O'REILLY®

NOWOCZESNY JAVASCRIPT

POZNAJ ES6 I PRAKTYCZNE ZASTOSOWANIA NOWYCH ROZWIĄZAŃ

NICOLÁS BEVACQUA

Helion

Tytuł oryginału: Practical Modern JavaScript: Dive into ES6 and the Future of JavaScript

Tłumaczenie: Inez Okulska-Stanisławska

ISBN: 978-83-283-4229-3

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

Printed in Poland.

Spis treści

ROZDZIAŁ 2.

Najistotniejsze elementy ES6

Szósta edycja języka oferuje szeroki wachlarz ulepszeń składni, które nie wprowadzają zmian łamiących kompatybilność wsteczną i które omówię w tym rozdziale. Wiele z nich to po prostu lukier składniowy, czyli możliwości, które mogły pojawić się już w ES5, ale przy użyciu bardziej skomplikowanych fragmentów kodu. Są też jednak zmiany, które nie są jedynie ozdobnikami, lecz oferują całkowicie nowy sposób deklaracji zmiennych przy użyciu `let` oraz `const`, o czym przekonasz się pod koniec tego rozdziału.

W literałach obiektu w ES6 również wprowadzono parę zmian i od nich zacznę.

2.1. Literały obiektu

Literałem obiektu jest każda deklaracja obiektu używająca zapisu klamrowego {}, tak jak w poniższym przykładzie.

```
var book = {
   title: 'Modular ES6',
   author: 'Nicolas',
   publisher: 'O´Reilly'
}
```

ES6 wnosi do składni literałów obiektu kilka ulepszeń, takich jak skrócone definiowanie właściwości, generowane nazwy właściwości i definicje metod. Przyjrzyjmy się bliżej tym możliwościom i potencjalnym przykładom ich zastosowania.

2.1.1. Zwięzła deklaracja właściwości

Czasem deklarujemy obiekty z jedną właściwością lub wieloma właściwościami; ich wartościami są referencje do zmiennych o tej samej nazwie. Możemy np. mieć kolekcję `listeners` i w celu przypisania do właściwości nazwanej `listeners` literału obiektu musimy powtórzyć tę nazwę. Poniższy fragment kodu pokazuje typowy przykład, gdzie mamy deklarację literału obiektu z kilkoma takimi powtarzającymi się właściwościami.

```
var listeners = []
function listen() {}
var events = {
   listeners: listeners,
   listen: listen
}
```

Kiedykolwiek znajdziesz się w takiej sytuacji, możesz ominąć przypisanie wartości oraz średnik i skorzystać z nowego zapisu z ES6. Jak pokazuje poniższy przykład, nowa zwięzła składnia ES6 implikuje przypisanie:

```
var listeners = []
function listen() {}
var events = { listeners, listen }
```

Później, w drugiej części książki dowiesz się, że zapis klamrowy przy przypisaniu wartości właściwości pomaga oczyścić kod z niepotrzebnych powtórzeń, bez ingerencji w jego działanie. W poniższym przykładzie ponownie zaimplementowano część `localStorage`, API przeglądarki do trwałego przechowywania danych jako wewnątrzpamięciowy ponyfill[1]. Gdyby nie skrócona składnia, obiekt `storage` byłby o wiele bardziej rozwlekły w zapisie.

```
var store = {}
var storage = { getItem, setItem, clear }
function getItem(key) {
   return key in store ? store[key] : null
}
function setItem(key, value) {
   store[key] = value
}
function clear() {
   store = {}
}
```

To pierwsza z wielu możliwości ES6, które mają na celu uproszczenie zawiłości w kodzie, dzięki czemu łatwiej nad nim zapanować. Kiedy już przyzwyczaisz się do tej składni, zauważysz, że poprawią się zarówno czytelność, jak i produktywność programowania.

2.1.2. Generowane nazwy właściwości

Czasami musisz zadeklarować obiekt, który zawiera właściwości o nazwach bazujących na zmiennych czy innych wyrażeniach JavaScriptu, co pokazuje poniższy fragment napisany w ES5. Załóżmy np., że `expertise` otrzymujesz jako parametr funkcji i nie jest to wartość, którą znałeś już wcześniej.

```
var expertise = 'journalism'
var person = {
   name: 'Sharon',
   age: 27
}
person[expertise] = {
   years: 5,
   interests: ['international', 'politics', 'internet']
}
```

[1] Podobnie jak polyfille, ponyfille (*https://mjavascript.com/out/ponyfills*) to implementacje po stronie użytkownika funkcji, które nie są dostępne w każdym środowisku uruchomieniowym JavaScriptu. O ile polyfille próbują wypełniać lukę w danym środowisku, żeby mogło działać, tak jakby rzeczywiście miało dostępną daną funkcję, o tyle ponyfille implementują brakujące funkcje jako osobne moduły, które nie zaśmiecą środowiska uruchomieniowego. Ma to taką zaletę, że nie koliduje z oczekiwaniami zewnętrznych, niezależnych od naszego kodu bibliotek (które nie wiedzą o polyfillu), które mogą być używane w danym środowisku.

Literały obiektu w ES6 nie są ograniczone do deklaracji za pomocą statycznych obiektów. Za pomocą generowanych nazw właściwości możesz dowolne wyrażenie zapisać w nawiasie kwadratowym i użyć go jako nazwy właściwości. I kiedy parser dojdzie do miejsca deklaracji, Twoje wyrażenie zostanie wykonane i użyte jako nazwa właściwości. Poniższy przykład pokazuje, jak te części kodu, które już widzieliśmy, za jednym zamachem deklarują obiekt `person`, bez konieczności użycia drugiej instrukcji dodającej właściwość `expertise` do obiektu `person`.

```
var expertise = 'journalism'
var person = {
   name: 'Sharon',
   age: 27,
   [expertise]: {
      years: 5,
      interests: ['international', 'politics', 'internet']
   }
}
```

Nie możesz łączyć zwięzłej składni z generowanymi nazwami właściwości. Zwięzły zapis to po prostu lukier składniowy, który jest wykorzystywany w czasie kompilacji kodu JavaScript i pomaga uniknąć powtórzeń, podczas gdy generowane nazwy właściwości są wykonywane w czasie rzeczywistym. Biorąc pod uwagę fakt, że chcemy połączyć te dwie niekompatybilne ze sobą funkcje, poniższy przykład wyrzuciłby błąd. W większości przypadków taka kombinacja prowadziłaby do powstania kodu, który byłby trudny do zinterpretowania dla innych programistów, więc chyba lepiej dla Ciebie, że nie da się tego w ten sposób użyć.

```
var expertise = 'journalism'
var journalism = {
   years: 5,
   interests: ['international', 'politics', 'internet']
}
var person = {
   name: 'Sharon',
   age: 27,
   [expertise]
// błąd składni!
}
```

Typowym scenariuszem użycia generowanej nazwy właściwości jest chęć dodania encji do obiektu mapującego, który używa pola `entity.id` jako klucza, co widać w poniższym przykładzie. Zamiast używać trzeciej instrukcji, gdzie dodalibyśmy `grocery` do mapy `groceries`, możemy wpisać tę deklarację do literału obiektu `groceries`.

```
var grocery = {
   id: 'bananas',
   name: 'Bananas',
   units: 6,
   price: 10,
   currency: 'USD'
}
var groceries = {
   [grocery.id]: grocery
}
```

Może też się okazać, że za każdym razem, gdy funkcja otrzymuje parametr, powinniśmy użyć go do zbudowania obiektu. W kodzie ES5 trzeba by alokować zmienną deklarującą literał obiektu,

dodać dynamiczną właściwość, a na końcu zwrócić obiekt. Poniższy przykład pokazuje dokładnie taki przypadek podczas tworzenia koperty, która mogłaby zostać później użyta do wiadomości ajaksowych, zgodnych z pewną konwencją — posiadają one właściwość `error` z opisem, jeśli coś się nie udaje, i właściwość `success`, gdy wszystko się uda.

```
function getEnvelope(type, description) {
   var envelope = {
      data: {}
   }
   envelope[type] = description
   return envelope
}
```

Generowane nazwy właściwości pomagają zapisać tę samą funkcję w sposób bardziej skondensowany, przy użyciu jednej instrukcji.

```
function getEnvelope(type, description) {
   return {
      data: {},
      [type]: description
   }
}
```

Ostatnie ulepszenie dotyczące literałów obiektu odnosi się do funkcji.

2.1.3. Definiowanie metody

Zazwyczaj możesz deklarować metodę obiektu jako właściwość obiektu. W następnym przykładzie utworzymy mały emiter zdarzeń, który obsługuje różne rodzaje zdarzeń. Robi to za pomocą metody `emitter#on`, której można użyć do rejestracji funkcji obsługi zdarzenia. Natomiast metoda `emitter#emit` służy do wywołania konkretnego zdarzenia.

```
var emitter = {
   events: {},
   on: function (type, fn) {
      if (this.events[type] === undefined) {
         this.events[type] = []
      }
      this.events[type].push(fn)
   },
   emit: function (type, event) {
      if (this.events[type] === undefined) {
         return
      }
      this.events[type].forEach(function (fn) {
         fn(event)
      })
   }
}
```

Od wersji ES6 możesz deklarować metody w literale obiektu, używając nowej składni definiowania metod. W tym przypadku możemy zrezygnować ze średnika i słowa kluczowego `function`. To nowa zwięzła alternatywa dla tradycyjnej deklaracji metody, w której trzeba było właśnie wstawiać słowo `function`. Poniższy przykład pokazuje, jak będzie wyglądał nasz obiekt `emitter`, jeśli skorzystamy z definicji metody.

```
var emitter = {
   events: {},
   on(type, fn) {
      if (this.events[type] === undefined) {
         this.events[type] = []
      }
      this.events[type].push(fn)
   },
   emit(type, event) {
      if (this.events[type] === undefined) {
         return
      }
      this.events[type].forEach(function (fn) {
         fn(event)
      })
   }
}
```

Jeszcze innym sposobem deklaracji funkcji w ES6 są funkcje strzałki; mamy do dyspozycji całą ich paletę. Przyjrzyjmy się bliżej, czym są te funkcje, jak się je deklaruje i jak się zachowują pod kątem semantyki.

