Tomasz Sochacki

JAVASCRIPT

TECHNIKI ZAAWANSOWANE

Poznaj bliżej możliwości JavaScript
Naucz się z nim pracować szybciej i skuteczniej
Przejdź na zaawansowany poziom webdesignu

Helion

Redaktor prowadzący: Grzegorz Krzystek
Projekt okładki: Studio Gravite

Helion S.A.
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/zaazaj
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

ISBN: 978-83-283-5640-5

Printed in Poland.

Spis treści

Rozdział 1.

Obiekty Map i Set

Jeśli masz doświadczenie w programowaniu w języku JavaScript, to zapewne najczęściej do tworzenia struktur danych stosowałeś do tej pory obiekty. Są one nieuporządkowanym zbiorem zawierającym pary *klucz-wartość*. Obiekty mają jednak pewne ograniczenia, które czasami powodują problemy. W tym rozdziale omówię kilka innych struktur danych, które mogą być ciekawą alternatywą dla tradycyjnych obiektów. Nie oznacza to jednak, że od tej pory masz zawsze używać tylko struktur Map czy Set. Każdy typ danych ma swoje przeznaczenie i prawdopodobnie po przeczytaniu tego rozdziału nadal w większości sytuacji będziesz stosował głównie obiekty, jednakże w pewnych specyficznych przypadkach będziesz umiał wykorzystać inne struktury, co może istotnie ułatwić pracę.

Wady standardowych obiektów

Standardowe obiekty w języku JavaScript mają pewne ograniczenia, do których zalicza się najczęściej fakt, że klucze muszą być ciągami znakowymi; nie mogą to być np. inne obiekty. Obecnie klucze mogą być co prawda również obliczane, jednakże wynikowo i tak musi to być ciąg znakowy. Przykładowe obiekty można więc deklarować w następujący sposób:

```
const obj = {
  key1: 'value1',
  key2: 'value2',
  key3: 'value3',
};
```

Lub przy użyciu obliczanych wartości dla kluczy:

```
const someKey = 'xyz';
const obj = {
  [someKey]: 'some value'
};
Obj.xyz; // "some value"
```

Niekiedy jednak możemy chcieć powiązać jakąś inną strukturę niż ciąg znakowy, np. obiekt z jakąś wartością. W przypadku zwykłych obiektów JavaScript próba takiego przypisania spowoduje błąd:

```
const person = {
  name: 'Jan',
  age: 35
```

```
};
const obj = {
  [person]: 'some value'
};

Obj.person;      // undefined
Obj['person']; // undefined
Obj;             // {[object Object]: "some value"}
```

Zastanówmy się nad powyższym kodem. Stworzyliśmy stałą `person`, która jest obiektem z jakimiś właściwościami. Następnie w obiekcie `obj` chcielibyśmy skojarzyć obiekt `person` z wartością „`some value`". Zastosowaliśmy w tym celu obliczoną wartość klucza. Ostatecznie jednak obiekt `obj` nie zawiera żadnej właściwości o nazwie `person`, zawiera natomiast klucz „`[object Object]`". Widzimy tutaj wyraźnie, że język JavaScript zawsze wymaga, aby klucz był ciągiem znakowym, i w razie potrzeby konwertuje obliczoną wartość na postać string. Każdy obiekt, na którym wywołamy metodę `toString`, zwróci taki właśnie ciąg znakowy. Nie jest to jednak zachowanie, które chcieliśmy osiągnąć.

Kolejny problem to fakt, że obiekty domyślnie nie są strukturami iterowalnymi. Mogłoby się wydawać, że skoro jest to zbiór par *klucz-wartość*, to nie powinno być problemu z przeiterowaniem po takiej strukturze, jednakże tak nie jest. Co więcej, obiekty są strukturami nieuporządkowanymi; poszczególne pary nie mają indeksów, po których można by się do nich odnieść i tym samym poznać konkretną kolejność, w jakiej się znajdują. Spróbujmy przeiterować po obiekcie znaną Ci już metodą `forEach`:

```
typeof obj.forEach; // "undefined"
```

Widzimy jednak, że obiekty nie mają tej metody, gdyż nie znajduje się ona w `Object.proto`
`↪type`, czyli prototypie wszystkich obiektów w języku JavaScript. Spróbujmy więc wykorzystać pętlę `for-of`:

```
for (const key of obj) {
  console.log(key);
};
// Uncaught TypeError: obj is not iterable
```

Tym razem próba również kończy się niepowodzeniem i wyraźnie widzimy komunikat mówiący o tym, że obiekt nie jest iterowalny. Mamy jednak do dyspozycji kilka metod globalnych obiektu `Object`, które pozwalają pobrać klucze, wartości lub pary *klucz-wartość* z obiektu i zwrócić je w formie tablicy, która ma znane Ci już metody `Array.prototype`, takie jak `forEach`, `map`, `filter`:

```
Object.keys(obj);    // ["key1", "key2", "key3"]
Object.values(obj);  // ["value1", "value2", "value3"]
Object.entries(obj); // [["key1", "value1"], ["key2", "value2"], ["key3", "value3"]]
```

W wielu przypadkach wykorzystanie tych metod w zupełności wystarczy, jednakże czasami wygodniejsze byłoby operowanie na iterowalnej strukturze, co zapewniają nam typy `Map` i `Set`. Zanim jednak zaczniesz nagminnie stosować obiekty `Map` w celu uzyskania możliwości łatwej iteracji, warto zastanowić się, czy w tym przypadku na pewno tego typu struktura jest odpowiednia. Często wystarczy bowiem zwykła struktura `Array`.

W dalszej części tej książki omówię dokładniej struktury iterowalne i generatory, jednakże już teraz możemy spróbować „dodać iterowalność" do standardowego obiektu JavaScript:

```
const obj = {
  key_1: 'value 1',
  key_2: 'value 2',
  [Symbol.iterator]: function* () {
    for (let key in this) {
      yield [key, this[key]]
    }
  }
};

for (const [key, value] of obj) {
  console.log(`${key} | ${value}`);
}
// key_1 | value 1
// key_2 | value 2
```

Tworzymy tutaj zwykły obiekt JavaScript z dwoma kluczami: `key_1` i `key_2`, z przypisaniem im jakieś wartości typu string. Dodatkowo tworzymy także klucz `Symbol.iterator`, któremu przypisujemy funkcję (generator) zwracającą kolejne wartości w formie tablicy `[key, value]`. W tym momencie możemy przeiterować po tym obiekcie pętlą `for-of`. Widzimy jednak, że kod ten jest nieco skomplikowany, i z pewnością nie chcielibyśmy dodawać go do każdego obiektu, który chcielibyśmy iterować.

W tym przypadku w zasadzie taki sam efekt moglibyśmy osiągnąć przy użyciu metody `Object.entries`:

```
Object.entries(obj).forEach(([key, value]) => console.log(`${key} | ${value}`));
```

Nie są to jednak do końca takie same rozwiązania, ale temat ten omówię dokładniej przy analizie iteratorów i generatorów.

