ECMAScript 6 – standard przyszłości

Programiści długo musieli czekać na kolejny krok rozwojowy języka JavaScript. Po kolejnych próbach stworzenia nowego, lepszego standardu członkowie ECMA International doszli do porozumienia i w połowie 2015 roku oficjalnie zamknęli standard nowej wersji języka ECMAScript. Tym samym developerzy otrzymali narzędzie, które będzie w przyszłości podstawą każdej aplikacji webowej, a biorąc pod uwagę nadchodzący Internet Rzeczy, ECMAScript 6 i jego następne wersje będą dosłownie otaczać każdego z nas. Z tego powodu warto już teraz poznać nowe funkcjonalności standardu przyszłości.

HISTORIA JĘZYKA JAVASCRIPT

Początki języka JavaScript sięgają połowy lat 90. XX wielu. Był to okres, w którym witryny internetowe zaczynały pełnić rolę interaktywnych aplikacji. Statyczny język HTML został wzbogacony o aplety języka Java. Kod apletów w myśl zasady „write once, run everywhere" mógł być uruchomiony na każdej platformie systemowej, dając użytkownikom systemu Windows, Unix/Linux czy też Mac OS dostęp do interaktywnych stron z poziomu przeglądarki internetowej. Takie rozwiązanie miało jednak swoje wady. Binarny kod apletu musiał zostać pobrany, a następnie uruchomiony w przeglądarce, wykorzystując w tym celu środowisko języka Java. Zważywszy na to, że większość użytkowników Internetu korzystała z łączy telefonicznych, wczytywanie apletu było dla nich zniechęcające. Drugą wadą tego rozwiązania była konieczność uruchomienia maszyny wirtualnej Java i zaangażowanie w to mocy obliczeniowej komputera lokalnego. Cały proces trwał więc zbyt długo i pochłaniał za dużo mocy obliczeniowej. Problem ten postanowił rozwiązać Brendan Eich, programista w firmie Netscape. W maju 1995 roku w 10 dni stworzył pierwszą wersję nowego języka, nazywając go LiveScript. W grudniu tegoż roku, za zgodą firmy SUN, zmienił jego nazwę na JavaScript. Był to zabieg czysto marketingowy, mający na celu spopularyzowanie nowego języka poprzez nawiązanie do innego, bardzo popularnego w tamtym okresie.

Od 1996 roku organizacja ECMA (*European Computer Manufacturers Association*) wzięła pod swoje skrzydła język JavaScript w celu stworzenia jednolitych standardów. W wyniku prac powstał oficjalny standard języka zwany ECMAScript, a w niedalekiej przyszłości inżynierowie opublikowali udoskonaloną wersję standardu pod nazwą ECMAScript 2. W efekcie producenci przeglądarek internetowych mogli implementować jednolity standard języka. Pozwoliło to na uruchamianie skryptów niezależnie od przeglądarki. Idea „write once, run everythere" zaczerpnięta z języka Java zaczynała funkcjonować również w JavaScript.

W 1999 roku nastąpił kolejny przełom w rozwoju JavaScript; został opublikowany następny standard zwany ECMAScript 3. Wprowadził on nowe możliwości, jak np. obsługę wyjątków `try... catch`, wyrażenia regularne, instrukcję `switch`, operator `===`, pętlę do... `while` oraz metody numeryczne, obsługi dat i ciągów znakowych. Ciężko dziś wyobrazić sobie pisanie aplikacji bez korzystania z którejś z tych funkcjonalności. Dlatego też ta wersja standardu stała się najbardziej popularna i jest stosowana do dnia dzisiejszego. Opierają się o nią zarówno proste skrypty, jak i zaawansowane aplikacje webowe.

Następnym krokiem, jaki musiał zrobić nowoczesny język programowania, było wprowadzenie programowania obiektowego. Trzecia wersja standardu miała przygotować język na wprowadzenie modelu klas, przestrzeni nazw, zasięgów blokowych, pakietów i wielu innych technik programowania obiektowego. Rozpoczęto pracę nad kolejną wersją standardu. W międzyczasie firma Macromedia wprowadziła swoją własną implementację języka nazwaną ActionScript. Miał on być rozwinięciem dotychczasowych standardów w kierunku czwartej wersji ECMAScript. Głównym problemem w tej interpretacji było przerwanie kompatybilności wstecznej. Problem spotęgowało pojawienie się w 2005 roku technologii AJAX. Dzięki niej i powstającym bibliotekom takim jak dojo, prototype stworzenie interaktywnej aplikacji stało się o wiele łatwiejsze. Popularność JavaScript wzrastała w ogromnym tempie. Interaktywne aplikacje webowe zaczęły być motorem napędowym Internetu. Pomimo oczywistych wad JS zaczął być powszechnie stosowany. Nic więc dziwnego, że społeczność developerska nie akceptowała faktu, że wraz z wprowadzeniem ECMAScript 4 ich aplikacje przestaną działać. Z tego powodu pomimo kilku szkiców koncepcyjnych i propozycji nigdy nie ukazał się oficjalny standard. Czwarta wersja została ostatecznie anulowana. ActionScript pozostał odrębnym językiem, niemającym już wiele wspólnego z dalszym rozwojem ECMAScript.

