O'REILLY®

Helion

KYLE SIMPSON

ECMAScript 6 *i* DALEJ

TAJNIKI JĘZYKA

JavaScript

JS

Tytuł oryginału: You Don't Know JS: ES6 & Beyond

Tłumaczenie: Piotr Rajca

ISBN: 978-83-283-2311-7

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/tajnjs
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

ROZDZIAŁ 5.

Kolekcje

Kolekcje oraz dostęp do danych o określonej strukturze to kluczowe elementy niemal wszystkich programów pisanych w języku JavaScript. Od samego początku istnienia języka aż do chwili obecnej podstawowymi mechanizmami używanymi do tworzenia struktur danych są tablice i obiekty. Oczywiście, powstawały także biblioteki, które korzystając z tablic i obiektów, implementowały struktury danych wyższego poziomu.

Jednak w ES6 niektóre najbardziej przydatne (i zoptymalizowane pod względem wydajności działania!) spośród tych abstrakcji danych zostały dodane jako macierzyste elementy języka.

Zaczniemy od przedstawienia **tablic określonego typu** (ang. *TypedArrays*), rozwiązania wprowadzonego kilka lat temu wraz z udostępnieniem wersji ES5, lecz zestandaryzowanego wyłącznie jako dodatek do WebGL, a nie do samego języka JavaScript. W przypadku ES6 tablice określonego typu zostały dodane bezpośrednio do specyfikacji języka, co nadało im najwyższy status.

Mapy przypominają nieco obiekty (pary klucz-wartość), jednak w ich przypadku kluczem może być nie tylko łańcuch znaków, lecz dowolna wartość — nawet inny obiekt lub mapa! Zbiory są podobne do tablic (list wartości), ale umieszczane w nich wartości są unikalne; w przypadku próby dodania duplikatu zostanie on pominięty. Istnieją także „słabe" (pod względem odzyskiwania i oczyszczania pamięci) odpowiedniki tych dwóch struktur danych: WeakMap oraz WeakSet.

Tablice określonego typu

Zgodnie z informacjami podanymi w książce *Tajniki języka JavaScript. Typy i gramatyka*, należącej do tej serii wydawniczej, język JavaScript posiada zestaw wbudowanych typów danych, takich jak `number` oraz `string`. Kuszące byłoby przypuszczenie, że termin „tablice określonego typu" oznacza tablicę wartości pewnego konkretnego typu, na przykład: tablicę zawierającą wyłącznie łańcuchy znaków.

Jednak w przypadku tablic określonego typu chodzi raczej o zapewnienie strukturalnego dostępu do danych binarnych przy wykorzystaniu składni charakterystycznej dla tablic (dostępu z użyciem indeksów itd.). Słowo „typ" w nazwie tego mechanizmu odnosi się raczej do „widoku" prezentującego zbiory bitów, stanowiącego w zasadzie odwzorowanie określające, czy bity powinny być traktowane jako tablica 8-bitowych liczb całkowitych ze znakiem, czy też jako 16-bitowe liczby całkowite ze znakiem itd.

W jaki sposób tworzymy taki „zbiór bitów”. Jest on nazywany „buforem”, a najbardziej bezpośredni sposób jego tworzenia polega na użyciu konstruktora `ArrayBuffer(..)`:

```
var buf = new ArrayBuffer( 32 );
buf.byteLength;                          // 32
```

Po wykonaniu powyższych instrukcji zmienna `buf` będzie zawierać bufor binarny o długości 32 bajtów (czyli 256 bitów), który początkowo został wypełniony zerami. Sam bufor nie daje nam żadnych możliwości interakcji z jego zawartością, a jedyną operacją, jaką możemy na nim wykonać, jest sprawdzenie jego wielkości przy użyciu właściwości `byteLength`.

Kilka mechanizmów różnych platform internetowych korzysta z takich buforów lub je zwraca; należą do nich: `FileReader#readAsArrayBuffer(..)`, `XMLHttpRequest#send(..)` oraz `ImageData` (dane płócien).

Jednak na taki bufor możemy następnie nałożyć „widok”, którym jest właśnie tablica określonego typu. Przyjrzyjmy się poniższemu przykładowi:

```
var arr = new Uint16Array( buf );
arr.length;                              // 16
```

Zmienna `arr` jest teraz tablicą określonego typu, prezentującą zawartość 256-bitowego bufora `buff` jako 16-bitowe liczby całkowite bez znaku, co oznacza, że będzie ona miała 16 elementów.