Kolejną trudnością w przypadku zwykłych obiektów jest pozyskanie liczby właściwości takiej struktury. Załóżmy, że chcemy się dowiedzieć, ile pól zawiera obiekt:

```
const obj = {
  key_1: 'value 1',
  key_2: 'value 2',
};
```

W przypadku zwykłych obiektów pozostaje nam najczęściej wykorzystanie metody `Object.keys`:

```
Object.keys(obj).length; // 2
```

Efekt co prawda jest taki, jakiego oczekiwaliśmy, jednakże nie jest to zbyt efektywny sposób, gdyż wymaga wykonania iteracji po całym obiekcie w celu wyciągnięcia jego kluczy, a następnie zwrócenia listy kluczy w formie tablicy. Obie te akcje są w zasadzie zbędne, gdyż nas interesuje tylko i wyłącznie liczba kluczy w obiekcie.

Podstawowe informacje o strukturach typu Map

Struktury `Map` również są listą par *klucz-wartość*, jednakże kluczem mogą być dowolne wartości, a nie tylko ciągi znakowe string, jak w przypadku obiektów (lub symbole). Ponadto lista ta jest listą uporządkowaną. Rozważmy prosty przykład, w którym tworzymy trzy obiekty reprezentujące jakieś przedmioty, i następnie każdy z nich łączymy z określonym rodzajem zniżki:

```
const itemA = { id: '1', price: "50.00" };
const itemB = { id: '2', price: "10.00" };
const itemC = { id: '3', price: "5.00" };

const itemsWithDiscounts = new Map();
itemsWithDiscounts.set(itemA, { discountType: 'type_1' });
itemsWithDiscounts.set(itemB, { discountType: 'type_2' });

itemsWithDiscounts.get(itemA); // {discountType: "type_1"}
itemsWithDiscounts.get(itemB); // {discountType: "type_2"}
itemsWithDiscounts.get(itemC); // undefined
```

Zauważ, w jaki sposób tworzona jest struktura `Map`. Na początku używamy konstruktora `new Map()`, co pozwala utworzyć pustą mapę. Następnie wykorzystujemy metodę `set` w celu dodania kolejnych par *klucz-wartość*. W tym przypadku kluczami nie są jednak stringi, lecz obiekty `itemA` oraz `itemB`.

W celu odczytania wartości dla danego klucza używamy metody `get`, która przyjmuje tylko jeden parametr — klucz. Zwróć jednak uwagę, że parametr ten również nie jest stringiem, lecz po prostu wskazaniem na obiekt.

Dla obiektu `itemC` nie utworzyliśmy w naszej mapie żadnej pary *klucz-wartość*, dlatego próba odczytu wartości dla tego klucza zwraca znaną nam w świecie JavaScriptu wartość `undefined`.

W przypadku typów `Map` nie mamy możliwości stosowania zapisu z użyciem nawiasów kwadratowych, do czego jesteśmy przyzwyczajeni, pracując z tablicami:

```
itemsWithDiscounts[itemA]; // undefined
```

Zamiast tego musimy korzystać ze zdefiniowanego do tego celu API, stosując metody `set` oraz `get`. Nie jest to jednak istotne utrudnienie, szczególnie gdy wykorzystamy w danej sytuacji zalety, jakie zapewnia nam struktura `Map`.

W każdej chwili możemy w prosty sposób sprawdzić liczbę elementów mapy, bez konieczności wykonywania iteracji po jej elementach, jak w przypadku standardowych obiektów. W tym celu wystarczy wykorzystać właściwość `size`:

```
itemsWithDiscounts.size; // 2
```

Zwróć jednak uwagę, że `size` jest właściwością struktury `Map`, a nie metodą (brak nawiasów, inaczej niż w wywołaniu metody czy funkcji).

Z kolei usuwanie elementów odbywa się za pomocą przeznaczonej do tego metody `delete` lub `clear`, jeśli chcemy wyczyścić całą strukturę z istniejących w niej elementów:

```
itemsWithDiscounts.size; // 2

itemsWithDiscounts.delete(itemB); // true
itemsWithDiscounts.size; // 1

itemsWithDiscounts.clear();
itemsWithDiscounts.size; // 0

itemsWithDiscounts.delete('some-value'); // false
```

Metoda `delete` przyjmuje jako parametr klucz, który wskaże element do usunięcia. Kluczem może być oczywiście dowolna wartość, zarówno ciąg znakowy string, jak i obiekt, tak jak w powyższym przypadku. Dodatkowo metoda ta zwraca wartość `boolean` w zależności od tego, czy udało się skutecznie usunąć dany element. Próba usunięcia elementu, który nie istnieje w strukturze `Map`, nie spowoduje zgłoszenia żadnego błędu, lecz po prostu zwróci wartość `false`.

Metoda `clear` natomiast powoduje usunięcie wszystkich elementów struktury `Map`, co wyraźnie pokazuje odczyt właściwości `size`.

Różne sposoby tworzenia struktur Map

Do utworzenia struktury `Map` nie możemy wykorzystać zapisu literalnego z użyciem nawiasów klamrowych lub kwadratowych, które są zarezerwowane w języku JavaScript, odpowiednio, dla standardowych obiektów oraz dla tablic. Musimy więc wykorzystać jawnie konstruktor `Map`. Następnie możemy posłużyć się metodą set, dostępną w prototypie obiektu `Map`.

Metoda ta przyjmuje dwa parametry — klucz oraz wartość (`key`, `value`), jaka zostanie do tego klucza przypisana. Kluczem może być oczywiście dowolna wartość, podobnie jak drugi parametr `value`. Poniżej przedstawiono taki właśnie sposób tworzenia obiektów `Map`:

```
const x = { id: '1' };
const y = { id: '2' };
const z = { id: '3' };

const map = new Map();
map.set(x, 'value 1');
map.set(y, 'value 2');
map.set(z, 'value 3');

Map.size;    // 3
map.get(y); // "value 2"
```

Istnieje jednak również inny sposób. Konstruktor `Map` może przyjąć jako parametr obiekt iterowalny, zwracający listę tablic, w których pierwszy element wskazuje na klucz, a drugi na przypisywaną do niego wartość. Przeanalizujmy zatem taki właśnie sposób tworzenia struktury `Map`:

```
const x = { id: '1' };
const y = { id: '2' };
```

```
const z = { id: '3' };

const map = new Map([
  [x, 'value 1'],
  [y, 'value 2'],
  [z, 'value 3']
]);

Map.size;    // 3
map.get(y); // "value 2"
```

Taka metoda może być przydatna np. w sytuacji, gdy klucze i wartości są tworzone dynamicznie z jakieś innej struktury i nie chcemy wielokrotnie wywoływać metody `set` przez zastosowanie pętli itp. Obiektem iterowalnym jest oczywiście również obiekt `Map`. Wiedza ta pozwala nam np. na proste tworzenie kopii innej mapy:

```
const someObj = { x: 5 };

const map = new Map();
map.set(someObj, 'value 1');

const copyMap = new Map(map);

map.get(someObj);       // "value 1"
copyMap.get(someObj); // "value 1"

map.set(someObj, 'new value'); // zmiana wartości
map.get(someObj);                  // "new value"
copyMap.get(someObj);             // "value 1"
```

Tworząc mapę `copyMap`, do jej konstruktora przekazaliśmy bezpośrednio obiekt `map`. Zauważ, że zmiana wartości przypisanej do klucza `someObj` w mapie pierwszej nie powoduje zmian w mapie drugiej. Taki sposób tworzenia kopii jest możliwy, gdyż mapa przekazana do konstruktora `new Map` zachowa się tak samo jak jawne wywołanie jej metody `entries`:

```
const copyMap2 = new Map(map.entries());
copyMap2.get(someObj); // "new value"

const copyMap3 = new Map(copyMap.entries());
copyMap3.get(someObj); // "value 1"
```

Metoda `entries` zwróci bowiem tablicę, która będzie listą tablic zawierających dwa elementy — klucz oraz skojarzoną z nim wartość. Struktura ta jest analogiczna do list tworzonych za pomocą metody `Object.entries` ze standardowych obiektów javascriptowych.