2.2. Funkcje strzałki

W JavaScripcie zazwyczaj deklaruje się funkcje przy użyciu kodu, gdzie — tak jak w poniższym przykładzie — podajemy nazwę, listę parametrów oraz ciało funkcji.

```
function name(parameters) {
   // ciało funkcji
}
```

Możesz też utworzyć funkcję anonimową, omijając jej nazwę podczas przypisania zmiennej lub właściwości do funkcji oraz podczas jej wywołania.

```
var example = function (parameters) {
   // ciało funkcji
}
```

Od wersji ES6 możesz już korzystać z funkcji strzałki jako innego sposobu zapisu anonimowej funkcji. Pamiętaj, że jest wiele nieco różniących się od siebie wersji tego zapisu. Poniższy fragment kodu pokazuje funkcję strzałki, która jest bardzo podobna do funkcji anonimowej, zademonstrowanej przed chwilą. Jedyną różnicą zdaje się być brakujące słowo klucz `function` oraz obecność strzałki `=>` na prawo od listy parametrów.

```
var example = (parameters) => {
   // ciało funkcji
}
```

Chociaż funkcje strzałki wyglądają bardzo podobnie do typowych anonimowych funkcji, w rzeczywistości są od nich diametralnie różne — funkcji strzałki nie można bezpośrednio nazwać, chociaż nowoczesne środowiska uruchomieniowe mogą wywnioskować jej nazwę ze zmiennej, do której taka funkcja jest przypisana; nie mogą być też użyte jako konstruktory, ani nie posiadają właściwości `prototype`, co oznacza, że nie możesz użyć słowa kluczowego `new` w połączeniu z funkcją strzałki. Funkcje te są też ograniczone zasięgiem leksykalnym i dlatego nie modyfikują znaczenia `this`.

Zagłębmy się zatem w semantycznych różnicach względem funkcji tradycyjnych, dostępnych sposobach deklaracji funkcji strzałki oraz praktycznych przykładach ich zastosowania.

2.2.1. Zasięg leksykalny

W ciele funkcji strzałki `this`, `arguments` oraz `super` odnoszą się do już istniejącego zasięgu, ponieważ funkcja strzałki nie tworzy nowego zasięgu. Rozważ poniższy przykład — mamy tu obiekt `timer` z licznikiem `seconds` oraz metodą `start`, zdefiniowaną przy użyciu składni, o której była mowa wcześniej. Uruchamiamy `timer`, czekamy parę sekund i zgłaszamy aktualną ilość czasu (`seconds`), która upłynęła.

```
var timer = {
   seconds: 0,
   start() {
      setInterval(() => {
         this.seconds++
      }, 1000)
   }
}
timer.start()
setTimeout(function () {
   console.log(timer.seconds)
}, 3500)
// <- 3
```

Gdybyśmy zdefiniowali funkcję przekazywaną do `setInterval` jako zwykłą funkcję anonimową zamiast używać funkcji strzałki, wówczas `this` odnosiłoby się do kontekstu tej anonimowej funkcji, a nie — tak jak tu — do kontekstu metody `start`. Moglibyśmy zaimplementować `timer`, używając deklaracji typu `var self = this` na początku metody `start`, i potem referencji do `self` zamiast `this`. W przypadku funkcji strzałki zanika dodatkowa komplikacja związana z utrzymaniem referencji kontekstowych, dzięki czemu możemy się skupić na funkcjonalności kodu.

Analogicznie leksykalne powiązania w funkcjach strzałki w ES6 oznaczają również, że wywołania funkcji nie mogą modyfikować kontekstu `this` poprzez użycie `.call`, `.apply`, `.bind` itp. Ograniczenie to zazwyczaj jest przydatne, bo gwarantuje, że kontekst zawsze będzie zachowany i niezmienny.

Przejdźmy teraz do poniższego przykładu. Jak myślisz, co wypisze `console.log`?

```
function puzzle() {
   return function () {
      console.log(arguments)
   }
}
puzzle('a', 'b', 'c')(1, 2, 3)
```

Odpowiedź brzmi następująco: `arguments` odnoszą się tutaj do kontekstu funkcji anonimowej i dlatego wypisane zostaną argumenty przekazane do tej funkcji. A w tym przypadku są to `1, 2, 3`.

A co z kolei się stanie, gdy w poprzednim przykładzie użyjemy funkcji strzałki zamiast funkcji anonimowej?

```
function puzzle() {
   return () => console.log(arguments)
}
puzzle('a', 'b', 'c')(1, 2, 3)
```

W tym przypadku obiekt `arguments` odnosi się do kontekstu funkcji `puzzle`, ponieważ funkcje strzałki nie tworzą domknięcia. Dlatego też wypisane zostaną argumenty `'a'`, `'b'`, `'c'`.

Jak wspominałem, istnieje sporo odmian funkcji strzałki, ale dotąd przyjrzeliśmy się tylko ich pełnej wersji. Jakie są inne sposoby reprezentacji funkcji strzałki?

2.2.2. Odmiany funkcji strzałki

Spójrzmy raz jeszcze na tę składnię funkcji strzałki, którą zdążyliśmy poznać do tej pory.

```
var example = (parameters) => {
   // ciało funkcji
}
```

Jeśli funkcja strzałki posiada dokładnie jeden parametr, wówczas można pominąć opcjonalny nawias. Jest to przydatne, jeśli przekazujemy funkcję strzałki do innej metody, bo wtedy redukujemy liczbę nawiasów, co sprawia, że kod staje się bardziej czytelny dla ludzkiego oka.

```
var double = value => {
   return value * 2
}
```

Funkcje strzałki są często używane do prostych funkcji, takich jak funkcja `double`, którą już omawiałem. Poniższa odmiana funkcji strzałki pozbywa się ciała funkcji. Zamiast niego wystarczy wyrażenie, takie jak `value * 2`. Kiedy funkcja zostanie wywołana, wykonywane jest wyrażenie i zwracany wynik. Instrukcja `return` jest wywoływana domyślnie — nie trzeba też używać nawiasów oznaczających ciało funkcji, dlatego wystarczy pojedyncze wyrażenie.

```
var double = (value) => value * 2
```

Zauważ, że możesz połączyć ukryty nawias i domyślny `return`, tworząc jeszcze bardziej zwięzłą funkcję strzałkową.

```
var double = value => value * 2
```

Domyślne zwracanie literałów obiektu

Jeśli chcesz zastosować domyślną instrukcję `return` dla literałów obiektu, musisz umieścić ją w nawiasie. W przeciwnym razie kompilator zinterpretuje Twoje nawiasy klamrowe jako początek i koniec bloku funkcji.

```
var objectFactory = () => ({ modular: 'es6' })
```

W poniższym przykładzie JavaScript interpretuje nawiasy klamrowe jako ciało naszej funkcji strzałki. Co więcej, `number` jest interpretowany jako etykieta[2], co sprawia, że mamy wyrażenie `value`, które niczego nie robi. A ponieważ znajdujemy się w bloku i nie zwracamy niczego, zmapowane wartości będą `undefined`.

```
[1, 2, 3].map(value => { number: value })
// <- [undefined, undefined, undefined]
```

[2] Etykiety (*https://mjavascript.com/out/label*) są stosowane w celu identyfikacji instrukcji. Etykiet można używać wraz z instrukcją `break`, żeby wskazać, z której sekwencji chcemy wyjść, oraz z instrukcją `continue`, żeby było wiadomo, którą sekwencję chcemy kontynuować.

Jeśli nasza próba domyślnego zwracania literału obiektu zakładałaby więcej niż jedną właściwość, wówczas kompilator nie byłby w stanie poprawnie zinterpretować drugiej właściwości, co z kolei poskutkowałoby wyrzuceniem `SyntaxError`.

```
[1, 2, 3].map(value => { number: value, verified: true })
// <- SyntaxError
```

Wstawienie wyrażenia w nawias rozwiązuje ten problem, ponieważ kompilator przestanie interpretować je jako blok funkcji. Zamiast tego deklaracja obiektu stanie się wyrażeniem odnoszącym się do literału obiektu, który chcemy domyślnie zwrócić.

```
[1, 2, 3].map(value => ({ number: value, verified: true }))
/* <- [
   { number: 1, verified: true },
   { number: 2, verified: true },
   { number: 3, verified: true }]
*/
```

Teraz, kiedy już rozumiesz funkcje strzałki, możemy pomyśleć o ich zaletach i sytuacjach, w których mogą się przydać.

2.2.3. Zalety i przykłady zastosowania

Z zasady nie warto bezrefleksyjnie wykorzystywać możliwości ES6, kiedy tylko się da. Lepiej pomyśleć o każdym przypadku indywidualnie i zastanowić się, czy wdrożenie nowej cechy rzeczywiście polepszy czytelność i stabilność kodu. Możliwości z ES6 nie zawsze są lepsze od tego, czym dysponowaliśmy do tej pory, i traktowanie ich w ten sposób nie jest najlepszym pomysłem.

Jest kilka przypadków, w których użycie funkcji strzałki może nie mieć większego sensu. Jeśli np. masz pokaźną funkcję składającą się z wielu linii kodu, zamiana `function` na `=>` raczej niespecjalnie polepszy Twój kod. Funkcje strzałki są najskuteczniejsze dla krótkich instrukcji, gdzie słowo kluczowe `function` i idąca za nim składnia stanowią istotną część wyrażenia funkcji.

Odpowiednie nazwanie funkcji dodaje kontekst, co sprawia, że człowiek może łatwiej zinterpretować taki kod. Funkcje strzałki nie mogą być bezpośrednio nazywane, ale można je nazwać pośrednio poprzez przypisanie ich do zmiennej. W poniższym przykładzie przypisujemy funkcję strzałki do zmiennej `throwError`. Kiedy wywołanie tej funkcji poskutkuje błędem, zrzut stosu poprawnie zidentyfikuje ją jako `throwError`.

```
var throwError = message => {
   throw new Error(message)
}
throwError('this is a warning')
<- Uncaught Error: this is a warning at throwError
```

Funkcje strzałki są zgrabne, jeśli trzeba definiować anonimowe funkcje, które prawdopodobnie tak czy inaczej powinny być leksykalnie związane. Funkcje te w niektórych sytuacjach sprawiają też, że Twój kod staje się bardziej zwięzły. Są szczególnie przydatne w większości przypadków programowania funkcjonalnego, np. przy korzystaniu z `.map`, `.filter` czy `.reduce` w kolekcjach, co pokazuje poniższy przykład.

```
[1, 2, 3, 4]
   .map(value => value * 2)
   .filter(value => value > 2)
```

```
   .forEach(value => console.log(value))
// <- 4
// <- 6
// <- 8
```

2.3. Destrukturyzacja przypisania

To jedna z najbardziej elastycznych i wyrazistych możliwości w ES6. A także jedna z najprostszych. Wiąże właściwości z tyloma zmiennymi, ilu tylko sobie zażyczysz. Działa dla obiektów, tablic, a nawet list parametrów `function`. Przyjrzyjmy się jej dokładnie, zaczynając od obiektów.