Kolejność zapisu bajtów

Koniecznie należy zrozumieć i zapamiętać, że tablica `arr` jest odwzorowywana z wykorzystaniem określonej kolejności zapisu bajtów (*big-endian* lub *little-endian*), odpowiadających ustawieniom platformy, na której działa środowisko JavaScript. Ten fakt może być problemem, gdy dane binarne zostaną przygotowane z użyciem innej kolejności zapisu bajtów, lecz muszą być interpretowane na platformie, w której kolejność zapisu bajtów jest przeciwna.

Oba terminy (*big-endian* oraz *little-endian*) określają, czy najmniej znaczący bajt (kolekcja 8 bitów) wielobajtowej liczby — takiej jak 16-bitowe liczby bez znaku, których użyliśmy w ostatnim przykładzie — zostanie zapisany z prawej, czy z lewej strony sekwencji bajtów reprezentujących liczbę.

Wyobraźmy sobie liczbę dziesiętną 3085, która musi być reprezentowana przy użyciu 16 bitów. Gdyby taka liczba miała być reprezentowana przy użyciu jednego, 16-bitowego pojemnika, to niezależnie od kolejności zapisu miałaby dwójkową postać `0000110000001101` (lub `0c0d` szesnastkowo).

Gdybyśmy jednak chcieli zapisać ją jako dwie liczby 8-bitowe, to zastosowana kolejność zapisu bajtów miałaby ogromne znaczenie dla postaci tej liczby umieszczonej w pamięci:

- `0000110000001101` / `0c0d` (w przypadku stosowania zapisu *big-endian*),
- `0000110100001100` / `0d0c` (w przypadku stosowania zapisu *little-endian*).

Jeśli pobraliśmy bity `0000110100001100` reprezentujące liczbę 3085 w systemie korzystającym z zapisu *little-endian*, lecz spróbujemy ich użyć w systemie korzystającym z zapisu *big-endian*, to okaże się, że uzyskamy liczbę 3340 (w systemie dziesiętnym) lub `0d0c` (w systemie szesnastkowym).

Zapis *little-endian* jest obecnie najczęściej używaną w internecie reprezentacją danych, choć oczywiście istnieją przeglądarki, w których nie jest on stosowany. Bardzo ważne jest, by zrozumieć, że znaczenie ma sposób zapisu danych binarnych używany zarówno przez ich producenta, jak i przez konsumenta.

Oto prosta metoda sprawdzania używanej kolejności zapisu danych, przedstawiona na witrynie MDN:

```
var littleEndian = (function() {
    var buffer = new ArrayBuffer( 2 );
    new DataView( buffer ).setInt16( 0, 256, true );
    return new Int16Array( buffer )[0] === 256;
})();
```

Zmienna `littleEndian` będzie przyjmować wartości `true` lub `false`, w większości przeglądarek powinna mieć wartość `true`. Powyższy kod korzysta z konstruktora `DataView(..)` zapewniającego najwyższy i najdokładniejszy poziom kontroli nad dostępnością (ustawianiem i pobieraniem) bitów z widoku utworzonego na bazie bufora. Trzeci parametr metody `setInt16(..)` zastosowanej w powyższym kodzie informuje `DataView`, jaka kolejność zapisu danych ma być zastosowana w wykonywanej operacji.

Nie należy mylić kolejności zapisu danych binarnych przechowywanych w buforze tablicy ze sposobem reprezentacji danej liczby po udostępnieniu jej w programie JavaScript. Na przykład wywołanie `(3085).toString(2)` zwróci łańcuch znaków `"110000001101"`, który przy pominięciu czterech początkowych `0` wydaje się reprezentacją używającą zapisu *little-endian*. W rzeczywistości reprezentacja ta bazuje na pojedynczym widoku 16-bitowym, a nie na widoku dwóch bajtów (po 8 bitów każdy). Powyższe rozwiązanie stanowi najlepszy sposób sprawdzenia kolejności zapisu bajtów stosowanej przez używane środowisko.