Modyfikowanie elementów struktury Map

Pracując ze zwykłymi obiektami w języku JavaScript, jesteśmy przyzwyczajeni do łatwego edytowania wartości różnych pól oraz dodawania nowych za pomocą prostej składni. W przypadku obiektów `Map` sytuacja wygląda nieco inaczej: tutaj musimy zawsze po prostu nadpisać wartość dla danego klucza. Przeanalizujmy poniższy przykład:

```
const x = { id: '1' };
const y = { id: '2' };
const z = { id: '3' };

const map = new Map([
  [x, 'value 1'],
  [y, 'value 2'],
  [z, 'value 3']
]);

map.get(x); // "value 1"

map.set(x, 'new value');
map.get(x); // "new value"
```

Początkowo pod kluczem wskazanym przez obiekt x jest przypisana wartość „value 1”. Następnie nadpisujemy ją przez wywołanie metody set, wskazując ponownie ten sam obiekt jako klucz. Operacja ta nie zgłosi żadnego błędu, gdyż mapy w języku JavaScript są obiektami mutowalnymi, więc ich nadpisanie jest jak najbardziej możliwe. Od tej chwili w pamięci przechowywana jest już tylko nowa wartość pod kluczem x.

Przyjrzyjmy się jednak nieco bardziej złożonemu przypadkowi, w którym wartościami skojarzonymi z kluczami są nie proste wartości string, lecz obiekty:

```
const x = { id: '1' };
const y = { id: '2' };
const z = { id: '3' };

const map = new Map([
  [x, { key: 'value 1' }],
  [y, { key: 'value 2' }],
  [z, { key: 'value 3' }],
]);

map.get(x); // {key: "value 1"}

map.set(x, { ...map.get(x), secondKey: 'some value' });
map.get(x); // {key: "value 1", secondKey: "some value"}
```

Tym razem naszym zadaniem jest zmodyfikowanie wartości skojarzonej z kluczem x tak, aby obiekt ten miał dodatkową właściwość secondKey, nie tracąc jednocześnie poprzednich pól. W tym celu wykorzystaliśmy operator destrukturyzacji na wartości zwróconej przez metodę map.get(x), co pozwala nam odtworzyć wszystkie pola z dotychczasowego obiektu, a następnie dodajemy nowy klucz.

Częściej jednak zachodzi inna sytuacja, gdy tworząc mapę, przypisujemy do poszczególnych kluczy referencje do różnych obiektów. W tym momencie edycja takiego obiektu zostanie od razu uwzględniona w wartości skojarzonej z danym kluczem, gdyż klucze wskazują nie na kopię obiektów, lecz na referencje do nich:

```
const obj = { id: '1' };
const map = new Map([['key', obj]]);

map.get('key'); // {id: "1"}
```

```
// Modyfikujemy wartość obiektu, na który wskazuje klucz 'key':
obj.id = 'new-value';
map.get('key'); // {id: "new-value"}
```

Musimy o tym pamiętać, szczególnie jeśli obiekty są przekazywane w wielu miejscach, które mogą je modyfikować. W takiej sytuacji obiekt `Map` będzie zawsze zawierał aktualną wersję obiektu stanowiącego wartość dla danego klucza, czyli stan po ostatnich modyfikacjach obiektu.

Istnieje jednak pewna sytuacja, w której modyfikacja obiektu stanowiącego wartość klucza (referencja do tego obiektu) nie spowoduje zmiany wartości podczas jej pobierania ze struktury `Map`. Mowa tutaj o usuwaniu jakiegoś pola z obiektu. Przeanalizujmy to na prostym przykładzie:

```
const person = {
  name: 'Tomek',
  address: {
    city: 'Warszawa'
  }
};

const map = new Map();
map.set('address', person.address);

map.get('address'); // {city: "Warszawa"}

delete person.address;

map.get('address'); // {city: "Warszawa"}
Person;             // {name: "Tomek"}
```

Widzimy tutaj wyraźnie, że po usunięciu pola `address` nie jest ono już widoczne w obiekcie `person`, jednakże jego dawna wartość nadal jest widoczna w mapie. Musimy uważać na takie zachowanie, jeśli z jakiegoś powodu chcielibyśmy usuwać pola w obiektach, na które wskazują wartości w strukturze `Map`.

Sprawdzanie, czy struktura Map zawiera szukany klucz

W przypadku standardowych obiektów JavaScript, jak również podczas pracy z tablicami często zachodzi konieczność nie tyle iteracji po całej kolekcji, lecz także sprawdzenia, czy zawiera ona jakiś konkretny element, lub wyszukania określonej wartości na podstawie różnych kryteriów. W przypadku tablic mamy do dyspozycji kilka metod, np. `Array.prototype.includes`, `Array.prototype.find`, `Array.prototype.findIndex`.

Struktury `Map` nie mają obecnie takich metod w swoim API, ale być może zostaną one w przyszłości dodane. Istnieje jednak metoda `has`, która sprawdza, czy w mapie istnieje pozycja o wskazanym kluczu. Znamy również oczywiście metodę `get`, która pobiera nam wartość skojarzoną z kluczem. Nie należy jej jednak używać do sprawdzania istnienia pary *klucz-wartość*, gdyż może to powodować wystąpienie tzw. *false negative*:

```
const first = { id: 1 };
const second = { id: 2 };
```

```
const map = new Map([
  [first, 'some text'],
  [second, null]
]);

!!map.get(first);         // true
!!map.get(second);        // false
!!map.get('other key'); // false

map.has(first);           // true
map.has(second);          // true
map.has('other key');     // false
```

W przypadku klucza `first` w obu przypadkach zgodnie z oczekiwaniami dostajemy wartość `true`, co oznacza, że mapa zawiera pod wskazanym kluczem jakąś wartość. Zwróć jednak uwagę na drugi przykład. Tym razem mapa faktycznie zawiera klucz skojarzony z obiektem `second`, lecz przypisana do niego wartość to `null`. Zrzutowanie wartości `null` na `boolean`, jak wiemy z podstaw języka JavaScript, zawsze zwróci wartość `false`. W tym momencie moglibyśmy więc uznać (błędnie), że mapa nie zawiera takiego klucza. Problem ten rozwiązuje wykorzystanie metody `has`, która nie interesuje się wartością skojarzoną z kluczem, lecz sprawdza, czy dany klucz istnieje w strukturze, niezależnie od przypisanej do niego wartości.