Przed osobami odpowiedzialnymi za rozwój języka pojawił się więc kolejny problem: jak ulepszać język, jednocześnie zachowując kompatybilność wsteczną. Punktem wyjścia stała się trzecia wersja języka. Dopiero w 2009 roku rozwojowa wersja oznaczona numerem 3.1 doczekała się oficjalnego wydania. Wraz z tym wydarzeniem zmieniono jej nazwę na ECMAScript 5. Nowy standard wprowadzał m.in. natywną obsługę JSON, semantykę obiektów globalnych, settery i gettery. Najważniejszą jednak zmianą było wprowadzenie trybu ścisłego. Usystematyzował on tworzenie kodu poprzez wprowadzenie restrykcji, jak np. brak możliwości przypisania wartości do niezdefiniowanej zmiennej.

Powyższe zmiany stworzyły solidne podstawy do rozbudowy języka o programowanie obiektowe. W 2013 roku światło dzienne ujrzała szósta wersja standardu. Wprowadza ona ogromną ilość zmian. Część z nich była już obecna w porzuconym ECMAScript 4. Teraz jednak jej implementacja zachowuje kompatybilność wsteczną.

ECMASCRIPT 6

Zanim nowa wersja standardu ujrzała światło dzienne, musiało dojść do zgody trzech głównych graczy na scenie przeglądarek internetowych: firm Google, Microsoft oraz organizacji Mozilla. Od 2010 roku każda z firm lobbowała nad wprowadzeniem swojego standardu jako głównego nurtu w rozwoju języka. Google stworzył język Dart, niekompatybilny z językiem JavaScript, ale z możliwością kompilowania do niego. W tym czasie Microsoft rozwijał TypeScript. Stanowił on nadzbiór JS, co w praktyce miało oznaczać możliwość uruchomienia aplikacji napisanych w JavaScript za pomocą interpretera. W porównaniu do Dart zachowywał więc kompatybilność wsteczną. W tym czasie Mozilla pracowała nad poprawieniem wydajności obsługi skryptów

w swojej przeglądarce, tworząc asm.js. Dojście do porozumienia wydawało się trudne. Punktem przełomowym był moment, w którym Google zdało sobie sprawę z faktu, że nie jest w stanie przekonać producentów przeglądarek i developerów do szerszego stosowania języka Dart. Zrobił więc ukłon w stronę TypeScript. Od tego czasu prace ruszyły do przodu. Wspólna praca doprowadziła do jednej wizji przyszłości języka, jednocześnie na dalszy plan odsuwając swoje własne rozwiązania, nie zawsze będące kompatybilne z głównym nurtem ECMAScript. Szósta wersja standardu została oficjalnie zatwierdzona 17 czerwca 2015 roku.

PISAĆ W ES6 JUŻ DZIŚ

Opublikowanie standardu bynajmniej nie zakończyło prac nad rozwojem. Teraz producenci przeglądarek internetowych oraz innych środowisk, jak np. Node.js, które bazują na JavaScript, dostosowują swoje produkty do nowych wytycznych. Tak więc chcąc zacząć pisać aplikacje z użyciem ES6, pierwszą rzeczą, na jaką musimy zwrócić uwagę, jest sprawdzenie, czy użyte funkcjonalności są wspierane przez środowisko, w jakim będą uruchamiane nasze skrypty. Doskonałym narzędziem jest `ECMAScript compatibility table` znajdujące się pod adresem http://kangax.github.io/compat-table/es6/. Jeśli środowisko, w jakim pracujemy, nie wspiera danej właściwości standardu, możemy skorzystać z bibliotek kompilujących nasz kod do starszej wersji. Przykładem takiej biblioteki jest Babel.js. Dołączając ją do projektu, uzyskujemy dostęp do niewspieranych w środowisku funkcjonalności. Dobrym narzędziem do testowania i praktycznej nauki jest http://www.es6fiddle.net. Trzeba jednak pamiętać, że działa ono w przeglądarce internetowej i jest oparte o bibliotekę Traceur, a wprowadzany kod jest kompilowany w locie. Przed rozpoczęciem pracy należy więc sprawdzić w tabeli kompatybilności, czy dany element standardu jest obsługiwany przez tę bibliotekę.