Wiele widoków

Do jednego bufora można dołączyć wiele różnych widoków. Oto przykład:

```
var buf = new ArrayBuffer( 2 );

var view8 = new Uint8Array( buf );
var view16 = new Uint16Array( buf );

view16[0] = 3085;
view8[0];                       // 13
view8[1];                       // 12

view8[0].toString( 16 );        // "d"
view8[1].toString( 16 );        // "c"

// zamiana (jakby zmiana kolejności zapisu!)
var tmp = view8[0];
view8[0] = view8[1];
view8[1] = tmp;

view16[0];                      // 3340
```

Konstruktory tablic określonego typu mają wiele różnych sygnatur. Na razie przedstawione zostały tylko te, do których przekazywany jest sam bufor. Niemniej jednak ta forma konstruktorów umożliwia także podanie dwóch opcjonalnych parametrów: `byteOffset` oraz `length`. Oznacza to, że tworzony widok tablicy określonego typu może się zaczynać w innym miejscu niż wskazywane przez indeks `0` i może obejmować mniejszy zakres danych niż pełna długość bufora.

Technika ta może się okazać całkiem przydatna, jeśli dane binarne umieszczone w buforze nie mają jednolitej wielkości bądź rozmieszczenia.

Załóżmy, że dysponujemy binarnym buforem, w którym najpierw jest zapisana liczba składająca się z dwóch bajtów (tak zwane „słowo"), następnie dwie liczby jednobajtowe, a następnie liczba zmiennoprzecinkowa o długości czterech bajtów. Poniższy przykład pokazuje, w jaki sposób można pobrać te wszystkie dane, korzystając z różnych widoków bazujących na tym samym buforze, lecz używających różnych przesunięć oraz długości:

```
var first = new Uint16Array( buf, 0, 2 )[0],
    second = new Uint8Array( buf, 2, 1 )[0],
    third = new Uint8Array( buf, 3, 1 )[0],
    fourth = new Float32Array( buf, 4, 4 )[0];
```

Konstruktory tablic określonego typu

Oprócz formy z parametrami `(buffer, [offset, [length]])`, przedstawionej w poprzednich przykładach, dostępne są także inne postaci konstruktorów tablic określonego typu:

- *`[konstruktor]`*`(length)` — tworzy nowy widok na bazie nowego bufora o długości `length`.
- *`[konstruktor]`*`(typedArr)` — tworzy nowy widok oraz nowy bufor, kopiując do niego zawartość przekazanego widoku `typedArr`.
- *`[konstruktor]`*`(obj)` — tworzy nowy widok oraz nowy bufor, do którego zostają skopiowane dane pobrane podczas przeglądania obiektu lub obiektu przypominającego tablicę.

Na poniższej liście przedstawiłem wszystkie konstruktory tablic określonego typu dostępne w języku ES6:

- `Int8Array` (8-bitowe liczby całkowite ze znakiem), `Uint8Array` (8-bitowe liczby całkowite bez znaku),
  - `Uint8ClampedArray` (8-bitowe liczby całkowite bez znaku, przy czym podczas inicjalizacji każda wartość jest ograniczana do zakresu od 0 do 255),
- `Int16Array` (16-bitowe liczby całkowite ze znakiem), `Uint16Array` (16-bitowe liczby całkowite bez znaku),
- `Int32Array` (32-bitowe liczby całkowite ze znakiem), `Uint32Array` (32-bitowe liczby całkowite bez znaku),
- `Float32Array` (32-bitowe liczby zmiennoprzecinkowe, IEEE-754),
- `Float64Array` (64-bitowe liczby zmiennoprzecinkowe, IEEE-754).

Tablice określonego typu tworzone przy użyciu tych konstruktorów są niemal takie same jak zwyczajne, macierzyste tablice języka JavaScript. Różnica pomiędzy nimi polega na tym, że tablice określonego typu mają stałą, niezmienną długość, a ich zawartość stanowią wartości tego samego „typu".

Poza tymi rozbieżnościami oba rodzaje tablic dysponują tymi samymi metodami udostępnianymi przez ich konstruktory. Z tego względu zapewne będziemy mogli używać ich jako zwyczajnych tablic bez konieczności przeprowadzania jakichkolwiek konwersji.

Oto przykład:

```
var a = new Int32Array( 3 );
a[0] = 10;
a[1] = 20;
a[2] = 30;

a.map( function(v){
    console.log( v );
} );
// 10 20 30

a.join( "-" );
// "10-20-30"
```

Podczas korzystania z tablic określonego typu nie można używać niektórych metod `Array.prototype`; chodzi o metody, których zastosowanie nie ma w danym przypadku sensu, takie jak mutatory (`splice(..)`, `push(..)` itd.) oraz metoda `concat(..)`.