W języku JavaScript rzutowanie wartości do `boolean` z wykorzystaniem dwóch wykrzykników jest dość częstą praktyką, jednakże należy zdawać sobie sprawę z tego typu niuansów takiego podejścia. Pracując z mapami, stosuj więc zawsze w takich przypadkach metodę `has`, aby nie narażać się na błędne wyniki, co mogłoby spowodować np. błędy logiczne w działaniu aplikacji.

Wyszukiwanie elementów w strukturze Map

Może zdarzyć się jednak również sytuacja, w której będziemy chcieli wyszukać w strukturze `Map` element na podstawie nie klucza, lecz skojarzonej z nim wartości. Niestety nie dysponujemy jeszcze gotową metodą z API obiektu `Map` do tego celu, ale możemy spróbować samodzielnie napisać taką metodę w prototypie struktury `Map`:

```
const x = { id: '1' };
const y = { id: '2' };

const map = new Map();
map.set(x, 'value 1');
map.set(y, 'value 2');

if (!Map.prototype.find) {
  Object.defineProperty(Map.prototype, 'find', {
    enumerable: false,
    value: function(searched) {
      return [...this.entries()].find(([_, value]) => value === searched);
    }
  });
}

map.find('value 2');      // [ {id: "2"}, "value 2"]
map.find('some value'); // undefined
```

Jeśli zamierzasz w jakikolwiek sposób rozszerzać prototyp obiektów globalnych, zawsze pamiętaj sprawdzić, czy dana metoda nie jest czasem dostępna, aby przypadkowo nie nadpisać jej natywnej implementacji. Nasza funkcja wyszukująca element mapy na podstawie wskazanej wartości wykorzystuje w tym celu mechanizmy dostępne w tablicach. Aby to zrobić, musimy przekonwertować strukturę `Map` na obiekt tablicowy z `Array.prototype`. W naszym przykładzie zastosowaliśmy metodę `entries` z obiektu `Map` (na który wskazuje w tym momencie wartość `this`) w połączeniu z destrukturyzacją. Następnie, dysponując już pełnoprawną tablicą, możemy korzystać z wygodnej metody `Array.prototype.find`. Na koniec zwracamy znalezioną wartość lub `undefined`.

Rozwiązanie to działa w powyższym przykładzie zgodnie z oczekiwaniami, jednakże ma istotną wadę — potrafi weryfikować elementy tylko na podstawie prostych typów wartości lub na podstawie referencji do obiektów, które są w mapie skojarzone z jakimś kluczem:

```
const obj = { id: '1' };
map.set('key', obj);

map.find(obj);           // ["key", { id: '1' }]
map.find({ id: '1' }); // undefined

const arr = [1, 2, 3];
map.set('key_1', arr);

map.find(arr);        // ["key_1", [1, 2, 3]]
map.find([1, 2, 3]); // undefined
```

Jeśli próbujemy wyszukać element przez wskazanie referencji do obiektu, to nasza metoda działa poprawnie. Jednak próba wskazania innego obiektu o takich samych polach zwraca już wartość `undefined`. Jeśli się zastanowimy dokładniej, to zachowanie takie jest poprawne w przypadku powyższej implementacji, gdyż w języku JavaScript dwa obiekty o tych samych polach nie są identyczne i próba ich porównania zwróci `false`:

```
const a = { id: '1' };
const b = { id: '1' };

a === b; // false
{ id: '1' } === { id: '1' } // false
```

Bezpieczniejszym rozwiązaniem będzie więc zaimplementowanie bardziej rozbudowanej funkcji, która pozwoli rzeczywiście weryfikować wskazaną wartość niezależnie od jej typu, zarówno dla wartości prostych (np. `string` czy `number`), jak i dla złożonych struktur (np. obiektów, tablic, map). Spróbujmy wykonać taką implementację:

```
if (!Map.prototype.find) {
  Object.defineProperty(Map.prototype, 'find', {
    enumerable: false,
    value:  function (searched) {
      const isObject = (object) => object != null && typeof object === 'object';
      const areObjects = (val1, val2) => isObject(val1) && isObject(val2);

      function deepEqual(object1, object2) {
        const keys1 = Object.keys(object1);
        const keys2 = Object.keys(object2);
```

```
        if (keys1.length !== keys2.length) {
          return false;
        }

        for (const key of keys1) {
          const val1 = object1[key];
          const val2 = object2[key];
          const areOnlyObjects = areObjects(val1, val2)
          if ((areOnlyObjects && !deepEqual(val1, val2)) || (!areOnlyObjects && val1
          ↪!== val2)) {
            return false;
          }
        }

        return true;
      }

     return [...this.entries()].find(([_, value]) => {
       if (areObjects(value, searched)) {
         return deepEqual(value, searched);
       }
       return value === searched;
     });
    }
  });
}
```

Przeanalizujmy powyższy kod krok po kroku. Najpierw wewnątrz naszej metody `find` tworzymy wewnętrzną funkcję sprawdzającą, czy wskazana wartość jest obiektem javascriptowym:

```
const isObject = (object) => object != null && typeof object === 'object';
```

Zwróć szczególną uwagę na sprawdzenie, czy wskazana wartość nie jest `null`. Nie wystarczy samo sprawdzenie poprzez operator `typeof`, gdyż dla wartości `null` otrzymamy:

```
typeof null; // "object"
```

Takie zachowanie występuje w języku JavaScript od początku jego istnienia i prawdopodobnie nie zostanie nigdy zmienione, tak aby została zachowana zgodność wsteczna istniejącego kodu. Kolejną funkcją pomocniczą jest funkcja, której zadaniem jest sprawdzenie, czy oba jej argumenty są obiektami:

```
const areObjects = (val1, val2) => isObject(val1) && isObject(val2);
```

Za chwilę będziemy porównywać ze sobą kolejne elementy mapy i tylko gdy obie wartości będą obiektami, będzie konieczne wykorzystanie metody `deepEqual`. W przeciwnym razie będziemy pracować na typach prostych, np. `string` czy `number`, i ich porównanie możemy wykonać bezpośrednio operatorem `===`.

Przyjrzyjmy się teraz funkcji `deepEqual`. Na początku tworzymy tablice zawierające klucze z obu obiektów stanowiących argumenty funkcji:

```
const keys1 = Object.keys(object1);
const keys2 = Object.keys(object2);
```

```
if (keys1.length !== keys2.length) {
  return false;
}
```

Gdy obiekty mają różną liczbę kluczy, to jest oczywiste, że nie są identyczne, w związku z czym od razu możemy zwrócić wartość `false`, bez konieczności sprawdzania wartości obiektów. Zastanówmy się w tym miejscu, co dokładnie zwróci nam metoda `Object.keys`. Metodę `deep ↪Equal` będziemy wywoływać tylko w kontekście wartości będących obiektami, a więc sprawdzonymi naszą funkcją `isObject`. W przypadku standardowych obiektów otrzymamy tablicę `Array<string>` zawierającą po prostu klucze obiektu. Struktury `Map` mogą jednak mieć również wartości oraz klucze, które są innymi mapami. Na początek jednak przeanalizujmy, jak nasz kod zachowa się dla dwóch jednakowych obiektów:

```
obj_1 = { x: 1 }
obj_2 = { x: 1 }

Object.keys(obj_1); // ['x']
Object.keys(obj_2); // ['x']

['x'].length === ['x'].length; // true
```

Obiekty mają taką samą liczbę kluczy, w związku z czym musimy zacząć sprawdzać ich wartości. Podobna sytuacja zajdzie, gdy przekażemy dwa różne obiekty mające tę samą liczbę kluczy. Nie jest to jednak istotny problem, gdyż sprawdzenie to jest tylko jednym z kilku walidatorów i służy do wstępnej oceny dwóch obiektów.