Przykłady zawarte w tym artykule zostały przetestowane w różnych środowiskach języka JavaScript. Zdarza się, że niektóre fragmenty kodu nie będą chciały działać w przeglądarce Chrome lub w starszych wersjach serwera node. W takich sytuacjach przy fragmencie kodu zostanie umieszczona stosowna informacja. Niektóre środowiska należy odpowiednio skonfigurować. Przykładem może być uruchamianie skryptów w środowisku Node. Aby była możliwa praca z ES6, uruchamianie skryptu powinno odbywać się z dodaniem opcji --harmony. Postać takiej komendy może mieć postać node --harmony script.js. Bardziej wymagające jest uruchomienie tzw. trybu eksperymentalnego w Google Chrome. Tryb ten można włączyć, wpisując w adresie przeglądarki: chrome://flags/#enable-javascript-harmony, a następnie aktywując tryb `harmony`. Po restarcie przeglądarka będzie miała dostęp do najnowszych funkcji JavaScript.

Pisząc aplikację dostępną dla szerszego grona użytkowników, kiedy nie wiemy, na jakich platformach będą pracować, musimy wziąć pod uwagę, że nie wszyscy będą mieli możliwość natywnej obsługi standardu ECMAScript 6. Dlatego też warto skorzystać z narzędzi kompilujących napisany przez nas kod do standardu ECMAScript 5, który jest obsługiwany przez przeważającą ilość platform. Wspomniana wcześniej biblioteka babel.js znakomicie się do tego nadaje, a jej różnorodne formy użycia pozwolą dostosować ją do różnych typów projektów, z jakimi przyjdzie nam pracować. Najszybszym sposobem na kompilację naszych skryptów jest skorzystanie z linii poleceń. W poniższym przykładzie najpierw zainstalujemy bibliotekę Babel.js jako globalny moduł serwera Node, a następnie wydamy polecenie kompilacji skryptu `script.js`:

Listing 1. Kompilacja pliku z linii poleceń

```
$ npm install --global babel
$ babel script.js

// Kompilacja w locie
$ babel script.js --watch --out-file script-compiled.js
```

W drugim przykładzie uruchamiamy sprawdzanie, czy oryginalny plik został zmodyfikowany. Jeśli biblioteka wykryje takie zdarzenie, automatycznie skompiluje nasz plik. Dzięki temu za każdym razem, gdy edytujemy nasz plik i zapiszemy zmiany, od razu możemy przystąpić do testowania jego funkcjonowania. Poniżej znajduje się przykład skompilowanego kodu zawierającego proste użycie funkcji strzałkowej (omawianej w dalszej części artykułu).

Llisting 2. Wynik kompilacji kodu ES6 do ES5 z użyciem biblioteki babel.js

```
//Przed kompilacją
(function test(){
  setTimeout( () =>{
    console.log(this);
  }, 1000);
})();

//Po kompilacji
"use strict";

(function test() {
  var _this = this;

  setTimeout(function () {
    console.log(_this);
  }, 1000);
})();
```

Pełna dokumentacja biblioteki wraz z przykładami znajduje się pod adresem https://babeljs.io. Tam również znajdują się ciekawe przykłady z wykorzystaniem ECMAScript 6.

Wielu programistów korzysta z narzędzi wspomagających, takich jak np. Gulp.js. Warto zmodyfikować swoje skrypty budujące wersje produkcyjne aplikacji, dodając do nich obsługę kompilatorów ECMAScript 6. Użyty wcześniej Babel.js jest szeroko wykorzystywany w różnego rodzaju systemach automatyzacji pracy. My skorzystamy z modułu stworzonego na potrzeby tworzenia aplikacji w systemie Gulp modułu. Aby móc z niego skorzystać, wydajemy polecenie `npm install --save-dev gulp-babel`. Poniższy listing zawiera prosty przykład użycia w/w modułu w celu stworzenia skompilowanej wersji naszego skryptu.