Trzeba pamiętać, że wielkość elementów tablic określonego typu jest ograniczona przez zadeklarowany typ tablicy. Jeśli używamy tablicy `Uint8Array` i spróbujemy zapisać w jednym z jej elementów daną, której wielkość przekracza 8 bitów, to zostanie ona przycięta tak, by liczba bajtów zapisywanej wartości odpowiadała zadeklarowanej wielkości elementów tablicy.

W niektórych przypadkach, na przykład w razie próby podniesienia do kwadratu liczb zapisanych w tablicy określonego typu, może to być przyczyną problemów. Przeanalizujmy następujący przykład:

```
var a = new Uint8Array( 3 );
a[0] = 10;
a[1] = 20;
a[2] = 30;

var b = a.map( function(v){
    return v * v;
} );

b;                    // [100, 144, 132]
```

Podniesienie do kwadratu wartości 20 oraz 30 powoduje wystąpienie przepełnienia bitowego. Ograniczenie to można ominąć, korzystając z funkcji *`TypedArray`*`#from(..)`:

```
var a = new Uint8Array( 3 );
a[0] = 10;
a[1] = 20;
a[2] = 30;

var b = Uint16Array.from( a, function(v){
    return v * v;
} );

b;                    // [100, 400, 900]
```

Więcej informacji na temat funkcji `Array.from(..)`, dostępnej także w tablicach określonego typu, można znaleźć w rozdziale 6., w punkcie „Funkcja statyczna Array.from(..)”. W podpunkcie „Odwzorowania” opisana została funkcja odwzorowująca, którą można przekazać jako drugi argument funkcji `Array.from(..)`.

Jedną z interesujących cech tablic określonego typu jest udostępnianie metody `sort(..)`, analogicznej do metody dostępnej w normalnych tablicach języka JavaScript. Jednak w przypadku tablic określonego typu metoda ta domyślnie porównuje liczby, a nie konwertuje wartości do postaci łańcuchowej w celu wykonania porównania alfabetycznego. Oto przykład zastosowania obu tych metod:

```
var a = [ 10, 1, 2, ];
a.sort();                                // [1,10,2]

var b = new Uint8Array( [ 10, 1, 2 ] );
b.sort();                                // [1,2,10]
```

Metoda *`TypedArray`*`#sort(..)` dysponuje także drugim, opcjonalnym parametrem — można go użyć do przekazania funkcji porównującej, która działa dokładnie tak samo jak funkcje porównujące stosowane w metodzie `Array#sort(..)`.

Mapy

Jeśli masz duże doświadczenie w programowaniu w języku JavaScript, to zapewne wiesz, że obiekty są podstawowym mechanizmem do tworzenia struktur danych zawierających nieuporządkowane pary klucz-wartość, nazywanych także mapami. Niemniej jednak podstawową wadą związaną z tworzeniem map przy wykorzystaniu obiektów jest to, że kluczami obiektów mogą być wyłącznie łańcuchy znaków.

Przeanalizujmy następujący przykład:

```
var m = {};

var x = { id: 1 },
    y = { id: 2 };

m[x] = "foo";
m[y] = "bar";

m[x];                                    // "bar"
m[y];                                    // "bar"
```

Co się tu dzieje? Otóż oba obiekty, `x` i `y`, zostają przekształcone do postaci łańcucha `"[object Object]"`, dlatego w obiekcie `m` zostanie utworzony tylko jeden klucz.

Niektórzy implementowali fałszywe mapy, tworząc dwie tablice, z których jedna zawierała klucze niebędące łańcuchami znaków, a druga — wartości. Oto przykład takiego rozwiązania:

```
var keys = [], vals = [];

var x = { id: 1 },
    y = { id: 2 };

keys.push( x );
vals.push( "foo" );
```

```
keys.push( y );
vals.push( "bar" );

keys[0] === x;                    // true
vals[0];                          // "foo"

keys[1] === y;                    // true
vals[1];                          // "bar"
```

Oczywiście nikt nie chciałby własnoręcznie zarządzać takimi równoległymi tablicami, dlatego zapewne warto by zdefiniować strukturę danych, której metody ukrywałyby szczegóły implementacyjne związane z zarządzaniem tymi tablicami. Oprócz konieczności zaimplementowania takiego rozwiązania jego kolejną podstawową wadą jest to, że złożoność działania takiej struktury danych nie jest już liniowa — O(1) — lecz wynosi O(n).