2.3.1. Destrukturyzacja obiektów

Wyobraź sobie, że masz program, w którym występują postaci z komiksów, a wśród nich Bruce Wayne, i chcesz zrobić odniesienie do właściwości obiektu, który go opisuje. Oto przykład obiektu, którego użyjemy dla Batmana.

```
var character = {
   name: 'Bruce',
   pseudonym: 'Batman',
   metadata: {
      age: 34,
      gender: 'male'
   },
   batarang: ['gas pellet', 'bat-mobile control', 'bat-cuffs']
}
```

Gdybyś chciał, żeby zmienna `pseudonym` odnosiła się do `character.pseudonym`, mógłbyś tak to zapisać w kodzie ES5. To typowe rozwiązanie, jeśli chcesz np. odnosić się do `pseudonym` w różnych miejscach swojego kodu i wolałbyś uniknąć wpisywania za każdym razem `character.pseudonym`.

```
var pseudonym = character.pseudonym
```

Z wykorzystaniem destrukturyzacji w przypisaniu składnia staje się nieco bardziej przejrzysta. Jak widać w poniższym przykładzie, nie musisz dwukrotnie pisać `pseudonym`, a i tak intencja zostanie poprawnie przekazana. Poniższa instrukcja jest równoważna z poprzednią napisaną w kodzie ES5.

```
var { pseudonym } = character
```

Tak jak możesz zadeklarować wiele zmiennych po przecinku, używając prostej instrukcji `var`, tak możesz deklarować wiele zmiennych wewnątrz nawiasu klamrowego wyrażenia destrukturyzowanego.

```
var { pseudonym, name } = character
```

W podobnym stylu możesz mieszać destrukturyzację z normalnym zapisem deklaracji zmiennych wewnątrz tej samej instrukcji `var`. Chociaż na pierwszy rzut oka może to nieco mylić, jednak od wybranego przez Ciebie stylu kodowania będzie zależało, czy należy deklarować różne zmienne za pomocą jednej instrukcji. W każdym razie pokazuje to elastyczność, jaką oferuje składnia destrukturyzacji.

```
var { pseudonym } = character, two = 2
```

Jeśli chcesz wyodrębnić właściwość nazwaną `pseudonym`, ale jednocześnie zadeklarować ją jako zmienną pod nazwą `alias`, możesz w tym celu użyć następującej składni destrukturyzującej, zwanej *aliasowaniem*. Zauważ, że możesz wykorzystać `alias` albo jakąkolwiek inną dostępną nazwę zmiennej.

```
var { pseudonym: alias } = character
console.log(alias)
// <- 'Batman'
```

Chociaż aliasy nie wyglądają prościej od zapisu z wersji ES5, `alias = character.pseudonym`, zaczynają mieć sens wtedy, gdy weźmiesz pod uwagę fakt, że destrukturyzacja służy głębokim strukturom, tak jak w poniższym przykładzie.

```
var { metadata: { gender } } = character
```

W takich przypadkach jak poprzedni, gdzie mieliśmy głęboko zagnieżdżoną właściwość, która była poddawana destrukturyzacji, mógłbyś za pomocą aliasów pobrać właściwość w jeszcze bardziej przejrzysty sposób. Przyjrzyj się poniższemu fragmentowi, gdzie właściwość nazwana `code` nie wskazywałaby na swoją zawartość tak jasno jak właściwość `colorCode`.

```
var { metadata: { gender: characterGender } } = character
```

Scenariusz, który przed chwilą przedstawiłem, występuje bardzo często, ponieważ właściwości są często nazywane w kontekście swoich nadrzędnych obiektów. Chociaż `palette.color.code` jest idealnie opisowa, `code` sama w sobie mogłaby już mieć wiele znaczeń, ale — na szczęście — aliasy, takie jak `colorCode`, pomagają ponownie przywrócić brakujący kontekst do nazwy zmiennej, bez konieczności rezygnowania z destrukturyzacji.

Za każdym razem, gdy w notacji ES5 próbujesz uzyskać dostęp do nieistniejącej właściwości, otrzymujesz wartość `undefined`.

```
console.log(character.boots)
// <- undefined
console.log(character['boots'])
// <- undefined
```

Przy użyciu destrukturyzacji zachowanie to nie ulega zmianie. Deklarując destrukturyzowaną zmienną dla właściwości, której brakuje, również otrzymasz wartość `undefined`.

```
var { boots } = character
console.log(boots)
// <- undefined
```

Destrukturyzowana deklaracja sięgająca do zagnieżdżonej właściwości obiektu nadrzędnego, który ma wartość `null` lub `undefined`, wyrzuci wyjątek, tak jak przy zwykłej próbie dostępu do właściwości `null` czy `undefined`.

```
var { boots: { size } } = character
// <- wyjątek
var { missing } = null
// <- wyjątek
```

Jeśli pomyślisz o tym fragmencie kodu jak o odpowiedniku kodu napisanego w ES5, który pokażę za chwilę, stanie się jasne, że wyrażenie musi wyrzucić wyjątek, zwłaszcza wtedy, gdy weźmiemy pod uwagę fakt, że destrukturyzacja jest w zasadzie lukrem składniowym.

```
var nothing = null
var missing = nothing.missing
// <- wyjątek
```

W ramach częściowej destrukturyzacji możesz ustawić domyślne wartości za każdym razem, kiedy wartość byłaby `undefined`. Wartość domyślna może być czymkolwiek, co przyjdzie Ci do głowy — mogą to być liczby, ciągi tekstowe, funkcje, obiekty, referencja do innej zmiennej itp.

```
var { boots = { size: 10 } } = character
console.log(boots)
// <- { size: 10 }
```

Domyślne wartości można też ustawić dla destrukturyzacji zagnieżdżonych właściwości.

```
var { metadata: { enemy = 'Satan' } } = character
console.log(enemy)
// <- 'Satan'
```

Jeśli chcesz połączyć je z aliasem, najpierw powinieneś wpisać alias, a potem wartość domyślną, tak jak to pokazałem niżej.

```
var { boots: footwear = { size: 10 } } = character
```

W składni destrukturyzacji można używać też składni generowanych nazw właściwości. W takiej sytuacji będziesz zmuszony dodać alias, który będzie mógł zostać użyty jako nazwa zmiennej. A to dlatego, że generowane nazwy właściwości dopuszczają arbitralne wyrażenia, przez co kompilator nie byłby w stanie wywnioskować, co jest nazwą zmiennej. W poniższym przykładzie używamy aliasu `value` oraz generowanych nazw właściwości w celu wyodrębnienia właściwości `boots` z obiektu `character`.

```
var { ['boo' + 'ts']: characterBoots } = character
console.log(characterBoots)
// <- true
```

Ta odmiana destrukturyzacji jest prawdopodobnie najmniej przydatna, ponieważ `characterBoots = character[type]` jest zazwyczaj prostsze niż `{ [type]: characterBoots } = character`, ponieważ jest instrukcją bardziej sekwencyjną. A to znaczy, że możliwość ta jest bardziej przydatna, kiedy masz właściwości, które chcesz zadeklarować w literale obiektu, a nie korzystać z późniejszej instrukcji ich przypisania.

To tyle, jeśli chodzi o obiekty w kwestii destrukturyzacji. A co z tablicami?

2.3.2. Destrukturyzacja tablic

Składnia destrukturyzacji tablic jest podobna do składni dotyczącej obiektów. Poniższy przykład pokazuje obiekt `coordinates`, który został rozłożony na dwie zmienne — `x` oraz `y`. Zauważ, że w notacji użyty został nawias kwadratowy zamiast klamrowego — to oznacza, że używamy destrukturyzacji tablicy, a nie obiektu. Zamiast szpikować swój kod szczegółami implementacji, takimi jak `x = coordinates[0]`, przy użyciu destrukturyzacji możesz przekazać odpowiedni sens w sposób jasny i bez konieczności wyraźnego odnoszenia się do indeksów, po prostu nazywając wartości.

```
var coordinates = [12, -7]
var [x, y] = coordinates
console.log(x)
// <- 12
```

Podczas destrukturyzacji tablic możesz ominąć właściwości, które Cię nie interesują, lub te, do których nie potrzebujesz referencji.

```
var names = ['James', 'L.', 'Howlett']
var [ firstName, , lastName ] = names
console.log(lastName)
// <- 'Howlett'
```

Destrukturyzacja tablic pozwala na ustawienie domyślnych wartości, tak jak w przypadku destrukturyzacji obiektu.

```
var names = ['James', 'L.']
var [ firstName = 'John', , lastName = 'Doe' ] = names
console.log(lastName)
// <- 'Doe'
```

Kiedy w ES5 musisz zamienić miejscami wartości dwóch zmiennych, zazwyczaj odwołujesz się do trzeciej, tymczasowej zmiennej, tak jak w poniższym kodzie.

```
var left = 5
var right = 7
var aux = left
left = right
right = aux
```

Destrukturyzacja pomoże uniknąć deklaracji zmiennej `aux` i skupić się na Twojej intencji. Ponownie w tego typu zastosowaniach destrukturyzacja pomaga przekazać właściwą intencję w sposób bardziej zwięzły i efektywny.

```
var left = 5
var right = 7
[left, right] = [right, left]
```

Ostatnim obszarem destrukturyzacji, który omówimy, są parametry funkcji.