Listing 3. Użycie biblioteki babel.js w systemie automatyzacji pracy Gulp

```
var gulp = require('gulp');
var babel = require('gulp-babel');

gulp.task('compile', function () {
  return gulp.src('src/script.js')
    .pipe(babel())
    .pipe(gulp.dest('dist'));
});
```

Mając już pewne podstawy teoretyczne oraz wiedząc, w jaki sposób używać nowych właściwości języka, możemy przejść do omawiania poszczególnych mechanizmów, jakie wprowadza nowy standard. Standard języka jest bardzo rozległy i omówienie dokładnie wszystkich mechanizmów wykracza poza ramy artykułu. Z tego względu zachęcam do zgłębiania możliwości języka we własnym zakresie. Oficjalna dokumentacja i stale rosnące zasoby w sieci pozwalają na zapoznanie się z wzorcami używanymi w ECMAScript 6.

W niniejszym artykule zastosowano podejście polegające na przedstawieniu prezentowanych właściwości w formie opisowej z użyciem krótkich fragmentów kodu. Fragmenty te mają za zadanie w sposób praktyczny przedstawić zasadę działania aktualnie omawianego zagadnienia. Dzięki temu można skupić się na zrozumieniu danej właściwości, jednocześnie nie zaśmiecając kodu niepotrzebnymi liniami. W codziennej pracy tworzony kod naturalnie będzie bardziej rozbudowany, jednak na potrzeby artykułu zaprezentowane przykłady są wystarczające.

ECMASCRIPT 6 – STANDARD PRZYSZŁOŚCI

Zasięg blokowy

Do tej pory zmienne w języku JavaScript posiadały zasięg funkcyjny. Oznaczało to, że każda zmienna dostępna jest z poziomu funkcji, w jakiej została zdefiniowana. Do czasu wprowadzenia trybu ścisłego w ECMAScript 5 zmienne deklarowane z pominięciem słowa kluczowego `var` posiadały zakres globalny. Wymuszenie deklaracji zmiennej przed jej pierwszym użyciem ograniczyło możliwość nadpisania zmiennych globalnych. Pozostała jednak nierozwiązana sprawa zasięgu blokowego. Przyjrzyjmy się przykładowi poniżej:

Listing 4. Przykład zasięgu zmiennych w JavaScript

```
function mainFoo(){
  'use strict';

  var variable1 = 10;
  var active   = true;

  if( active ){
    subFoo();
  }

  function subFoo(){
    var variable2 = 20;
    console.log( variable1 );
  }

  console.log( active );
  console.log( variable2 );

};

mainFoo();
```

W funkcji głównej `mainFoo()` mamy dwie zmienne: `variable1` oraz `active`. Jeśli wartość zmiennej `active` to logiczna prawda, wtedy wywoływana jest funkcja `subFoo();`. Mamy tu przykład dwóch zakresów. Jeden to zakres nadrzędny w funkcji `mainFoo()`, drugi w funkcji `subFoo()`. Próba wywołania w/w kodu spowoduje wyświetlenie błędu w momencie odwołania się do zawartości zmiennej `variable2`. Dzieje się tak, ponieważ zmienne zdeklarowane w funkcji `subFoo()` nie są dostępne w zakresie znajdującym się poziom wyżej. Inaczej sprawa przedstawia się ze zmienną `active`. Została ona zadeklarowana w zakresie nadrzędnym i jest dostępna również w zakresie podrzędnym. Jakakolwiek modyfikacja jej w funkcji `subFoo()` będzie skutkowała modyfikacją jej w całym zasięgu funkcji `mainFoo()`.

Jeszcze ciekawiej robi się w przypadku pętli. Jeden z najpopularniejszych przypadków zastosowania pętli w JavaScript wygląda następująco:

Listing 5. Przykład pętli for

```
for( var i = 0; i< tablica.length; i++){
….
}
// console.log(i);
```

Po wyjściu z pętli zmienna `i` przechowuje wartość z ostatniego przejścia. Taka sytuacja może prowadzić do powstania błędów przy ponownej próbie użycia tej zmiennej. Dodatkowo fakt istnienia zmiennej po wykonaniu pętli wprowadza w pewnym sensie nieład logiczny. Zakładamy bowiem, że zmienna `i` jest używana jedynie w tym konkretnym kontekście, jako iterator pętli, i nie jest używana nigdzie poza nią. Następujący przykład jeszcze wyraźniej prezentuje taką sytuację:

Listing 6. Zakres zmiennych w instrukcji warunkowej

```
var variable1 = 20;
if( true ){
  var tmp = variable1;
  variable1 = 3 * 4 + tmp;
}
console.log( tmp ); // 20
```

Zmienna `tmp` pomimo zdefiniowania jej wewnątrz instrukcji warunkowej jest dostępna również po jej wykonaniu. Podobnie jak w poprzednim przykładzie, ponowne użycie tej zmiennej może prowadzić do powstania błędów.