Na szczęście w języku ES6 nie ma już potrzeby stosowania takich rozwiązań! Wystarczy utworzyć mapę — `Map(..)`:

```
var m = new Map();

var x = { id: 1 },
    y = { id: 2 };

m.set( x, "foo" );
m.set( y, "bar" );

m.get( x );                       // "foo"
m.get( y );                       // "bar"
```

Jedyną wadą jest tu brak możliwości ustawiania i pobierania wartości mapy przy użyciu zapisu z nawiasami kwadratowymi (`[ ]`), charakterystycznego dla tablic. Zamiast tego operacje te trzeba wykonywać odpowiednio przy użyciu metod `set(..)` i `get(..)`.

Do usuwania elementów mapy nie należy używać operatora `delete`, lecz metody `delete(..)`:

```
m.set( x, "foo" );
m.set( y, "bar" );

m.delete( y );
```

Całą zawartość mapy można usunąć przy użyciu metody `clear()`. Z kolei długość mapy (czyli liczbę jej kluczy) można pobrać, używając właściwości `size` (nie `length`):

```
m.set( x, "foo" );
m.set( y, "bar" );
m.size;                           // 2

m.clear();
m.size;                           // 0
```

W wywołaniu konstruktora `Map(..)` można także przekazać obiekt iterowalny (więcej informacji na temat takich obiektów podałem w rozdziale 3., w podrozdziale „Iteratory”), który musi zwracać listę tablic, przy czym pierwszy element każdej tablicy jest kluczem, a drugi — skojarzoną z nim wartością. Ten format iteracji idealnie odpowiada efektom działania metody `entries()`, opisanej w następnym punkcie rozdziału. Dzięki temu bardzo łatwo można tworzyć kopie map:

```
var m2 = new Map( m.entries() );

// ma ten sam efekt co:
var m2 = new Map( m );
```

Ponieważ instancja mapy jest obiektem iterowalnym, a jej domyślny iterator działa tak samo jak metoda `entries()`, to preferowany jest drugi, krótszy sposób wywoływania konstruktora.

Oczywiście istnieje także możliwość ręcznego przekazania do konstruktora `Map(..)` listy *elementów* mapy (czyli tablicy dwuelementowych tablic zawierających pary klucz-wartość). Takie rozwiązanie przedstawia poniższy przykład:

```
var x = { id: 1 },
    y = { id: 2 };

var m = new Map( [
    [ x, "foo" ],
    [ y, "bar" ]
] );

m.get( x );                         // "foo"
m.get( y );                         // "bar"
```

Wartości map

W celu pobrania wartości mapy należy wywołać metodę `values(..)`, która zwraca iterator. W rozdziałach 2. i 3. poznaliśmy różne metody sekwencyjnego przetwarzania iteratorów (czyli podobnego do sposobów korzystania z tablic), na przykład z użyciem operatora `...` lub pętli `for .. of`. Informacje na temat metody `Array.from(..)` można także znaleźć w rozdziale 6., w punkcie „Tworzenie tablic i podtypów". Przeanalizujmy poniższy przykład:

```
var m = new Map();

var x = { id: 1 },
    y = { id: 2 };

m.set( x, "foo" );
m.set( y, "bar" );

var vals = [ ...m.values() ];

vals;                               // ["foo","bar"]
Array.from( m.values() );           // ["foo","bar"]
```

Zgodnie z informacjami podanymi w poprzednim rozdziale, elementy mapy można przejrzeć, korzystając z metody `entries()` (bądź domyślnego iteratora mapy). Oto przykład wykorzystania tej metody:

```
var m = new Map();

var x = { id: 1 },
    y = { id: 2 };

m.set( x, "foo" );
m.set( y, "bar" );

var vals = [ ...m.entries() ];
```

```
vals[0][0] === x;                    // true
vals[0][1];                          // "foo"

vals[1][0] === y;                    // true
vals[1][1];                          // "bar"
```

Klucze map

W celu pobrania wartości mapy należy wywołać metodę `keys(..)`, która zwraca iterator udostępniający wszystkie klucze mapy:

```
var m = new Map();

var x = { id: 1 },
    y = { id: 2 };

m.set( x, "foo" );
m.set( y, "bar" );

var keys = [ ...m.keys() ];

keys[0] === x;                       // true
keys[1] === y;                       // true
```

Z kolei metoda `has(..)` pozwala sprawdzić, czy mapa zawiera konkretny klucz:

```
var m = new Map();

var x = { id: 1 },
    y = { id: 2 };

m.set( x, "foo" );

m.has( x );                          // true
m.has( y );                          // false
```

W gruncie rzeczy mapy pozwalają nam skojarzyć obiektem (kluczem) jakąś informację (wartość) bez faktycznego zapisywania tej informacji w danym obiekcie.