2.3.3. Domyślne ustawienia parametrów funkcji

Parametry funkcji w ES6 również oferują możliwości określenia wartości domyślnych. Poniższy przykład definiuje domyślną wartość parametru `exponent`, przypisując mu najczęściej używaną wartość.

```
function powerOf(base, exponent = 2) {
   return Math.pow(base, exponent)
}
```

Ustawienia domyślne można również zaaplikować do parametrów funkcji strzałki. Jeśli mamy wartości domyślne w funkcjach strzałki, musimy te listy parametrów wstawić w nawiasy, nawet jeśli znajduje się tam tylko jeden parametr.

```
var double = (input = 0) => input * 2
```

Wartości domyślne nie są ograniczone do parametru funkcji znajdującego się najbardziej po prawej, tak jak w niektórych językach programowania. Możesz ustawić wartość domyślną dla każdego parametru, znajdującego się na każdej pozycji.

```
function sumOf(a = 1, b = 2, c = 3) {
   return a + b + c
}
console.log(sumOf(undefined, undefined, 4))
// <- 1 + 2 + 4 = 7
```

W JavaScripcie nierzadko pisze się funkcję z obiektem `options`, która zawiera różne właściwości. Możesz określić domyślny obiekt `options`, jeśli żaden nie zostanie wprowadzony, tak jak pokazałem w poniższym kodzie.

```
var defaultOptions = { brand: 'Volkswagen', make: 1999 }
function carFactory(options = defaultOptions) {
   console.log(options.brand)
   console.log(options.make)
}
carFactory()
// <- 'Volkswagen'
// <- 1999
```

Problem z takim podejściem polega na tym, że w momencie, w którym użytkownik `carFactory` dostarczy obiekt `options`, stracisz wszystkie swoje domyślne wartości.

```
carFactory({ make: 2000 })
// <- undefined
// <- 2000
```

Możemy połączyć domyślne wartości funkcji z destrukturyzacją i otrzymać to, co najlepsze w obu tych podejściach.

2.3.4. Destrukturyzacja parametrów funkcji

Lepszym podejściem niż tylko określenie wartości domyślnej byłaby całkowita destrukturyzacja obiektu `options`, ustawiająca domyślne wartości dla każdej z właściwości oddzielnie, wewnątrz wspólnego wzorca destrukturyzacji. Takie podejście pozwala również odnosić się do każdej z opcji bez konieczności przechodzenia przez obiekt `options`. W ten sposób straciłbyś jednak możliwość bezpośredniej referencji do tego obiektu, co w niektórych sytuacjach mogłoby przysporzyć kłopotów.

```
function carFactory({ brand = 'Volkswagen', make = 1999 }) {
   console.log(brand)
   console.log(make)
}
carFactory({ make: 2000 })
// <- 'Volkswagen'
// <- 2000
```

Aczkolwiek w tej sytuacji znów stracilibyśmy wartość domyślną, jeśli użytkownik nie dostarczy obiektu `options`. Co oznacza, że `carFactory()` zgłosi wyjątek, jeśli obiekt `options` nie będzie istniał. Można tego uniknąć, używając składni pokazanej w poniższym fragmencie kodu, gdzie dodany został pusty obiekt jako domyślna wartość parametru `options`. Pusty obiekt zostanie wówczas zapełniony wartościami domyślnymi ze wzorca destrukturyzacji.

```
function carFactory({
   brand = 'Volkswagen',
   make = 1999
} = {}) {
   console.log(brand)
   console.log(make)
}
carFactory()
// <- 'Volkswagen'
// <- 1999
```

Poza wartościami domyślnymi, możesz użyć destrukturyzacji dla parametrów funkcji, żeby określić rodzaj obiektów, jakie może obsłużyć Twoja funkcja. Przyjrzyj się poniższemu przykładowi, gdzie mamy obiekt `car` o wielu różnych właściwościach. Obiekt `car` opisuje właściciela, rodzaj auta, producenta, datę produkcji oraz preferencje właściciela w momencie zakupu auta.

```
var car = {
   owner: {
      id: 'e2c3503a4181968c',
      name: 'Donald Draper'
   },
   brand: 'Peugeot',
   make: 2015,
   model: '208',
   preferences: {
      airbags: true,
      airconditioning: false,
      color: 'red'
   }
}
```

Gdybyśmy chcieli zaimplementować funkcję, która bierze pod uwagę tylko wybrane właściwości parametru, dobrym pomysłem mogłaby być bezpośrednia referencja do tych właściwości za pomocą jawnej destrukturyzacji. Zaletą takiego rozwiązania jest fakt, że wówczas, czytając sygnaturę funkcji, dowiemy się o każdej wymaganej właściwości.

Jeśli destrukturyzujemy wszystko z góry, łatwo zauważyć, kiedy dane wejściowe nie są zgodne z kontraktem funkcji. Poniższy przykład pokazuje, że można każdą potrzebną nam właściwość zdefiniować w liście parametrów, jasno określając kształt obiektów, które może obsłużyć nasze API `getCarProductModel`.

```
var getCarProductModel = ({ brand, make, model }) => ({
   sku: brand + ':' + make + ':' + model,
   brand,
   make,
   model
})
getCarProductModel(car)
```

Zobaczmy, w czym jeszcze, oprócz domyślnych wartości i wypełniania obiektu `options`, przydatna jest destrukturyzacja.

2.3.5. Przykłady zastosowania destrukturyzacji

Za każdym razem, gdy funkcja zwraca obiekt lub tablicę, destrukturyzacja sprawia, że interakcja staje się o wiele bardziej zwięzła. Poniższy przykład pokazuje funkcję, która zwraca obiekt z kilkoma koordynatami, z których bierzemy tylko te, które nas interesują, czyli `x` oraz `y`. Unikamy deklaracji pośredniej wartości `point`, która raczej przeszkadza, jednocześnie wcale nie poprawiając czytelności kodu.

```
function getCoordinates() {
   return { x: 10, y: 22, z: -1, type: '3d' }
}
var { x, y } = getCoordinates()
```

Przypadek użycia domyślnych wartości danych opcji prosi się o powtórzenie. Wyobraź sobie, że masz funkcję `random`, która wytwarza losowe wartości liczbowe w zakresie od `min` do `max`, domyślnie wynoszące od 0 do 10. To wyjątkowo ciekawa alternatywa dla nazwanych parametrów w językach z silnym typowaniem, takich jak Python czy C#. Wzorzec, w którym jesteś w stanie zdefiniować wartości domyślne dla opcji, a potem pozwolić je pojedynczo nadpisywać, daje ogromne pole manewru.

```
function random({ min = 1, max = 10 } = {}) {
   return Math.floor(Math.random() * (max - min)) + min
}
console.log(random())
// <- 7
console.log(random({ max: 24 }))
// <- 18
```

Wyrażenia regularne to kolejna kwestia idealnie nadająca się do destrukturyzacji — pozwala ona na nazwanie grup zwróconych przez funkcję `match` bez konieczności sięgania po indeksy liczbowe. Mamy tu przykład `RegExp`, którego można by użyć do parsowania prostych dat, oraz przykład destrukturyzacji tych dat na poszczególne ich komponenty. Pierwszy wpis w tak otrzymanej tablicy jest zarezerwowany dla surowych danych wejściowych w postaci ciągu tekstowego i możemy go odrzucić.

```
function splitDate(date) {
   var rdate = /(\d+).(\d+).(\d+)/
   return rdate.exec(date)
}
var [ , year, month, day] = splitDate('2015-11-06')
```

Jednak musisz uważać, bo jeśli wyrażenie regularne nie znajdzie dopasowania, funkcja zwróci `null`. Być może więc lepszym podejściem byłoby sprawdzenie całości pod kątem zgodności, zanim przeprowadzimy destrukturyzację, tak jak to pokazuje poniższy kod.

```
var matches = splitDate('2015-11-06')
if (matches === null) {
   return
}
var [, year, month, day] = matches
```

Przejdźmy teraz do operatorów resztowych i rozłożenia.

2.4. Parametr resztowy i operator rozłożenia

Zanim pojawił się ES6, interakcja z arbitralną liczbą parametrów funkcji była raczej skomplikowana. Trzeba było użyć obiektu `arguments`, który nie jest tablicą, ale ma właściwość `length`. Zazwyczaj kończyło się to zamianą obiektu `arguments` na właściwą tablicę poprzez użycie `Array#slice.call`, tak jak w poniższym fragmencie.

```
function join() {
   var list = Array.prototype.slice.call(arguments)
   return list.join(', ')
}
join('first', 'second', 'third')
// <- 'first, second, third'
```

ES6 oferuje lepsze rozwiązanie tego problemu, a mianowicie parametry resztowe.

2.4.1. Parametry resztowe

Teraz możesz ostatni parametr w każdej funkcji JavaScriptu poprzedzić trzema kropkami, zmieniając go w specjalny „parametr resztowy". Jeśli jest on jedynym parametrem funkcji, wówczas otrzyma wszystkie argumenty przekazywane do funkcji — co działa podobnie jak rozwiązanie `.slice`, które widzieliśmy przed chwilą, ale pozwala uniknąć skomplikowanego konstruktu typu `arguments` oraz jest określone w liście parametrów.

```
function join(...list) {
   return list.join(', ')
}
join('first', 'second', 'third')
// <- 'first, second, third'
```

Parametry nazwane, znajdujące się przed parametrem resztowym nie wejdą do obiektu `list`.

```
function join(separator, ...list) {
   return list.join(separator)
}
join('; ', 'first', 'second', 'third')
// <- 'first; second; third'
```

Zauważ, że funkcja strzałki z parametrem resztowym musi zawierać nawiasy nawet wtedy, kiedy ma tylko jeden parametr. W innym przypadku otrzymalibyśmy błąd `SyntaxError`. W poniższym kodzie mamy piękny przykład na to, że kombinacja funkcji strzałki z parametrem resztowym może dać zwięzłe wyrażenie funkcji.

```
var sumAll = (...numbers) => numbers.reduce(
   (total, next) => total + next
)
console.log(sumAll(1, 2, 5))
// <- 8
```

Porównaj to z wersją ES5 tej samej funkcji — różnica tkwi w złożoności. Chociaż bywa zwięzła, funkcja `sumAll` potrafi zdezorientować czytelnika kodu, jeśli nie jest przyzwyczajony do metody `.reduce`, ale też z powodu użycia aż dwóch funkcji strzałki. To jeden z kompromisów w kwestii skomplikowania kodu, o których opowiem dalej w tej książce.

```
function sumAll() {
   var numbers = Array.prototype.slice.call(arguments)
   return numbers.reduce(function (total, next) {
      return total + next
   })
}
console.log(sumAll(1, 2, 5))
// <- 8
```

Następny w kolejce jest operator rozłożenia (ang. *spread operator*). Również zapisuje się go z użyciem trzech kropek, ale ma nieco inne zastosowanie.