Inaczej jednak będzie w przypadku wartości typu `Map`:

```
const map_1 = new Map([['key', 'value']]);
const map_2 = new Map([['key', 'value']]);

Object.keys(map_1); // []
Object.keys(map_2); // []
```

Widzimy więc, że w tym przypadku nasz pierwszy walidator zawsze zwróci `false`, gdyż zawsze będzie porównywał dwie puste tablice. Wykryjemy tu jednak od razu przypadek, gdy jedna wartość będzie mapą, ale druga — zwykłym obiektem, dla którego tablica będzie zawierała listę kluczy. Powyższy kod będzie więc musiał rozpocząć dalszy etap walidacji, polegający na dokładnym porównywaniu wartości dla poszczególnych kluczy. W naszym przypadku nie jest to problemem, gdyż cała funkcja `deepEqual` zadziała poprawnie, jednakże pamiętaj o takim zachowaniu, gdybyś z jakichś powodów chciał kiedyś zastosować sprawdzenie:

```
Array.isArray(map_1); // true
```

Na przykład w celu dalszej iteracji po kluczach takiej mapy, spodziewając się tutaj tablicy z `Array. ↪prototype`. Można sobie poradzić z tym problemem przez zastosowanie metody `Map.proto ↪type.keys`, która zwróci obiekt tablicopodobny `Map Iterator`, zawierający już faktyczne klucze i umożliwiający iterację po nich.

Kolejnym krokiem jest sprawdzenie dokładnych wartości obu elementów:

```
for (const key of keys1) {
  const val1 = object1[key];
  const val2 = object2[key];
  const areOnlyObjects = areObjects(val1, val2)
  if ((areOnlyObjects && !deepEqual(val1, val2)) || (!areOnlyObjects && val1 !== val2)) {
    return false;
  }
}
```

Iterując po kluczach jednego z obiektów, analizujemy, czy nie zachodzi jedna z dwóch sytuacji:

- obie wartości są obiektami, lecz nie są to obiekty o identycznych kluczach i odpowiadających im wartościach (w tym celu rekurencyjnie wywołujemy metodę `deepEqual`)
- lub czy wartości nie są obiektami i jednocześnie czy nie są jednakowe.

W każdej z tych sytuacji kończymy dalsze sprawdzanie i zwracamy wartość `false`. Jeśli w każdej iteracji walidacja zakończy się powodzeniem, cała metoda `deepEqual` zwróci `true`.

Skoro mamy już przygotowane trzy funkcje pomocnicze, czas przejść do właściwego sprawdzenia, czy mapa zawiera klucz o wartości zgodnej ze wskazaną jako parametr `searched`:

```
return [...this.entries()].find(([_, value]) => {
  if (areObjects(value, searched)) {
    return deepEqual(value, searched);
  }
  return value === searched;
});
```

Na początku używamy metody `Map.prototype.entries`, aby utworzyć iterator zawierający rozdzielone ze sobą pary *klucz-wartość*, a następnie z zastosowaniem destrukturyzacji konwertujemy ten iterator na postać tablicy z `Array.prototype`, co pozwala nam wykorzystać wygodną funkcję `Array.prototype.find`.

Metoda ta przyjmuje jako parametr funkcję szukającą, której argumentami są kolejne tablice o dwóch elementach (klucz, wartość). Nas interesuje jedynie wartość, dlatego klucze oznaczyliśmy zmienną „_”. W języku JavaScript jest ogólnie przyjęta praktyka, aby nieużywane parametry oznaczyć właśnie w ten sposób. Moglibyśmy również nie określać nazwy tej zmiennej i zastosować zapis:

```
find(([, value])
```

jednakże wiele osób, w tym ja, uważa, że zapis ten jest mniej czytelny i trudniejszy w zrozumieniu, szczególnie dla osób dopiero zdobywających doświadczenie w języku JavaScript. Nie starajmy się zawsze za wszelką cenę dążyć do skracania kodu i na siłę wykorzystywać wszystkich możliwości języka, gdyż czasami może to prowadzić do niepotrzebnego pogorszenia czytelności kodu i utrudnić jego utrzymanie.

W metodzie `find` dla każdego elementu wykonujemy proste sprawdzenie — jeżeli obie wartości są obiektami, to wywołujemy metodę `deepEqual` (która w przypadku obiektów zagnieżdżonych może zostać wywołana rekurencyjnie), a dla wartości typu prostego dokonujemy zwykłego porównania operatorem `===`.

Ostatecznie metoda `Array.prototype.find` zwraca albo znalezioną wartość, co w naszym przypadku da tablicę dwuelementową [szukany klucz, wartość], albo `undefined`, gdy żaden z elementów nie spełni wskazanych kryteriów.

Gdybyśmy musieli przeszukiwać naprawdę duże struktury `Map` lub porównywać ze sobą obiekty o dużym poziomie zagnieżdżenia, to z pewnością powyższa implementacja wymagałaby poprawy, ale w większości typowych przypadków powinna spełnić nasze oczekiwania.

Skoro omówiliśmy implementację, to spróbujmy przetestować, czy nasza metoda faktycznie działa zgodnie z oczekiwaniami:

```
const map = new Map();
map.set({ id: '1' }, 'value 1');
map.set({ id: '2' }, {someKey: '123'});
map.set({ id: '3' }, ['a', 'b']);

map.find('value 1');         // [{id: "1"}, 'value 1']
map.find({someKey: '123'}); // [{id: "2"}, {someKey: '123'}]
map.find(['a', 'b']);         // [{id: "3"}, ['a', 'b']]

map.find('other-value');      // undefined
```

Jak widzimy, wszystkie elementy zostały poprawnie wyszukane. Działa to zarówno z wartościami prostymi, jak i złożonymi, takimi jak obiekt czy tablica. Co więcej, metoda poprawnie weryfikuje również przypadki, w których wartości są innymi strukturami `Map`:

```
const value = new Map([['key', { id: '1' }]]);
const map = new Map([['someKey', value]]);

map.find(value)[0]; // "someKey"
map.find(value)[1]; // Map(1) {"key" => {…}}
```

Jeśli masz doświadczenie w pracy z językiem JavaScript, to prawdopodobnie jesteś także zaznajomiony z formatem JSON i metodą `JSON.stringify`, która (w uproszczeniu) dokonuje serializacji obiektu do postaci ciągu znakowego. Jednocześnie wycina ona elementy, które nie są zgodne z formatem JSON, w tym np. metody obiektu. W większości sytuacji do pracy ze strukturami zawierającymi wartości, które nie są metodami, np. dane pochodzące z backendu, będziemy jednak wykorzystywać metodę `Map.prototype.find`. Wiele poradników jako jeden ze sposobów porównywania obiektów proponuje wykorzystanie właśnie metody `stringify`. Zalecam jednak całkowite odrzucenie takiego podejścia, gdyż jest ono bardzo podatne na błędy.