Choć kluczami map mogą być dowolne wartości, to jednak zazwyczaj będą nimi obiekty, gdyż łańcuchy znaków oraz wartości innych typów prostych mogą być stosowane jako klucze zwyczajnych obiektów. Innymi słowy, najprawdopodobniej wciąż będziemy używali zwyczajnych obiektów zamiast map, wyjąwszy sytuacje, gdy niektóre lub wszystkie klucze obiektu także muszą być obiektami — zastosowanie mapy będzie wówczas lepszym rozwiązaniem.

Jeśli użyjemy obiektu jako klucza mapy, a następnie obiekt ten zostanie porzucony (zostaną usunięte wszystkie odwołania do niego) w ramach próby odzyskania zajmowanej przez niego pamięci, to element mapy nie zostanie usunięty. W celu umożliwienia odzyskania pamięci zajmowanej przez element mapy trzeba go z niej jawnie usunąć. W następnym punkcie, poświęconym mapom typu `WeakMap`, poznamy lepsze rozwiązanie, jeśli chodzi o stosowanie obiektów jako kluczy i odzyskiwanie pamięci.

Mapy typu WeakMap

Tak zwane „słabe mapy”, `WeakMap`, to specjalny rodzaj map, który dysponuje w zasadzie takim samym zewnętrznym zachowaniem jak normalne mapy, lecz różni się od nich pod względem wewnętrznego sposobu przydzielania pamięci (a zwłaszcza możliwości jej odzyskiwania).

W przypadku map typu `WeakMap` kluczami mogą być wyłącznie obiekty. Obiekty te są przechowywane w **słaby** sposób, co oznacza, że w przypadku ich usunięcia przez mechanizm odzyskiwania pamięci (GC) usunięty zostanie także cały element mapy. Tego działania nie można jednak zaobserwować, gdyż jedynym sposobem, by mechanizm odzyskiwania pamięci usunął obiekt, jest usunięcie wszystkich referencji do tego obiektu — to z kolei oznacza, że w programie nie będzie żadnych referencji, które pozwoliłyby sprawdzić, czy dany obiekt istnieje w mapie, czy nie.

Pod wszelkimi innymi względami interfejs API mapy typu `WeakMap` w dużym stopniu przypomina interfejs zwyczajnych map, choć jest nieco bardziej ograniczony:

```
var m = new WeakMap();

var x = { id: 1 },
    y = { id: 2 };

m.set( x, "foo" );

m.has( x );                         // true
m.has( y );                         // false
```

Mapy typu `WeakMap` nie udostępniają właściwości `size`, metody `clear()` ani iteratorów pozwalających na przeglądanie ich kluczy, wartości oraz elementów. A zatem nawet jeśli usuniemy referencję `x`, powodując tym samym usunięcie skojarzonego z nią elementu mapy `m` podczas następnego cyklu odzyskiwania pamięci, to i tak nie będziemy mogli się o tym przekonać. Pozostanie nam jedynie wierzyć na słowo!

Mapy typu `WeakMap`, podobnie jak zwyczajne mapy, pozwalają na swoiste kojarzenie informacji z obiektami. Są one szczególnie przydatne, kiedy nie dysponujemy całkowitą kontrolą nad używanymi obiektami, na przykład gdy są nimi elementy DOM. A zatem, jeśli obiekty używane jako klucze map mogą być usuwane i zajmowana przez nie pamięć powinna zostać zwolniona w ramach procesu odzyskiwania pamięci (GC), to zastosowanie mapy typu `WeakMap` będzie optymalnym rozwiązaniem.