2.4.2. Operator rozłożenia

Operator rozłożenia może być użyty do zamiany każdego iterowalnego obiektu w tablicę. Rozłożenie wyrażenia może być wykorzystane do różnych celów, m.in. związanych z literałem tablicy czy wywołaniem funkcji. Poniższy przykład wykorzystuje `…arguments` w celu zmiany parametrów funkcji w literał tablicowy.

```
function cast() {
   return [...arguments]
}
cast('a', 'b', 'c')
// <- ['a', 'b', 'c']
```

Możemy użyć operatora rozłożenia, żeby rozłożyć ciąg tekstowy na tablicę, zawierającą każdy pojedynczy znak tego ciągu.

```
[...'show me']
// <- ['s', 'h', 'o', 'w', ' ', 'm', 'e']
```

Możesz umieścić dodatkowe elementy po lewej lub prawej stronie operatora i wciąż otrzymasz efekt, który Cię interesuje.

```
function cast() {
   return ['left', ...arguments, 'right']
}
cast('a', 'b', 'c')
// <- ['left', 'a', 'b', 'c', 'right']
```

Rozłożenie jest skutecznym sposobem na łączenie wielu tablic. Poniższy przykład pokazuje, że można rozłożyć tablicę na literał tablicowy, rozszerzając jej elementy w miejscu.

```
var all = [1, ...[2, 3], 4, ...[5], 6, 7]
console.log(all)
// <- [1, 2, 3, 4, 5, 6, 7]
```

Zauważ, że operator rozłożenia nie jest ograniczony do tablicy i obiektu `arguments`. Można go zastosować do każdego iterowalnego obiektu. Iterowalność to protokół w ES6, który pozwala Ci na zmianę każdego obiektu w coś, co będzie się poddawało iteracji. Przyjrzymy się temu w rozdziale 4.

Przesuwanie i rozłożenie

Jeśli chcesz wyodrębnić element czy dwa z początku tablicy, najczęstszym podejściem będzie użycie `.shift`. I chociaż jest ono całkiem funkcjonalne, to jednak poniższy fragment kodu pokazuje, że trudno je zrozumieć na pierwszy rzut oka, ponieważ w celu wyciągnięcia każdorazowo innego elementu z początku obiektu `list` aż dwukrotnie używa metody `.shift`. Punkt ciężkości kładzie się więc tu, tak jak w wielu innych przypadkach sprzed ery ES6, na zmuszanie języka do tego, żeby działał po naszej myśli.

```
var list = ['a', 'b', 'c', 'd', 'e']
var first = list.shift()
var second = list.shift()
console.log(first)
// <- 'a'
```

W ES6 możesz połączyć operator rozłożenia z destrukturyzacją tablicy. Poniższy fragment kodu jest podobny do poprzedniego, z tą różnicą, że używamy jednej linijki i ta jedna linijka lepiej opisuje to, co robimy, zamiast powtarzać wywołanie `list.shift()`, tak jak wcześniej.

```
var [first, second, ...other] = ['a', 'b', 'c', 'd', 'e']
console.log(other)
// <- ['c', 'd', 'e']
```

Stosowanie operatora rozłożenia sprawia, że możesz skoncentrować się na implementacji potrzebnych Ci opcji, nie zaprzątając sobie głowy samym językiem. Większa efektywność wyrażeń i krótsza walka z ograniczeniami języka to wspólne cechy nowych możliwości wprowadzonych przez ES6.

Wcześniej, przed ES6, za każdym razem, gdy miałeś dynamiczną listę argumentów, którą trzeba było zastosować do wywołania funkcji, musiałeś użyć `.apply`. Było to mało eleganckie, bo `.apply` ustala kontekst `this`, a w takiej sytuacji nie chcesz sobie przecież dodatkowo zawracać głowy.

```
fn.apply(null, ['a', 'b', 'c'])
```

Oprócz rozłożenia na tablice, możesz również rozłożyć elementy tablicy na poszczególne parametry w wywołaniu funkcji. Poniższy przykład pokazuje, że można używać operatora rozłożenia, by przekazać dowolną liczbę argumentów do funkcji `multiply`.

```
function multiply(left, right) {
   return left * right
}
var result = multiply(...[2, 3])
console.log(result)
// <- 6
```

Rozłożenie argumentów na poszczególne parametry w wywołaniu funkcji można dowolnie połączyć z argumentami regularnymi, podobnie jak z literałami tablicy. Następny przykład wywołuje funkcję `print` z kilkoma parametrami regularnymi i kilkoma tablicami rozłożonymi do listy parametrów. Zauważ, że parametr resztowy `list` bez problemu obsługuje wszystkie parametry. Rozłożenie i parametr resztowy sprawiają, że intencje kodu stają się bardziej przejrzyste, a jednocześnie nie umniejszają funkcjonalności.

```
function print(...list) {
   console.log(list)
}
print(1, ...[2, 3], 4, ...[5])
// <- [1, 2, 3, 4, 5]
```

Innym ograniczeniem `.apply` jest fakt, że połączenie go ze słowem kluczowym `new` podczas tworzenia egzemplarza obiektu rozwleka całe wyrażenie. Mamy tutaj przykład takiego połączenia, które ma na celu utworzenie obiektu `Date`. Zapomnij na chwilę o tym, że miesiące w dacie JavaScriptu liczą się od zera, co sprawia, że 11 oznacza grudzień, i pomyśl, ile z poniższych linii kodu trzeba poświęcić na to, żeby język zaczął łaskawie robić to, czego potrzebujemy. A chodzi przecież tylko o utworzenie instancji obiektu `Date`.

```
new (Date.bind.apply(Date, [null, 2015, 11, 31]))
// <- Thu Dec 31 2015
```

Jak pokazuje następny fragment, operator rozłożenia usuwa tę niepotrzebną zawiłość, dzięki czemu zostajemy tylko z tym, co naprawdę istotne. Mamy słowo kluczowe `new`, które używa operatora `...`, żeby rozłożyć listę argumentów w wywołaniu funkcji, i mamy `Date`. Tyle.

```
new Date(...[2015, 11, 31])
// <- Thu Dec 31 2015
```

Tabela na następnej stronie podsumowuje przykłady użycia, o których wspominaliśmy, omawiając operator rozłożenia.

2.5. Literały szablonu

Literały szablonu to ogromne ulepszenie regularnych ciągów tekstowych w języku JavaScript. Zamiast używać pojedynczych czy podwójnych cudzysłowów, szablony literału deklaruje się poprzez użycie lewego apostrofu (ang. *backtick*), tak jak to pokazuję poniżej.

```
var text = `To mój pierwszy literał szablonu`
```

Przykład użycia	ES5	ES6
Konkatenacja	`[1, 2].concat(more)`	`[1, 2, ...more]`
Dodanie elementów tablicy do listy	`list.push.apply(list, items)`	`list.push(...items)`
Destrukturyzacja	`a = list[0], other = list.slice(1)`	`[a, ...other] = list`
Połączenie słowa kluczowego `new` i metody `apply`	`new (Date.bind.apply(Date, [null,2015,31,8]))`	`new Date(... [2015,31,8])`

Biorąc pod uwagę fakt, że szablony literału są ograniczone lewymi apostrofami, możesz teraz zadeklarować ciąg tekstowy zawierający zarówno pojedynczy, jak i podwójny cudzysłów, bez konieczności otaczania ich znakami ucieczki, tak jak to było dotychczas.

```
var text = `Znalazłem w Google'u informację o tym „smaczku" z ES6 i jestem pod wrażeniem!`
```

Jedną z najatrakcyjniejszych możliwości szablonów literału jest ich zdolność do interpolacji wyrażeń JavaScriptu.

2.5.1. Interpolacja ciągów tekstowych

Z wykorzystaniem szablonów literału jesteś w stanie wewnątrz swojego szablonu interpolować każde wyrażenie JavaScriptu. Kiedy program dochodzi do takiego ciągu tekstowego, wykonuje go, a my otrzymujemy z powrotem już skompilowany wynik. Poniższy przykład interpoluje zmienną `name` do literału szablonu.

```
var name = 'Shannon'
var text = `Witaj, ${ name }!`
console.log(text)
// <- 'Witaj, Shannon!'
```

Ustaliliśmy już, że możesz użyć każdego wyrażenia JavaScriptu, nie tylko zmiennych. Możesz pomyśleć o każdym wyrażeniu wewnątrz literału szablonu jako definiowaniu zmiennej przed uruchomieniem szablonu i potem łączeniu tej zmiennej z pozostałym ciągiem tekstowym. Jednak to sprawia, że kod staje się łatwiejszy do opanowania, bo nie zawiera ręcznie łączonych ciągów tekstowych i wyrażeń JavaScriptu. Zmienne, których używasz w wyrażeniu, funkcje, które wywołujesz itp., powinny być dostępne w tym zasięgu, w którym zdefiniowano szablon.

W zależności od stylu kodowania, który stosujesz, możesz zdecydować, ile treści chcesz zmieścić w takich interpolowanych wyrażeniach. Poniższy fragment kodu pokazuje np. utworzenie instancji obiektu `Date` i sformatowanie go do zapisu czytelnego dla człowieka — i wszystko to dzieje się wewnątrz literału szablonu.

```
`Obecna data i godzina to ${ new Date().toLocaleString() }.`
// <- obecna data i godzina to 26.08.2015, 15:15:20.
```

Możesz interpolować również operacje matematyczne.

```
`2+3 równa się ${ 2 + 3 }`
// <- 2+3 równa się 5'
```

Możesz nawet zagnieżdżać szablony literału, ponieważ są one poprawnymi wyrażeniami JavaScriptu.

```
`Ten szablon literału ${ `jest ${ 'zagnieżdżony' }` }!`
// <- 'Ten szablon literału jest zagnieżdżony!'
```

Inną zaletą literału szablonu jest obsługa wielowierszowych ciągów tekstowych.