Po pierwsze, metoda `stringify` bierze pod uwagę kolejność dodawania pól do obiektu, co sprawia, że dwa obiekty mające te same pola, lecz dodane w różnej kolejności zostaną zserializowane do różnych ciągów znakowych:

```
obj1 = { a:2, b:4 };
obj2 = { b:4, a:2 };

JSON.stringify(obj1); // "{"a":2,"b":4}"
JSON.stringify(obj2); // "{"b":4,"a":2}"

JSON.stringify(obj1) === JSON.stringify(obj2); // false
```

W normalnych sytuacjach kolejność dodawania pól do obiektu raczej nie będzie dla nas miała żadnego znaczenia; będzie nas interesować jedynie istnienie takich samych pól i odpowiadających im tych samych wartości. Zatem z użyciem metody `stringify` możemy przypadkowo odrzucić jako fałsz porównanie obiektów, które wg naszych założeń powinny jednak zostać uznane za identyczne (tzw. zjawisko *false negative*, często bardzo trudne do wykrycia, powodujące czasami błędy dopiero w fazie produkcyjnego działania aplikacji).

Drugą wadą tego rozwiązania jest fakt, że opierając się na porównaniu wartości zwracanych przez `JSON.stringify`, spowodujemy, że każde dwie struktury `Map` uznamy za identyczne. Wynika to z faktu, że serializacja mapy tą metodą zawsze zwraca ciąg:

```
JSON.stringify(new Map([['key', { id: '1' }]]))
"{}"

// a zatem:
const map_1 = new Map([['key-1', { id: '1' }]]);
const map_2 = new Map([['key-2', { id: '5' }]]);
JSON.stringify(map_1) === JSON.stringify(map_2); // true
```

To z kolei powoduje wystąpienie efektu *false positive*, czyli błędnej informacji zwrotnej wskazującej na identyczność obu struktur. Widzimy zatem, że próba wykorzystania metody `JSON.stringify` może spowodować, że spotka nas zarówno przypadek *false negative*, jak i *false positive*. Istnieje ryzyko, że błędy te ujawnią się dopiero w trakcie działania aplikacji. Konia z rzędem dla osoby, która będzie w stanie szybko zlokalizować przyczynę problemu…

Struktury WeakMap

Struktury `WeakMap` charakteryzują się podobnym interfejsem jak typy `Map`, jednakże różni je wewnętrzny sposób przydzielania pamięci. Jeśli obiekt reprezentujący klucz w strukturze `WeakMap` zostanie usunięty w procesie czyszczenia GC (*garbage collector*), to automatycznie usunięty zostanie też cały element `WeakMap`. Z tego względu kluczami w tym przypadku mogą być tylko obiekty — nie stworzymy `WeakMap` z kluczami typu prostego `string`, `number` itp.

W języku JavaScript, aby obiekt został usunięty, muszą zostać usunięte wszystkie referencje wskazujące na niego. Usunięcie pozycji w `WeakMap` zostanie wykonane automatycznie. Nie będziemy mogli jednak tego bezpośrednio sprawdzić. Typ `WeakMap` może być przydatny np. wtedy, gdy pracujemy z obiektami, które mogą być likwidowane dynamicznie, takimi jak referencje do elementów DOM.

Struktury `WeakMap` są podobne do typów `Map`, jednakże udostępniają mniej rozbudowane API. Nie możemy skorzystać z właściwości `size` czy metody `clear`. Struktura `WeakMap` nie udostępnia również iteratorów do pozyskania kluczy i wartości ani całych elementów.

Przeanalizujmy poniższy przykład:

```
let a = { id: 1 };
let b = { id: 2 };
```

```
const map = new WeakMap();
map.set(a, 'some value');
map.set(b, 'other value');

a = null;
```

W momencie gdy do obiektu a przypisujemy wartość null, obiekt zostaje zlikwidowany (zauważ, że na potrzeby testu musieliśmy w deklaracji obiektu użyć słowa kluczowego let zamiast const). Od tej chwili w najbliższym cyklu odzyskiwania pamięci silnik JavaScript może usunąć ze struktury WeakMap element o kluczu wskazującym na obiekt a.

Zobaczmy teraz nieco inny przypadek:

```
let key = { id: 1 };
let value = { id: 2 };

const map = new WeakMap();
map.set(key, value);

value = null;
```

W tym przypadku zarówno klucz, jak i wartość elementu są referencjami do obiektów. Likwidujemy jednak obiekt value, a nie key, jak miało to miejsce w poprzednim przypadku, w którym obiekt a był kluczem w strukturze WeakMap. Tutaj obiekt key nadal istnieje, w związku z tym proces odzyskiwania pamięci nie spowoduje likwidacji tego elementu w mapie. Po prostu referencja do obiektu key jest od tej chwili skojarzona z wartością null.

Zbiory unikatowych elementów Set i WeakSet

Strukturą podobną do typu Map jest tzw. zbiór, czyli typ Set. Jest to kolekcja unikatowych elementów, a dokładniej mówiąc: zbiór elementów o unikatowych kluczach. Interfejs tej struktury jest bardzo podobny do typów Map, z wyjątkiem metody set, która w tym przypadku dostępna jest pod nazwą add.

Zbiory możemy tworzyć przy użyciu metody add, dodając kolejne elementy:

```
const uniqueNumbers = new Set();
uniqueNumbers.add(1);
uniqueNumbers.add(2);
uniqueNumbers.add(2);
uniqueNumbers.add(3);

uniqueNumbers; // {1, 2, 3}
```

Zwróć uwagę, że próba przypisania po raz drugi wartości 2 nie powoduje żadnego błędu, a w ostatecznej strukturze uniqueNumbers cały czas mamy tylko jeden taki element. Prawdopodobnie częściej jednak spotkasz się z tworzeniem zbiorów za pomocą obiektu iterowalnego, np. tablicy czy ciągu znakowego:

```
new Set([1, 2, 2, 3, 3, 4]); // {1, 2, 3, 4}
new Set("abcdabcd");         // {"a", "b", "c", "d"}
```

Obiektem iterowalnym jest również struktura `Map` i ona także może zostać użyta jako parametr konstruktora `Set`. Pozwoli to stworzyć zbiór zawierający tylko elementy o unikatowych kluczach:

```
const map = new Map([
  ['x', 1], ['y', 2], ['x', 3]
]);
[...new Set(map)].forEach(([key, value]) => console.log(`${key}: ${value}`));
// x: 3
// y: 2
```

Zwróć jednak uwagę, jak dokładnie działa struktura `Set`. W obiekcie `Map` mamy dwa elementy pod kluczami „x", ale są one skojarzone z różnymi wartościami. Ostateczny obiekt `Set` zawiera jednak tylko jedną taką parę, na którą wskazuje w obiekcie `Map` ostatni element o kluczu `x`. Konstruktor `Set`, iterując po kolejnych elementach, w razie znalezienia klucza, który już istnieje, nadpisuje jego wartość. Struktury `Set` należy więc rozumieć jako unikatowe zbiory par *klucz-wartość*, lecz unikatowość dotyczy tylko kluczy.