Koniecznie należy zapamiętać, że w przypadku map typu `WeakMap` jedynie *klucze* są przechowywane w „słaby” sposób — nie dotyczy to wartości mapy. Przeanalizujmy poniższy przykład:

```
var m = new WeakMap();

var x = { id: 1 },
    y = { id: 2 },
    z = { id: 3 },
    w = { id: 4 };

m.set( x, y );

x = null;                   // { id: 1 } może zostać usunięty przez GC
y = null;                   // { id: 2 } może zostać usunięty przez GC
                            // ale tylko dlatego, że dotyczy to także obiektu { id: 1 }
```

```
m.set( z, w );
w = null;                    // { id: 4 } nie może zostać usunięty przez GC
```

Właśnie z tego powodu uważam, że odpowiedniejszą nazwą dla map tego typu byłoby `WeakKeyMap` — „mapy o słabych kluczach”.

Zbiory

Zbiory są kolekcjami unikalnych wartości (wszelkie duplikaty są pomijane).

Interfejs API zbiorów przypomina interfejs map. Zamiast metody `set(..)` zbiory używają metody `add(..)` (co można by potraktować jako swoisty żart), a poza tym nie udostępniają metody `get(..)`.

Przeanalizujmy poniższy przykład:

```
var s = new Set();

var x = { id: 1 },
    y = { id: 2 };

s.add( x );
s.add( y );
s.add( x );

s.size;                      // 2

s.delete( y );
s.size;                      // 1

s.clear();
s.size;                      // 0
```

Konstruktor `Set(..)` przypomina nieco konstruktor `Map(..)`, gdyż tak jak w przypadku map, można do niego przekazać daną iterowalną, taką jak inny zbiór lub zwyczajna tablica wartości. Jednak w odróżnieniu od konstruktora `Map(..)`, do którego należy przekazać listę *elementów* (tablicę tablic reprezentujących pary klucz-wartość), konstruktor `Set(..)` oczekuje przekazania listy *wartości* (tablicy wartości):

```
var x = { id: 1 },
    y = { id: 2 };

var s = new Set( [x,y] );
```

Zbiory nie potrzebują metody `get(..)`, gdyż w ich przypadku nie wykonujemy operacji pobierania wartości ze zbioru, a jedynie sprawdzamy, czy konkretna wartość w nim występuje:

```
var s = new Set();

var x = { id: 1 },
    y = { id: 2 };

s.add( x );

s.has( x );                  // true
s.has( y );                  // false
```

Algorytm porównywania używany przez metodę `has(..)` jest niemal identyczny z algorytmem stosowanym przez funkcję `Object.is(..)` (informacje na jego temat można znaleźć w rozdziale 6.), a jedyna różnica polega na tym, że wartości `-0` i `0` są traktowane jako identyczne, a nie różne.

Iteratory zbiorów

Zbiory udostępniają te same metody zwracające iteratory co mapy. W przypadku zbiorów działanie tych metod jest nieco inne, choć analogiczne do działania iteratorów map. Przeanalizujmy następujący przykład:

```
var s = new Set();

var x = { id: 1 },
    y = { id: 2 };

s.add( x ).add( y );

var keys = [ ...s.keys() ],
    vals = [ ...s.values() ],
    entries = [ ...s.entries() ];

keys[0] === x;
keys[1] === y;

vals[0] === x;
vals[1] === y;

entries[0][0] === x;
entries[0][1] === x;
entries[1][0] === y;
entries[1][1] === y;
```

Iteratory `keys()` oraz `values()` zwracają listę wszystkich unikalnych wartości zbioru. Z kolei iterator `entries()` zwraca listę tablic reprezentujących elementy zbioru, przy czym oba elementy tych tablic są unikalną wartością zbioru. Domyślnym iteratorem zbiorów jest `values()`.

Najużyteczniejszą cechą zbiorów jest charakterystyczna dla nich unikalność. Przeanalizujmy poniższy przykład:

```
var s = new Set( [1,2,3,4,"1",2,4,"5"] ),
    uniques = [ ...s ];

uniques;                                // [1,2,3,4,"1","5"]
```

Mechanizm określania unikalności wartości zapisywanych w zbiorze nie korzysta z konwersji typów, dlatego `1` i `"1"` są traktowane jako odrębne, unikalne wartości.

