```
function reflect(value) {
   return value;
}

console.log(typeof reflect);          // "function"
```

Pozostałe typy referencyjne trudniej określić, ponieważ operator `typeof` zwraca dla nich wszystkich łańcuch `"object"`. Może się to okazać problemem, jeśli korzysta się z wielu różnych typów. W takich sytuacjach z pomocą przychodzi inny operator języka JavaScript — `instanceof`.

Operator ten ma dwa parametry: obiekt i konstruktor. Jeśli obiekt jest instancją typu określonego przez konstruktor, operator zwraca `true`, a w przeciwnych przypadku — `false`. Oto przykład:

```
var items = [];
var object = {};

function reflect(value) {
   return value;
}

console.log(items instanceof Array);       // true
console.log(object instanceof Object);     // true
console.log(reflect instanceof Function);  // true
```

W powyższym kodzie za pomocą operatora `instanceof` jest testowanych kilka wartości. Wszystkie zostały prawidłowo zidentyfikowane jako instancje typów określonych przez podane konstruktory (mimo że obiekty były tworzone za pomocą literałów).

Operator `instanceof` poprawnie identyfikuje również typy dziedziczone po innych typach. Ze względu na to, że wszystkie typy wbudowane dziedziczą po `Object`, każdy obiekt będzie rozpoznany jako instancja typu `Object`.

Dobrze to ilustruje poniższy przykład, w którym pod tym kątem są sprawdzane wcześniej utworzone obiekty:

```
var items = [];
var object = {};

function reflect(value) {
   return value;
}

console.log(items instanceof Array);          // true
console.log(items instanceof Object);         // true
console.log(object instanceof Object);        // true
console.log(object instanceof Array);         // false
console.log(reflect instanceof Function);     // true
console.log(reflect instanceof Object);       // true
```

Wszystkie obiekty typów wbudowanych zostały rozpoznane jako instancje typu `Object`, ponieważ po nim dziedziczą.

Identyfikowanie tablic

Mimo że operator `instanceof` prawidłowo rozpoznaje tablice, jest jeden wyjątek, gdy sobie nie radzi. W aplikacjach webowych może dojść do sytuacji, gdy wartości są wymieniane między ramkami w obrębie tej samej strony internetowej. Problem polega na tym, że każda ramka ma swój własny globalny kontekst, a więc własne wersje `Object`, `Array` i pozostałych typów wbudowanych. Jeśli przekaże się tablicę z jednej ramki do drugiej, operator `instanceof` nie zadziała prawidłowo, ponieważ ta tablica jest instancją typu `Array` z innej ramki.

Problem ten rozwiązuje wprowadzona w ECMAScript 5 metoda `Array.isArray()`, która jednoznacznie określa, czy przekazana jej wartość jest tablicą, czy nie. Metoda ta zwraca `true`, jeśli przekaże się jej natywną tablicę, niezależnie od kontekstu, z którego pochodzi. Jest to najlepszy sposób na identyfikowanie tablic:

```
var items = [];
console.log(Array.isArray(items));      // true
```

Metodę `Array.isArray()` można stosować w środowiskach zgodnych z ECMAScript 5, a więc w zdecydowanej większości przeglądarek (Internet Explorer obsługuje ją od wersji 9.) i w środowisku Node.js.

Typy opakowujące

W języku JavaScript spore zamieszanie może wprowadzać koncepcja **typów opakowujących typy proste** (ang. *primitive wrapper types*). Istnieją trzy takie typy: `String`, `Number` i `Boolean`. Są to typy referencyjne, dzięki którym wartościami typów prostych można się posługiwać jak obiektami, co okazuje się bardzo przydatne. Wystarczy sobie wyobrazić bałagan w kodzie spowodowany koniecznością zastosowania innej składni lub przejścia na styl programowania proceduralnego z powodu, na przykład, potrzeby wyznaczenia podłańcucha z jakiejś zmiennej tekstowej.