2.5.2. Wielowierszowe literały szablonu

Jeśli chciałeś w JavaScripcie utworzyć wielowierszowy ciąg tekstowy, zanim pojawiły się literały szablonu, musiałeś stosować znaki końca linii, konkatenację, tablice, a nawet wykorzystywać tricki z użyciem komentarzy. Poniższy fragment podsumowuje jedną z najbardziej typowych reprezentacji wielowierszowego literału szablonu zgodnego ze standardem ES5.

```
var escaped =
'Pierwszy wiersz\n\
Drugi wiersz\n\
A potem trzeci'
var concatenated =
'Pierwszy wiersz \n' +
'Drugi wiersz\n' +
'A potem trzeci'
var joined = [
'Pierwszy wiersz',
'Drugi wiersz',
'A potem trzeci'
].join('\n')
```

W ES6 możesz zamiast tego użyć lewego apostrofu. Szablon literału domyślnie obsługuje wielowierszowe ciągi tekstowe. Zauważ, że nie ma znaków końca linii, nie ma też konkatenacji ani tablic.

```
var multiline =
`Pierwszy wiersz
Drugi wiersz
A potem trzeci`
```

Wielowierszowe ciągi tekstowe naprawdę cieszą, kiedy masz np. kawał kodu w HTML, do którego chcesz interpolować jakieś zmienne. Jeśli potrzebujesz wyświetlić listę wewnątrz szablonu, możesz iterować listę, mapując jej elementy na odpowiednie znaczniki, i zwrócić taki połączony wynik interpolowanego wyrażenia. Dzięki temu deklaracja komponentów wewnątrz szablonu staje się niezwykle prosta, co widać w kodzie poniżej.

```
var book = {
   title: 'Modułowy ES6',
   excerpt: ' Tutaj pojawia się jakiś dobrze przefiltrowany HTML ',
   tags: ['es6', 'template-literals', 'es6-in-depth']
}
var html = `<article>
   <header>
      <h1>${ book.title }</h1>
   </header>
   <section>${ book.excerpt }</section>
   <footer>
      <ul>
         ${
            book.tags
               .map(tag => `<li>${ tag }</li>`)
               .join('\n      ')
         }
```

```
      </ul>
    </footer>
  </article>`
```

Szablon, który właśnie przygotowaliśmy, zwróciłby w danych wyjściowych to, co widzimy w poniższym kodzie. Zauważ, że zachowane[3] zostały odstępy, a tagi `<li>` poprawnie zinterpretowane dzięki temu, że połączyliśmy je, używając kilku spacji.

```
<article>
  <header>
    <h1>Modułowy ES6</h1>
  </header>
  <section>Tutaj pojawia się jakiś dobrze przefiltrowany HTML</section>
  <footer>
    <ul>
      <li>es6</li>
      <li>template-literals</li>
      <li>es6-in-depth</li>
    </ul>
  </footer>
</article>
```

Wadą wielowierszowych ciągów tekstowych jest właśnie formatowanie wcięć. Poniższy przykład pokazuje typowe wcięcia w kodzie we wnętrzu literału szablonu zawartego wewnątrz funkcji. Chociaż moglibyśmy nie oczekiwać żadnych wcięć, nasz ciąg tekstowy będzie przesunięty o sześć spacji w prawo.

```
function getParagraph() {
  return `
    Drogi Rodzie,
    To jest literał szablonu, który ma wcięcie równe sześciu spacjom. A Ty,
    zdaje się, nie spodziewałeś się w ogóle żadnego wcięcia.
    Nico
  `
}
```

Chociaż nie jest to rozwiązanie idealne, możesz uniknąć tych wcięć za pomocą funkcji narzędziowej i usunąć je z każdego wiersza wynikowego ciągu tekstowego.

```
function unindent(text) {
  return text
    .split('\n')
    .map(line => line.slice(4))
    .join('\n')
    .trim()
}
```

Czasem może lepiej sprawdzić, co będzie wynikiem interpolowanego wyrażenia, zanim wkleimy je do naszego szablonu. Dla przypadków bardziej zaawansowanych można wykorzystać inną możliwość szablonów literału, zwaną *szablonami z tagami*.

[3] Użycie literału szablonu nie zachowuje automatycznie odstępów. W wielu przypadkach wystarczy wprowadzić odpowiednie wcięcia, by zadziałało, ale bądź ostrożny w stosowaniu bloków kodu z wcięciami, bo może to spowodować niechciane formatowanie z powodu bloków zagnieżdżonych.

2.5.3. Szablony z tagami

Domyślnie JavaScript interpretuje ukośnik \ jako znak kodowania o specjalnym znaczeniu. Przykładowo \n oznacza nową linię, \u00f1 to ñ itd. Możesz ominąć te zasady, używając szablonu z tagiem String.raw. Następny fragment pokazuje szablon literału wykorzystujący String.raw, który sprawia, że \n nie będzie już interpretowane jako nowa linia.

```
text = String.raw`"\n" jest wzięte dosłownie.
Czyli nie zostało zinterpretowane jako znak nowej linii.`
console.log(text)
// "\n" jest wzięte dosłownie.
// Czyli nie zostało zinterpretowane jako znak nowej linii.
```

Prefiks String.raw, który dodaliśmy do naszego literału szablonu, to szablon z tagiem, używany do parsowania danego szablonu. Szablony z tagiem otrzymują parametr z tablicą zawierającą statyczne części szablonu oraz wyniki wykonanych wyrażeń, każdy w zależności od własnego parametru.

W ramach przykładu rozważ szablon literału z tagiem z poniższego fragmentu.

```
tag`Witaj, ${ name }. Tak bardzo ${ emotion } , że Cię widzę!`
```

Wyrażenie w szablonie z tagiem będzie w praktyce przetłumaczone na poniższe wywołanie funkcji.

```
tag(
   ['Witaj, ', '. Tak bardzo ', ' , że Cię widzę!'],
   'Maurice',
   'cieszę się'
)
```

Wynikiem będzie ciąg tekstowy zbudowany w następujący sposób: do każdego elementu szablonu dołączane jest jedno wyrażenie tak długo, aż użyte zostaną wszystkie elementy szablonu.

Trudno interpretować listę argumentów bez rzucenia okiem na potencjalną implementację domyślnego literału szablonu tag, dlatego teraz właśnie tym się zajmiemy.

Poniższy fragment kodu pokazuje możliwą implementację domyślnego szablonu tag. Oferuje on tę samą funkcjonalność, co literał szablonu, gdy szablon z tagiem nie jest bezpośrednio użyty. Redukuje tablicę parts do pojedynczej wartości, a mianowicie wyniku literału szablonu. Wynik ten jest zainicjalizowany pierwszym elementem part i potem każdy następny element (part) szablonu jest poprzedzany jedną z wartości values.

Użyliśmy składni parametru resztowego ...values w celu łatwiejszego wychwycenia wyniku obliczenia każdego z wyrażeń w szablonie. Korzystamy też z funkcji strzałki z domyślną instrukcją return, zakładając, że jej wyrażenie jest relatywnie proste.

```
function tag(parts, ...values) {
   return parts.reduce(
      (all, part, index) => all + values[index - 1] + part
   )
}
```

Możesz wypróbować szablon tag korzystający z kodu, takiego jak ten w poniższym przykładzie. Zobaczysz, że otrzymasz takie same dane wyjściowe, jeśli pominiesz tag, ponieważ kopiujemy tu zachowanie domyślne.

```
var name = 'Maurice'
var emotion = 'cieszę się'
var text = tag`Witaj, ${ name }. Tak bardzo ${ emotion } ,że Cię widzę!`
console.log(text)
// <- 'Witaj, Maurice, tak bardzo cieszę się, że Cię widzę!'
```

Szablonu z tagiem można użyć w wielu sytuacjach. Jedną z nich może być zamiana danych wejściowych od użytkownika na wersaliki, które sprawią, że treść ciągu tekstowego nabierze charakteru bardziej ironicznego. Tak właśnie zadziała kod w poniższym przykładzie. Zmodyfikowaliśmy nieco `tag`, żeby zamienił zinterpolowane fragmenty ciągu tekstowego na wersaliki.

```
function upper(parts, ...values) {
   return parts.reduce((all, part, index) =>
      all + values[index - 1].toUpperCase() + part
   )
}
var name = 'Maurice'
var emotion = 'cieszę się'
upper`Witaj, ${ name }. Tak bardzo ${ emotion }, żę Cię widzę!`
// <- 'Witaj MAURICE, tak bardzo CIESZĘ SIĘ, że Cię widzę '
```

Zdecydowanie bardziej przydatnym przykładem jest natomiast automatyczna filtracja wyrażeń interpolowanych w naszym szablonie, możliwa dzięki użyciu szablonu z tagiem. Mając szablon, w którym wszystkie wyrażenia miałyby pochodzić z danych wejściowych od użytkownika, moglibyśmy hipotetycznie wykorzystać bibliotekę *sanitize* („cenzuruj”) do usunięcia tagów HTML czy innych niepożądanych elementów. W ten sposób moglibyśmy zapobiegać atakom XSS, uniemożliwiając wstawienie podejrzanych fragmentów kodu w pola tekstowe na naszej stronie.

```
function sanitized(parts, ...values) {
   return parts.reduce((all, part, index) =>
      all + sanitize(values[index - 1]) + part
   )
}
var comment = 'Evil comment<iframe src="http://evil.corp">
   </iframe>'
var html = sanitized`<div>${ comment }</div>`
console.log(html)
// <- '<div>Evil comment</div>'
```

Uff, niemal dopadłoby nas złośliwe `<iframe>`.

Zestaw składniowych nowości w ES6 zamykają instrukcje `let` oraz `const`.

2.6. Instrukcje let oraz const

Instrukcja `let` jest jedną z najbardziej znanych nowych możliwości ES6. Działa jak instrukcja `var`, ale różni się od niej zasięgiem.