Zbiory typu WeakSet

W odróżnieniu od map typu `WeakMap`, które w „słaby” sposób przechowują klucze (wartości są natomiast przechowywane „mocno”), zbiory typu `WeakSet` w „słaby” sposób przechowują wartości (klucze w zasadzie nie są w nich stosowane).

```
var s = new WeakSet();

var x = { id: 1 },
    y = { id: 2 };

s.add( x );
s.add( y );

x = null;                    // obiekt 'x' może zostać usunięty przez GC
y = null;                    // obiekt 'y' może zostać usunięty przez GC
```

Wartościami zbiorów typu `WeakSet` muszą być obiekty — w odróżnieniu od zwyczajnych zbiorów zapisywanie w nich wartości typów prostych nie jest dozwolone.

Podsumowanie

W języku ES6 dodanych zostało kilka użytecznych kolekcji, dzięki którym operowanie na danych w strukturalny sposób stało się prostsze i wydajniejsze.

Tablice określonego typu tworzą „widoki” buforów zawierających dane binarne, dzięki którym dane te są prezentowane jako wartości różnych typów liczbowych, na przykład: 8-bitowe liczby całkowite bez znaku lub 32-bitowe liczby zmiennoprzecinkowe. Możliwość dostępu do danych binarnych z użyciem zapisu tablicowego sprawia, że wykonywane operacje można zapisać w znacznie prostszej postaci, ułatwiając tym samym pielęgnację kodu oraz korzystanie ze złożonych danych, takich jak dane audiowizualne, dane płócien itd.

Mapy są kolekcjami para klucz-wartość, przy czym pozwalają, by kluczami były także obiekty, a nie tylko wartości typów prostych i łańcuchy znaków. Zbiory są listami unikalnych wartości (dowolnego typu).

Mapy typu `WeakMap` przechowują swoje klucze (obiekty) w słaby sposób, dzięki czemu mechanizm odzyskiwania pamięci może usunąć element mapy, jeśli to ona zawiera ostatnią referencję do danego klucza. Z kolei zbiory typu `WeakSet` w podobny — słaby — sposób przechowują swoje wartości, dzięki czemu, jeśli w zbiorze jest przechowywana ostatnia referencja do obiektu, to odpowiadający mu element zbioru może zostać usunięty przez mechanizm odzyskiwania pamięci.

Skorowidz

G

I

J

K

L

Ł

M

N

O

P

R

S

T

U

W

Z

DOWIEDZ SIĘ, JAKI BĘDZIE JS PRZYSZŁOŚCI!

JavaScript

Na pozór JavaScript jest prostym językiem o atrakcyjnych możliwościach. Jego złożone wewnętrzne mechanizmy muszą jednak zostać dokładnie przestudiowane, aby poczucie *prawdziwego zrozumienia* języka nie okazało się złudne. Zrozumienie subtelności JS jest o tyle ważne, że język ten ewoluuje. Najnowszy standard ECMAScript 6 to gwałtowny skok w przód i ogromna zmiana jakościowa, którą programista JS musi *bardzo dobrze* poznać!

Niniejsza publikacja jest częścią serii w całości poświęconej temu językowi. Przed lekturą warto poznać koncepcje opisane w poprzednich książkach tej serii, gdyż w tym tomie autor koncentruje się na nowych możliwościach standardu ES6, m.in. na nowych formach składniowych, różnorodnych formach organizacji kodu czy wspomagających interfejsach API. Szczególny nacisk położono na trudniejsze aspekty języka JavaScript, których wielu programistów unika lub w ogóle nie zna.

Dzięki tej książce poznasz:

- zasady stosowania nowej składni
- zasady organizowania kodu z wykorzystaniem iteratorów, generatorów, modułów oraz klas
- zasady asynchronicznego sterowania przepływem za pomocą obietnic i generatorów
- nowe metody pomocnicze obiektów macierzystych: Array, Object, Math, Number i String
- sposoby wykorzystania technik metaprogramowania
- plany rozwoju kolejnych wersji języka

KYLE SIMPSON

— jest Teksańczykiem, propagatorem Open Web i wielkim pasjonatem wszystkiego, co związane z językiem JavaScript. Ma dar przekazywania wiedzy, a przy tym zaraża entuzjazmem. Pisze książki, prowadzi warsztaty, występuje na konferencjach o tematyce technicznej oraz pozostaje aktywnym członkiem społeczności OSS.

Helion

46960 numer katalogowy

księgarnia internetowa

http://helion.pl

zamówienia telefoniczne

0 801 339900

0 601 339900

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 978-83-283-2311-7

9 788328 323117

cena: 39,90 zł

O'REILLY®