Typy opakowujące są automatycznie tworzone, w chwili gdy odczytywana jest wartość typu prostego `string`, `number` lub `boolean`. Przeanalizujmy prosty przykład. W pierwszym wierszu poniższego fragmentu kodu zmiennej `name` jest przypisywany łańcuch typu prostego (`string`). W drugim wierszu traktujemy tę zmienną jak obiekt — wywołujemy na niej metodę `charAt(0)` z zastosowaniem notacji z kropką.

```
var name = "Nicholas";
var firstChar = name.charAt(0);
console.log(firstChar);              // "N"
```

Aby zobrazować, co się dzieje w tym kodzie poza naszą wiedzą, trzeba go uzupełnić o kilka dodatkowych operacji:

```
// silnik JavaScriptu wykonuje takie operacje
var name = "Nicholas";
var temp = new String(name);
var firstChar = temp.charAt(0);
temp = null;
console.log(firstChar);              // "N"
```

Ponieważ potraktowaliśmy zmienną typu prostego jak obiekt, silnik JavaScriptu utworzył instancję typu `String`, tak by dało się wywołać metodę `charAt(0)`. Obiekt typu `String` istnieje tylko na potrzeby tej jednej operacji (proces ten określa się terminem **autoboxing**). Łatwo to sprawdzić — wystarczy dodać do zmiennej `name` jakąś właściwość (traktując tę zmienną jak zwykły obiekt):

```
var name = "Nicholas";
name.last = "Zakas";

console.log(name.last);              // undefined
```

W tym kodzie do łańcucha `name` staramy się dodać właściwość `last`. Kod wydaje się prawidłowy, ale wynik jego działania już nie. Co się dzieje? Gdy pracujemy z obiektami, możemy dodawać do nich nowe właściwości, kiedy tylko chcemy, a co ważne, pozostają one w obiekcie aż do chwili ich usunięcia. W przypadku typów opakowujących jest inaczej, ponieważ automatycznie tworzone obiekty są praktycznie od razu niszczone.

Powyższy fragment kodu zostanie uzupełniony przez silnik JavaScriptu w następujący sposób:

```
// silnik JavaScriptu wykonuje takie operacje
var name = "Nicholas";
var temp = new String(name);
temp.last = "Zakas";
temp = null;                    // tymczasowy obiekt zostaje zniszczony

var temp = new String(name);
console.log(temp.last);         // undefined
temp = null;
```

Nowa właściwość nie jest przypisywana do łańcucha, ale do tymczasowego obiektu, który jest od razu niszczony. Kiedy w dalszej części kodu chcemy odczytać tę właściwość, znów jest tworzony tymczasowy obiekt, który oczywiście jej nie posiada.

Jeżeli na zmiennej typu prostego zastosujemy operator `instanceof`, uzyskamy wynik `false`, mimo że — jak widzieliśmy — referencje są tworzone automatycznie:

```
var name = "Nicholas";
var count = 10;
var found = false;

console.log(name instanceof String);        // false
console.log(count instanceof Number);       // false
console.log(found instanceof Boolean);      // false
```

Dzieje się tak dlatego, że tymczasowe obiekty są tworzone tylko w sytuacji, gdy odczytywana jest wartość zmiennej. Operator `instanceof` niczego nie odczytuje, więc nie jest tworzony tymczasowy obiekt, w związku z czym zmienna nie jest instancją typu opakowującego. Obiekt opakowujący można utworzyć jawnie, ale ma to pewne efekty uboczne:

```
var name = new String("Nicholas");
var count = new Number(10);
var found = new Boolean(false);
```

```
console.log(typeof name);          // "object"
console.log(typeof count);         // "object"
console.log(typeof found);         // "object"
```

Jak widać, tworzymy w ten sposób ogólne obiekty, więc operator `typeof` nie może zidentyfikować faktycznego typu przechowywanych w nich danych.

Trzeba też wiedzieć, że w wielu sytuacjach instancje typów `String`, `Number` i `Boolean` zachowują się inaczej niż ich proste odpowiedniki. W poniższym fragmencie kodu korzystamy z obiektu `Boolean` o wartości `false`. Po uruchomieniu kodu w konsoli zostanie jednak wyświetlony komunikat `"Znaleziony"`, ponieważ w instrukcji warunkowej obiekt jest zawsze traktowany jak wartość `true`. Nie jest więc istotne, że ten obiekt opakowuje wartość `false` — jest obiektem, więc w tej sytuacji jest traktowany jako `true`.

```
var found = new Boolean(false);

if (found) {
   console.log("Znaleziony");      // to zostanie wykonane
}
```

Ręczne tworzenie instancji typów opakowujących może wprowadzać zamieszanie, więc lepiej tego unikać, chyba że ma się wyraźny powód. Korzystanie z obiektów opakowujących zamiast prostych wartości prowadzi najczęściej do trudnych do wykrycia błędów.