JavaScript ma od zawsze skomplikowany zestaw reguł dotyczących ustalania zasięgu, co doprowadza programistów do szaleństwa, kiedy próbują zrozumieć, jak w tym języku działają zmienne. Aż w końcu odkrywasz hoisting i JavaScript zaczyna nabierać nieco więcej sensu. *Hoisting* oznacza, że zmienna jest brana z miejsca, w którym została zadeklarowana, i przenoszona na początek swojego zasięgu. Przykładowo może wyglądać to tak, jak w poniższym kodzie.

```
function isItTwo(value) {
   if (value === 2) {
      var two = true
   }
   return two
}
isItTwo(2)
// <- true
isItTwo('two')
// <- undefined
```

Tego typu JavaScript działa, mimo że zmienna `two` została zadeklarowana w odgałęzieniu kodu i uzyskujemy do niej dostęp z zewnątrz, spoza tej gałęzi. Takie zachowanie wynika z faktu, że zmienne deklarowane za pomocą instrukcji `var` przyjmują domyślnie zasięg otoczenia, np. funkcji, czy zasięg globalny. A to, w połączeniu z hoistingiem, oznacza, że kod, który napisaliśmy wcześniej, zostanie zinterpretowany tak samo, jakbyśmy napisali coś w poniższym stylu.

```
function isItTwo(value) {
   var two
   if (value === 2) {
      two = true
   }
   return two
}
```

Czy nam się to podoba, czy nie, hoisting bywa bardziej mylący niż użycie zmiennych o zasięgu blokowym. Jednak zasięg blokowy działa raczej na poziomie nawiasów klamrowych, a nie na poziomie całej funkcji.

2.6.1. Zasięg blokowy i deklaracje let

Jeśli potrzebujemy głębszego poziomu zasięgu, to zamiast deklarować nowy obiekt `function`, możemy skorzystać z zasięgu bloku, który pozwala na wykorzystanie istniejących odgałęzień kodu, takich jak utworzone przez instrukcje `if`, `for` czy `while`. Możesz też budować dowolnie nowe bloki z użyciem nawiasów klamrowych {}. Może o tym nie wiesz, ale JavaScript pozwala na tworzenie nieograniczonej liczby bloków, jeśli tylko tak trzeba.

```
{{{{{ var deep = 'Ten fragment jest dostępny w zewnętrznym zasięgu.'; }}}}}
console.log(deep)
// <- 'Ten fragment jest dostępny w zewnętrznym zasięgu.'
```

Używając instrukcji `var`, która korzysta z zasięgu leksykalnego, można uzyskać dostęp do zmiennej `deep` również spoza bloku, w którym została zadeklarowana, i nie dostać ostrzeżenia o błędzie. Czasami jednak informacja o błędzie może być przydatna; dzieje się tak szczególnie wtedy, kiedy znajdujesz się w sytuacji, którą opisuje któreś z poniższych stwierdzeń.

- Dotarcie do wewnętrznej zmiennej łamie pewien rodzaj domknięcia w naszym kodzie.
- Wewnętrzna zmienna w ogóle nie powinna znaleźć się w zewnętrznym zasięgu.
- Blok, o który chodzi, ma wiele bratnich bloków, które chcą korzystać z tej samej nazwy zmiennej.
- Jeden z bloków nadrzędnych już posiada zmienną o tej samej nazwie, ale wciąż można by jej użyć w bloku wewnętrznym.

Instrukcja `let` jest więc alternatywą dla instrukcji `var`. Działa zgodnie z zasadami zasięgu blokowego, a nie domyślnego zasięgu leksykalnego. Kiedy stosujemy `var`, jedynym sposobem na uzyskanie głębszego zasięgu jest utworzenie funkcji zagnieżdżonej, ale przy użyciu `let` można po prostu otworzyć kolejny nawias klamrowy. A to oznacza, że nie potrzebujesz całkowicie nowej funkcji, żeby utworzyć nowy zasięg — wystarczy tylko {}.

```
let topmost = {}
{
   let inner = {}
   {
      let innermost = {}
   }
   // próby uzyskania stąd dostępu do innermost wyrzuciłyby błąd
}
// próby uzyskania stąd dostępu do inner wyrzuciłyby błąd
// próby uzyskania stąd dostępu do innermost wyrzuciłyby błąd
```

Przydatnym aspektem instrukcji `let` jest to, że możesz użyć ich podczas deklaracji pętli `for` i wtedy jej zmienne będą ograniczone do zasięgu wewnątrz pętli, tak jak pokazuje poniższy przykład.

```
for (let i = 0; i < 2; i++) {
   console.log(i)
       // <- 0
       // <- 1
}
console.log(i)
// <- i i is not defined
```

Jeśli weźmiemy pod uwagę fakt, że zmienne zdeklarowane wewnątrz pętli za pomocą instrukcji `let` będą związane z zasięgiem każdego kroku w tej pętli, to ich wiązania będą działały poprawnie w połączeniu z wywołaniem funkcji asynchronicznej, w przeciwieństwie do dotychczasowej konstrukcji z użyciem `var`. Spójrzmy na konkretne przykłady.

Na początek przyjrzyjmy się typowemu przykładowi na to, jak działa zasięg w `var`. Wiązanie `i` ma zasięg ograniczony do funkcji `printNumbers`, a jego wartość zwiększa się do 10 przy każdym przekroczeniu czasu wywołania zwrotnego, tak długo jak działa metoda wywołania zwrotnego — raz na 100 milisekund — `i` ma wartość równą 10 i dlatego za każdym razem wypisuje ten sam wynik.

```
function printNumbers() {
   for (var i = 0; i < 10; i++) {
      setTimeout(function () {
         console.log(i)
      }, i * 100)
   }
}
printNumbers()
```

Natomiast użycie deklaracji `let` zwiąże zmienną z zasięgiem bloku. I wtedy — owszem — w każdej iteracji wartość zmiennej wciąż będzie inkrementowana, ale będzie tworzone też nowe wiązanie, co oznacza, że wywołanie zwrotne zachowa odwołanie do wiązania przechowywującego wartość `i` w momencie wywołania zwrotnego, co oznacza, że nareszcie otrzymamy wyniki od 0 do 9.

```
function printNumbers() {
   for (let i = 0; i < 10; i++) {
      setTimeout(function () {
         console.log
```

```
        }
    }
printNumbers()
```

Jest jeszcze jedna kwestia związana z instrukcją `let` — to koncepcja zwana „tymczasowo martwą strefą”.

2.6.2. Tymczasowo martwa strefa

Mówiąc krótko — jeśli napiszesz kod taki, jak w poniższym przykładzie, otrzymasz błąd. Kiedy tylko program dotrze do zasięgu, ale przed dotarciem do instrukcji `let`, próba dostępu do zmiennej deklarowanej rzeczonym `let` poskutkuje błędem. Taka sytuacja znana jest jako tymczasowo martwa strefa (**TDZ** — ang. *Temporal Dead Zone*).

```
{
    console.log(name)
        // <- ReferenceError: name is not defined
    let name = 'Stephen Hawking'
}
```

Jeśli Twój kod próbuje uzyskać dostęp do właściwości `name` w jakikolwiek sposób, zanim program dojdzie do instrukcji `let name`, wówczas pojawi się błąd. Deklaracja funkcji, która odnosi się do zmiennej `name`, zanim zostanie ona zdefiniowana, jest w porządku, o ile funkcja ta nie jest wykonywana w momencie, w którym `name` znajduje się w tymczasowej martwej strefie, i o ile `name` będzie w TDZ do momentu, zanim program dojdzie do instrukcji `let name`. Poniższy fragment kodu nie zgłosi komunikatu o błędzie, ponieważ `return name` nie jest wykonywane, dopóki `name` nie opuści TDZ.

```
function readName() {
    return name
}
let name = 'Stephen Hawking'
console.log(readName())
// <- 'Stephen Hawking'
```

Jednak już poniższy fragment zgłosi błąd, bo dostęp do `name` występuje, zanim ta zmienna opuści TDZ.

```
function readName() {
    return name
}
console.log(readName())
// ReferenceError: name is not defined
let name = 'Stephen Hawking'
```

Zauważ, że semantyka tych przykładów nie zmienia się, kiedy dla `name` nie jest zadeklarowana żadna wartość. Poniższy kod również zgłosi błąd, ponieważ wciąż próbuje uzyskać dostęp do `name`, zanim opuści ona TDZ.

```
function readName() {
    return name
}
console.log(readName())
// ReferenceError: name is not defined
let name
```

Poniższy fragment kodu natomiast zadziała poprawnie, ponieważ opuszcza TDZ, zanim w jakikolwiek sposób uzyskuje dostęp do `name`.

```
function readName() {
   return name
}
let name
console.log(readName())
// <- undefined
```

Jedyna trudność polega tu na tym, by pamiętać, że można deklarować funkcję, która uzyskuje dostęp do zmiennej znajdującej się w tymczasowo martwej strefie, o ile instrukcje odpowiadające za taki dostęp nie zostaną wykonane przed dotarciem do deklaracji `let`.

Cały sens tymczasowo martwej strefy jest taki, że łatwiej wyłapuje się błędy w kodzie, polegające na próbie dostępu do zmiennej, zanim zostanie ona zadeklarowana. Przed wprowadzeniem ES6 takie sytuacje miały często miejsce — ze względu na hoisting i niedostatek konwencji kodowania. W ES6 łatwiej tego uniknąć. Pamiętaj, że hoisting ma wciąż rację bytu przy użyciu `let`. To znaczy, że zmienne będą tworzone w momencie wejścia do zasięgu i powstanie tymczasowo martwa strefa, ale te zmienne będą niedostępne aż do chwili, gdy wykonywanie kodu dojdzie do miejsca rzeczywistej deklaracji zmiennych, i dopiero wtedy opuścimy TDZ, a zmienne znów staną się dostępne.

Przebrnęliśmy przez tymczasową martwą strefę! Teraz przyszedł czas na zajęcie się instrukcją `const`, wyrażeniem podobnym do `let`, ale też istotnie różniącym się od niego.

2.6.3. Deklaracje const

Deklaracja `const`, podobnie jak `let`, działa na prawach zasięgu bloku i semantyki TDZ. W zasadzie semantyka TDZ została zaimplementowana ze względu na `const` i dopiero później zaadaptowana do `let`, dla zachowania spójności. Powodem, dla którego `const` potrzebowało semantyki TDZ, jest fakt, że inaczej można byłoby przypisać wartość do zmiennej poddanej hoistingowi, zanim wykonywanie programu dotrze do deklaracji `const`, co oznacza, że deklaracja ta sama w sobie wyrzuciłaby błąd. Tymczasowo martwa strefa dostarcza rozwiązania dla problemu, jakim była możliwość przypisania `const` tylko w czasie deklaracji, i pomaga uniknąć potencjalnych problemów podczas używania `let` oraz ułatwia ewentualną implementację innych możliwości, które czerpią korzyści z semantyki TDZ.