Zbiory `Set` nie mają metody `get`, umożliwiającej pobranie wartości (co jest częstą operacją w przypadku struktur `Map`). W praktyce jednak najczęściej zbiory wykorzystujemy do zapewnienia unikatowości i wystarczy nam metoda `has`, która sprawdza, czy dany element znajduje się w obiekcie `Set`:

```
const numbers = new Set([1, 2, 3]);
numbers.has(2);   // true
numbers.has(10); // false
```

Zbiory mają również metody zwracające iteratory `keys`, `values` oraz `entries`. Domyślnym iteratorem jest `values`, dlatego często spotykaną sytuacją jest wykorzystanie zbiorów do utworzenia tablicy, z której zostaną usunięte duplikaty:

```
const numbers = [1, 2, 2, 3, 3, 4];
const uniqueNumbers = [...new Set(numbers)];
uniqueNumbers; // [1, 2, 3, 4]
```

Stała `uniqueNumbers` jest tablicą z `Array.prototype`, ale z wykorzystaniem konstruktora `Set` utworzyliśmy nową tablicę na podstawie tablicy `numbers`, likwidując wartości, które występowały w niej więcej niż raz. Zastosowaliśmy tutaj destrukturyzację obiektu `Set`, którego domyślnym iteratorem są wartości (`values`).

Obiekty `Set`, podobnie jak `Map`, mają również właściwość `size`, określającą liczbę elementów:

```
const numbers = [1, 2, 2, 3, 3, 4];
numbers.length;        // 6
new Set(numbers).size; // 4
```

Podobnie jak w przypadku struktur `Map` i `WeakMap`, również tutaj dysponujemy typem `WeakSet`. Typ ten w sposób słaby przechowuje wartości, a klucze w zasadzie nie są dla nas istotne.

```
const set = new Set();
let a = { id: 1 };
let b = { id: 2 };
a = null; // obiekt "a" może zostać usunięty w procesie GC
```

Tworzymy własne rozszerzenia struktury Set

Struktura `Set` udostępnia stosunkowo mały zakres metod, ale możemy bez problemu samodzielnie napisać kilka przydatnych implementacji. W takich sytuacjach zawsze należy się zabezpieczyć, na wypadek gdyby w pewnym momencie do prototypu naszego obiektu (w tym przypadku — do `Set.prototype` lub `Array.prototype`) zostały włączone nowe metody o nazwach takich samych jak nazwy naszych implementacji.

Czasami zdarza się, że pracując z tablicami potrzebujemy wyszukać tylko elementy unikalne, a więc usunąć wszelkie duplikaty. Prototyp obiektu `Array` nie udostępnia obecnie takiej metody. W internecie jest wiele różnych przykładów implementacji, np. bazujących na metodzie `includes` czy `indexOf`. My spróbujemy stworzyć takie rozszerzenie `Array.prototype` z wykorzystaniem omówionych już obiektów `Set`:

```
if (!Array.prototype.unique) {
  Object.defineProperty(Array.prototype, 'unique', {
    enumerable: false,
    value:  function () {
      return [...new Set(this)];
    }
  });
}
```

Na początku sprawdzamy, czy przypadkiem prototyp obiektu `Array` nie ma już metody o nazwie `unique`, i tylko w razie jej braku dodajemy własną implementację. Należy jednak zawsze podchodzić ostrożnie do takich działań. Jeśli w przyszłości pojawi się w nowej wersji języka JavaScript metoda `Array.prototype.unique`, to najprawdopodobniej będzie ona zwracać właśnie listę unikatowych elementów z tablicy wejściowej. Można więc uznać, że nasza wersja jest bezpieczna. Zawsze musimy zadać sobie pytanie, jak duże jest ryzyko, że metoda natywna mogłaby mieć inne zachowanie od naszej. Jeśli taka obawa jest realna, to warto własnym metodom nadawać nazwy, które raczej nie wejdą w konflikt z metodami natywnymi (np. moglibyśmy utworzyć metodę `Array.prototype.getOnlyUniqueElements`). Decyzja o tym, jakie podejście wybrać, jest uzależniona m.in. od stopnia skomplikowania implementacji, od różnych przypadków granicznych itp., dlatego zawsze warto przeprowadzić taką analizę przed rozszerzeniem natywnych obiektów JS.

Dodatkowo ustawiamy deskryptor `enumerable` na wartość `false`, co zagwarantuje nam, że nasza dodatkowa metoda nie zostanie uwzględniona w żadnej iteracji pętli itp., co mogłoby spowodować błąd w działaniu aplikacji. Sama implementacja jest generalnie dość prosta: tworzymy nowy obiekt `Set` z tablicy wejściowej, co automatycznie gwarantuje nam likwidację duplikatów. Domyślnym iteratorem obiektów `Set` jest iterowanie po kluczach, dlatego możemy zastosować taki właśnie prosty zabieg z użyciem destrukturyzacji, dzięki czemu zwracany obiekt również jest tablicą. Pora przetestować nową metodę:

```
const numbers = [1, 1, 2, 3, 4, 4, 5, 5];

numbers.unique();                  // [1, 2, 3, 4, 5]
numbers.unique().map(n => n**2);   // [1, 4, 9, 16, 25]
```

Wartość zwrotna również jest obiektem z `Array.prototype`, zatem na tablicy `unique` możemy korzystać z pozostałych metod tablicowych, takich jak `map`, `filter`, `includes`, `reduce`.

Pracując bezpośrednio ze zbiorami `Set`, niekiedy musimy wykonać bardziej zaawansowane analizy. Jednym z przykładów może być sprawdzenie, czy zbiór A jest zawarty w zbiorze B, czyli czy stanowi jego podzbiór. Spróbujmy więc rozszerzyć prototyp obiektu `Set` o metodę `isSubsetOf`, która wywołana w kontekście zbioru A jako parametr, przyjmie zbiór B:

```
if (!Set.prototype.isSubsetOf) {
  Object.defineProperty(Set.prototype, 'isSubsetOf', {
    enumerable: false,
    value:  function (otherSet) {
      if (!(otherSet instanceof Set) || this.size > otherSet.size) {
        return false;
      }

      for (const element of this) {
        if (!otherSet.has(element)) {
          return false;
        }
      }

      return true;
    }
  });
}
```

Na samym początku sprawdzamy, czy na pewno mamy do czynienia ze strukturą `Set`, przy użyciu operatora `instanceof`. Jeżeli tak, to drugim wstępnym warunkiem jest porównanie liczby elementów w obu zbiorach. Jeśliby zbiór A zawierał więcej elementów niż zbiór B, to byłoby logiczne, że nie może on być jego podzbiorem, nie ma więc w takiej sytuacji sensu sprawdzanie kolejnych elementów obu zbiorów.