Podsumowanie

Mimo że w języku JavaScript nie ma klas, są typy. Wszystkie zmienne i fragmenty danych są powiązane z określonymi wartościami typu prostego lub referencjami. Pięć typów prostych (łańcuchy, liczby, wartości logiczne, `null` i `undefined`) reprezentuje wartości zapisywane bezpośrednio w obiekcie zmiennych dla danego kontekstu. Do identyfikowania tych typów można użyć operatora `typeof`. Wyjątkiem jest `null`, który można wykryć tylko przez bezpośrednie porównanie z wartością `null`.

Typy referencyjne w JavaScripcie najbliżej odpowiadają klasom znanym z innych języków. Obiekty są instancjami klas referencyjnych. Nowe obiekty można tworzyć za pomocą operatora `new` lub literału referencji.

Dostęp do właściwości i metod jest realizowany przede wszystkim za pomocą notacji z kropką, ale można też korzystać z notacji z nawiasami kwadratowymi. Funkcje są w JavaScripcie obiektami i można je identyfikować za pomocą operatora `typeof`. Aby sprawdzić, czy obiekt jest instancją określonego typu referencyjnego, należy użyć operatora `instanceof`.

Dzięki trzem typom opakowującym — `String`, `Number` i `Boolean` — zmienne typów prostych można traktować jak referencje. Silnik JavaScriptu automatycznie tworzy obiekty opakowujące, więc z poziomu kodu prostymi zmiennymi można się posługiwać jak obiektami, ale trzeba pamiętać, że obiekty te są tymczasowe, więc są niszczone zaraz po ich użyciu. Mimo że instancje typów opakowujących można tworzyć samodzielnie, nie jest to zalecane ze względu na wprowadzanie zamieszania i większe prawdopodobieństwo popełnienia pomyłki.

Skorowidz

G

H

I

J

K

L

M

N

O

P

R

S

T

W

Z

POZNAJ OBIEKTOWY CHARAKTER JĘZYKA JAVASCRIPT!

Programiści pracujący na co dzień z użyciem takich języków jak Java, C# czy C++ z pewnym pobłażaniem patrzą na JavaScript. Traktują go jako język nie do końca obiektowy, w którym można napisać program działający bez tworzenia klas i obiektów. Są w błędzie! JavaScript to język o ogromnych możliwościach, pozwalający na obiektowe tworzenie programów.
Nie wierzysz? Sięgnij po tę książkę i przekonaj się na własnej skórze!

Znajdziesz tu szczegółowe omówienie obiektowych elementów języka JavaScript. Poznasz podstawowe różnice pomiędzy typami prostymi i referencyjnymi oraz dowiesz się, jak sobie z nimi radzić w trakcie pracy z tym językiem. W kolejnych rozdziałach zaznajomisz się ze specyfiką funkcji w JavaScripcie oraz nauczysz się rozpoznawać charakterystyczne elementy obiektów. Ponadto Twoją uwagę powinien zwrócić rozdział poświęcony konstruktorom, prototypom oraz technikom dziedziczenia. Ta książka jest obowiązkową lekturą dla wszystkich programistów tworzących programy w języku JavaScript.

Dzięki tej książce:

- poznasz typy proste i referencyjne,
- nauczysz się korzystać z funkcji,
- zastosujesz obiekty w codziennej pracy,
- zaznajomisz się z konstruktorami i prototypami,
- poznasz wzorce tworzenia obiektów.

sięgnij po WIĘCEJ

KOD KORZYŚCI

helion.pl księgarnia internetowa

Nr katalogowy: 24849

Księgarnia internetowa
http://helion.pl

Zamówienia telefoniczne:
0 801 339900
0 601 339900

Helion

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

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

Informatyka w najlepszym wydaniu

ISBN 978-83-246-9592-8

9 788324 695928

Cena: 29,90 zł