Poniższy fragment kodu pokazuje, że `const`, dokładnie tak jak `let`, tworzy zmienne o zasięgu bloku.

```
const pi = 3.1415
{
   const pi = 6
   console.log(pi)
       // <- 6
}
console.log(pi)
// <- 3.1415
```

Wspominałem, że istnieją zasadnicze różnice pomiędzy `let` oraz `const`. Pierwszą jest fakt, że zmienne `const` muszą zostać zadeklarowane za pomocą inicjalizatora. Deklaracja `const` wymaga wartości początkowej, co pokazuję w poniższym przykładzie.

```
const pi = 3.1415
const e // SyntaxError, missing Initializer
```

Oprócz przypisania podczas inicjalizacji, różnica polega też na tym, że do zmiennych zadeklarowanych za pomocą `const` nie można później niczego przypisać. Jeśli zmienna została raz zainicjalizowana za pomocą `const`, nie możesz zmienić jej wartości. W trybie ścisłym próby modyfikacji wartości zmiennej `const` wyrzucą błąd. W trybie nieścisłym po prostu nie zadziałają, bez komunikowania o błędzie, tak jak to się dzieje w poniższym fragmencie.

```
const people = ['Tesla', 'Musk']
people = []
console.log(people)
// <- ['Tesla', 'Musk']
```

Zauważ, że tworzenie zmiennej za pomocą instrukcji `const` nie musi oznaczać, że przypisana wartość staje się niezmienna. To częste źródło nieporozumień, dlatego gorąco zachęcam do uważnej lektury poniższego ostrzeżenia.

Zmienne zadeklarowane za pomocą const nie są niezmienne

Użycie instrukcji `const` oznacza tylko tyle, że zmienna będzie zawsze miała referencję do tego samego obiektu lub pierwotnej wartości, ponieważ ta referencja się nie zmieni. Referencja sama w sobie pozostaje niezmienna, ale wartość przechowywana w tej zmiennej nie staje się przez to stała i dana raz na zawsze.

Poniższy przykład pokazuje, że nawet jeśli referencja zmiennej `people` nie może się zmienić, to tablica sama w sobie może zostać zmodyfikowana. Gdyby tablica była niezmienna, byłoby to niemożliwe.

```
const people = ['Tesla', 'Musk']
people.push('Berners-Lee')
console.log(people)
// <- ['Tesla', 'Musk', 'Berners-Lee']
```

Deklaracja `const` tylko zapobiega zmianie referencji powiązanej ze zmienną. Innym sposobem na zilustrowanie tej różnicy jest poniższy fragment kodu, gdzie tworzymy zmienną `people` za pomocą deklaracji `const`, a potem przypisujemy tę zmienną do zwykłego wiązania `var humans`. Możemy przepisać zmienną `humans` do innej referencji, ponieważ nie została zadeklarowana za pomocą `const`. Aczkolwiek `people` przepisać do innej referencji już nie możemy, właśnie ze względu na użycie deklaracji `const`.

```
const people = ['Tesla', 'Musk']
var humans = people
humans = 'zło'
console.log(humans)
// <- 'zło'
```

Jeśli naszym celem byłoby sprawienie, że wartość pozostanie niezmienna, musielibyśmy użyć funkcji `Object.freeze`. Wykorzystanie tej funkcji zapobiega rozszerzeniom danego obiektu, co pokazuje poniższy przykład.

```
const frozen = Object.freeze(
   ['Ice', 'Icicle', 'Ice cube']
)
frozen.push('Water')
// Uncaught TypeError: Can't add property 3
// object is not extensible
```

Poświęćmy teraz chwilę na omówienie zalet `const` oraz `let`.

2.6.4. Zalety deklaracji const oraz let

Nigdy nie powinno się używać nowych możliwości tylko dlatego, że są nowe. Tych oferowanych w ES6 warto używać przede wszystkim wtedy, gdy naprawdę istotnie poprawiają czytelność kodu i łatwość zapanowania nad nim. Instrukcja `let` potrafi w wielu przypadkach uprościć kod tam, gdzie normalnie użyłbyś instrukcji `var` na górze funkcji, żeby hoisting nie przyniósł niepożądanych skutków. Natomiast używając instrukcji `let`, mógłbyś umieścić swoje deklaracje na górze bloku kodu, zamiast na czele całej funkcji, i oszczędzić sobie mentalnych wycieczek na górę zasięgu.

Stosowanie instrukcji `const` to świetny sposób, żeby zapobiec niepożądanym zdarzeniom. Poniższy fragment kodu to murowany scenariusz na błąd, gdzie referencję do zmiennej `items` przekazujemy do funkcji `checklist`, która zwraca API `todo`, które z kolei wchodzi w interakcję z rzeczoną referencją `items`. Gdyby zmodyfikować zmienną `items` o referencję do innego elementu z listy, zaczęłyby się kłopoty — API `todo` wciąż by działało z dawną wartością `items`, ale `items` odnosiłoby się teraz do czegoś innego.

```
var items = ['a', 'b', 'c']
var todo = checklist(items)
todo.check()
console.log(items)
// <- ['b', 'c']
items = ['d', 'e']
todo.check()
console.log(items)
// <- ['d', 'e'], byłoby teraz ['c'], jeśli elementy pozostałyby stałe
function checklist(items) {
   return {
      check: () => items.shift()
   }
}
```

Ten rodzaj problemu trudno debugować, ponieważ mógłbyś stracić sporo czasu, zanim doszedłbyś do tego, że to referencja została zmodyfikowana. Instrukcja `const` pomaga unikać takich sytuacji, gdyż zwraca błędy wykonania (w trybie ścisłym), co pozwala wyłapać błąd w krótkim czasie.

Podobną korzyścią płynącą z używania instrukcji `const` jest możliwość wyłapania gołym okiem zmiennych, które nie są ponownie przypisane. Użycie `const` daje jasny sygnał, że wiązanie zmiennej jest tylko do odczytu i dlatego, czytając kod, mamy przynajmniej tę jedną kwestię z głowy.

Gdybyśmy wybrali domyślne korzystanie z `const` i użycie `let` dla zmiennych, które wymagają przepisania, wszystkie zmienne miałyby ten sam typ zasięgu, co ułatwiłoby zrozumienie kodu. Powodem, dla którego `const` proponuje się czasem jako „domyślny” typ deklaracji zmiennych, jest to, co przekonuje do tej deklaracji najbardziej — `const` zapobiega ponownym przypisaniom, tworzy zmienne o zasięgu bloku, a do zadeklarowanych wiązań nie można uzyskać dostępu, zanim nie zostanie wykonana deklaracja. Instrukcja `let` pozwala na ponowne przypisanie, ale poza tym zachowuje się jak `const`, więc jest raczej oczywistym wyborem, jeśli chodzi o zmienne wymagające ponownego przypisania.

Z drugiej strony, `var` jest bardziej złożoną deklaracją, bo chociaż trudno jej użyć w odgałęzieniach z powodu zasięgu funkcji, to pozwala na ponowne przypisanie i uzyskanie dostępu do zmiennych, zanim program wykona ich deklarację. Instrukcja `var` jest gorsza od deklaracji `const` i `let`, które robią mniej, i dlatego jest mniej uznawana w nowoczesnym JavaScripcie.

Na łamach tej książki będziemy praktykowali domyślne wykorzystanie `const` oraz użycie instrukcji `let` dla zmiennych, dla których pożądane jest ponowne przypisanie. Więcej o rozsądnych argumentach stojących za taką decyzją przeczytasz w rozdziale 9.

Skorowidz

F

G

H

I

J

K

L

Ł

M

N

O

P

R

S

T

U

W

Z

Notatki

Poznaj JavaScript z najlepszej strony!

Niegdyś JavaScript służył głównie twórcom stron WWW, obecnie jest używany nie tylko do pisania aplikacji przeglądarkowych, ale także do tworzenia aplikacji mobilnych i desktopowych, oprogramowywania różnych urządzeń, a nawet w projektach skafandrów kosmicznych dla NASA. By w pełni wykorzystać możliwości nowoczesnego JavaScriptu, trzeba dobrze poznać zmiany wprowadzone wraz ze standardem ECMAScript 6 (ES6). Są to bardzo daleko idące ulepszenia, dotyczące m.in. składni, semantyki, wbudowanych obiektów i metod.

Ta książka jest przeznaczona dla każdego, kto chce pogłębić znajomość JavaScriptu i gruntownie zapoznać się z ES6. Zawarty tu materiał został poukładany w taki sposób, aby ułatwić proces nauki i pozwolić na stopniowe przyswajanie kolejnych zagadnień. Po wprowadzeniu do języka i nowoczesnych narzędzi przedstawiono stosowanie funkcji asynchronicznych, destrukturyzację obiektów, dynamiczne importy, obietnice oraz generatory asynchroniczne. Opisano nowe elementy ES6: kolekcje, obiekty, ulepszenia obiektów wbudowanych. W książce znalazł się również szereg praktycznych uwag, dzięki którym tworzenie poprawnego, wydajnego i elastycznego kodu z pewnością będzie dużo łatwiejsze.

Niektóre zagadnienia omówione w książce:

- procesy rozwoju standardów JavaScript
- techniki sterowania przepływem programu
- tworzenie map obiektów
- obiekty wbudowane w ES6
- nowe obiekty Proxy i Reflect
- natywne moduły JavaScript

Nicolás Bevacqua jest inżynierem tworzącym interfejsy użytkownika. Jest też niekwestionowanym ekspertem programowania i niestrudzonym piewcą idei *open source*. Doskonale zna JavaScript i chętnie dzieli się swoją wiedzą z innymi pasjonatami tego języka. Pisze książki o kodowaniu i publikuje artykuły na Ponyfoo.com. Mieszka w Buenos Aires w Argentynie.

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

KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-283-4229-3
9 788328 342293

INFORMATYKA W NAJLEPSZYM WYDANIU
Cena: 49,00 zł