Jeśli żaden z tych warunków nie jest spełniony, przechodzimy do weryfikacji elementów zbioru A. Wykorzystaliśmy w tym celu pętlę `for-of`, co pozwala nam na wyjście z iteracji w przypadku trafienia na element, który nie znajduje się w zbiorze B (co jednocześnie oznacza, że nie każdy element zbioru A jest zawarty w zbiorze B, zatem jako całość zbiór A nie może być podzbiorem zbioru B).

Alternatywnie moglibyśmy wykorzystać konwersję zbioru na tablicę z użyciem metody `Array.prototype.every` w celu iteracji po elementach zbioru A wraz z metodą `Array.prototype.includes` w celu sprawdzenia obecności elementu w zbiorze B. Implementacja taka wiązałaby się jednak z niepotrzebnym nakładem czasu na konwersję obiektu `Set` na obiekt `Array`.

Jeśli każdy element zbioru A zawiera się w zbiorze B, to po wyjściu z pętli `for-of` zwracamy po prostu wartość `true`.

Cofnijmy się jednak do miejsca, gdzie sprawdzamy, czy argument `otherSet` faktycznie jest obiektem typu `Set`. Używając w tym celu operatora `instanceof`, narażamy się na ryzyko otrzymania tzw. *false positive* dla wyrażenia:

```
new (class MySet extends Set {}) instanceof Set; // true
```

Jeśli tworzylibyśmy bibliotekę udostępnioną dla świata zewnętrznego, to powinniśmy przewidywać różne tego typu przypadki graniczne i odpowiednio się przed nimi zabezpieczyć. Tworząc jednak tego typu implementację na potrzeby tylko własnej aplikacji, możemy czasami przyjąć nieco większe uproszczenia. Możemy bowiem przyjąć, że mamy do pewnego stopnia kontrolę nad tym, w jaki sposób użyjemy tej metody. Warto jednak przeanalizować tego typu przypadki i zobaczyć, jak własna implementacja się zachowa w takich sytuacjach.

Zweryfikujmy teraz, czy powyższa implementacja spełnia nasze oczekiwania:

```
const set_A = new Set([1, 2, 3, 4, 5]);
const set_B = new Set([2, 3]);
const set_C = new Set([0, 1, 2]);
const set_D = new Set([1, 2, 3, 4, 5, 6]);

set_B.isSubsetOf(set_A); // true
set_C.isSubsetOf(set_A); // false
set_D.isSubsetOf(set_A); // false
set_B.isSubsetOf('invalid Set type'); // false
```

Kolejnym przykładem rozszerzania prototypu obiektu `Set` może być utworzenie metody zwracającej tzw. przecięcie dwóch zbiorów, czyli wyznaczającej ich część wspólną. Ćwiczenie to pozwoli Ci utrwalić wiedzę z zakresu pracy ze zbiorami i ich API:

```
if (!Set.prototype.intersectionWith) {
  Object.defineProperty(Set.prototype, 'intersectionWith', {
    enumerable: false,
    value:  function (otherSet) {
      if (!(otherSet instanceof Set)) {
        return false;
      }

      const intersection = new Set();

      for (const element of otherSet) {
        if (this.has(element)) {
          intersection.add(element);
        }
      }

      return intersection;
    }
  });
}
```

Na początku tworzymy nowy obiekt `Set`. Następnie, iterując po zbiorze `otherSet`, sprawdzamy, czy zbiór wejściowy (`this`) zawiera dany element ze zbioru przekazanego jako parametr metody `intersectionWith`. Ostatnim etapem jest zwrócenie zbioru zawierającego wspólne elementy lub pustego obiektu `Set`. Takie zachowanie gwarantuje nam, że zawsze wartością zwrotną będzie obiekt `Set`, co pozwoli bezpiecznie wykonać na nim inne metody `Set.prototype`.

```
const set_A = new Set([1, 2, 3, 4, 5]);
const set_B = new Set([2, 3]);
const set_C = new Set([0, 1, 2]);
const set_D = new Set([1, 2, 3, 4, 5, 6]);
```

```
const set_E = new Set([10, 11, 12]);

set_B.intersectionWith(set_A); // Set(2) {2, 3}
set_C.intersectionWith(set_A); // Set(2) {1, 2}
set_D.intersectionWith(set_A); // Set(5) {1, 2, 3, 4, 5}
set_E.intersectionWith(set_A); // Set(0) {}
```

Aby utrwalić sobie wiedzę z zakresu obiektów `Map` oraz `Set`, stwórz sobie kolejne przykładowe rozszerzenia domyślnych prototypów tych obiektów. To pozwoli Ci dokładnie zrozumieć, jak wygląda praca z takimi strukturami.

Podsumowanie

W tym rozdziale poznałeś dwa typy danych: `Map` oraz `Set`, które w niektórych sytuacjach mogą być zamiennikami dla najczęściej używanych standardowych obiektów czy tablic. Nie oznacza to oczywiście, że teraz masz całkowicie zrezygnować ze stosowania obiektów na rzecz np. struktury `Map`. Zawsze warto dokładnie przeanalizować cel, w jakim tworzy się dany obiekt z danymi, i to, w jaki sposób chce się z nim pracować.

Ponadto szczególnie obiekty `Set` mogą być strukturą pomocniczą, ułatwiającą niektóre zadania, chociażby wykorzystanie ich do prostego utworzenia tablicy pozbawionej duplikatów.

JAVASCRIPT

TECHNIKI ZAAWANSOWANE

JavaScript rozwija się wraz z internetem. Początkowo był używany do pisania kodu prostych interakcji na stronach WWW, dziś pozwala budować pełnoprawne programy umożliwiające dynamiczną wymianę danych z serwerami, obsługę urządzeń peryferyjnych komputera, jak kamera, mikrofon, różnego rodzaju czujniki itp. Ponadto język ten może być stosowany do tworzenia wydajnych aplikacji serwerowych, a także do programowania urządzeń takich jak mikroroboty czy silniki elektryczne.

Jeśli znasz podstawy tego języka, jeśli zdarzyło Ci się już coś w nim napisać, ten podręcznik jest właśnie dla Ciebie. Pozwoli Ci uzyskać wiedzę i umiejętności, dzięki którym dołączysz do grona programistów tworzących w JavaScript oprogramowanie dostępne praktycznie na każdą platformę — od części serwerowej, przez przeglądarki internetowe i aplikacje natywne dla smartfonów, po takie urządzenia jak smartwatch, smart TV i wiele innych.

Skoro potrafisz tworzyć proste aplikacje, swobodnie poruszasz się w składni języka JavaScript i wśród typów zmiennych, rozumiesz i umiesz korzystać z asynchroniczności, to czas najwyższy na:

- **upraszczanie kodu aplikacji i korzystanie z wbudowanych mechanizmów języka**
- **przyjrzenie się iteratorom i generatorom**
- **wykorzystanie możliwości, jakie w JavaScript daje asynchroniczność**
- **pracę wielowątkową**
- **wyjście poza proste zbiory danych**
- **programowanie reaktywne**

Narzędzie dla webdevelopera

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

ISBN 978-83-283-5640-5

Cena: 54,90 zł