Rozwiązaniem tego problemu jest wprowadzenie zasięgu blokowego. Ograniczy to „wyciek" zmiennych i jednocześnie sprawi, że działanie kodu będzie zgodne z intencjami programisty. W ECMAScript 6 wprowadzenie zasięgu blokowego odbywa się poprzez słowo kluczowe `let`, zastępujące znane nam `var`. Przykład z Listingu 6, w którym zastosujemy zasięg blokowy, przedstawia się następująco:

Listing 7. Przykład zastosowania zasięgu blokowego

```
var variable1 = 20;
if( true ){
  let tmp = variable1;
  variable1 = 3 * 4 + tmp;
}
console.log( typeof tmp ); // undefined
```

Obecnie zmienna `tmp` nie jest dostępna po wyjściu z instrukcji warunkowej. W tym miejscu należy zaznaczyć, że deklaracja zmiennej poprzez `let` nie wpływa bezpośrednio na zmienną, lecz na blok, w którym została zadeklarowana. Umożliwia to przesłonięcie zmiennej zadeklarowanej w bloku nadrzędnym, np.:

Listing 8. Przesłonięcie zmiennych w bloku podrzędnym

```
function f2(){

  'use strict';
  let v1 = 1;

  if( true ){

    let v1 = 2;
    console.log( v1 );
  }

  console.log( v1 );

  {
    let v1 = 3;
    console.log( v1 );
  }

  console.log( v1 );
}

f2();
//2
//1
//3
//4
//1
```

Ma to swoje konsekwencje. Przede wszystkim możemy tworzyć lokalne zasięgi blokowe, niebędące częścią żadnej instrukcji warunkowej ani pętli. Taki przykład mamy w Listingu 8, gdzie zmienna v1 została zadeklarowana lokalnie z wartością 3. Do tej pory chcąc uzyskać zasięg lokalny, należało skorzystać z funkcji. Bardzo często był to zabieg wpływający niekorzystnie na czytelność kodu. Użycie nawiasów klamrowych sprawia, że kod staje się bardziej intuicyjny i łatwiejszy do czytania. Istotne jest, aby podczas korzystania z `let` pamiętać o tym, że zasięg lokalny tworzą właśnie nawiasy klamrowe. Z tego też powodu skrócona instrukcja warunkowa nie zadziała poprawnie:

Listing 9. Błędne deklarowanie zmiennej

```
if( true )
  let y =1;
```

Wiemy już, jak definiować zasięg funkcyjny i zasięg blokowy. Naturalnie więc pojawia się pytanie, który jest lepszy i czy w ECMAScript 6 `let` zastąpi `var`. Odpowiedź jest krótka: nie. Obie deklaracje zmiennych mają swoje zadania w kodzie. Zasięg funkcyjny, który obowiązywał przez lata, jest bardzo dobrze utrwalony w świadomości developerów. Duża ilość skryptów oparta jest o ten mechanizm. Zasięg blokowy powinien być stosowany wszędzie tam, gdzie zachodzi taka potrzeba. Szczególnie w sytuacjach, w których potrzebujemy zdefiniować zmienne wykorzystywane w danej chwili, jak np. zmienne tymczasowe w instrukcjach warunkowych czy też iteratory w pętlach. Tworzenie zasięgów lokalnych na potrzeby wykonania obliczeń też ma swoje uzasadnienie, jak np. przesłonięcie zmiennej, gdy chcemy mieć pewność, że nie nadpiszemy jej oryginalnej wartości zmiennej użytej wcześniej w kodzie. Istnieją jednak przypadki, w których zastosowanie `let` spowoduje nieprawidłowe funkcjonowanie. Poniższy przykład prezentuje taką właśnie sytuację:

Listing 10. Nieodpowiednie użycie let

```
(function (){
  'use strict';

  var showTmp = function(){

    console.log('Wartosc tmp to: ' + tmp);
  };

  if ( true ){
    let tmp = 10;
    setTimeout( showTmp, 2000 );

  }
})();
```

Zmienna `tmp` zadeklarowana za pomocą `let` ma zakres instrukcji warunkowej. Jej użycie poza nią skończy się niepowodzeniem. Jeśli zmienna zostałaby zadeklarowana z użyciem `var`, byłaby dostępna również w funkcji `showTmp()`.

Stałe

Kolejną nowością wprowadzoną w ECMAScript 6 są stałe. Do tej pory JavaScript nie oferował mechanizmu definiowania stałych. ES5 wprowadził możliwość symulacji danych tego typu. Odbywa się to poprzez stworzenie niemodyfikowalnych właściwości obiektu globalnego `window` (lub `global` – w zależności od środowiska uruchomieniowego).

Listing 11. Symulacja stałych

```
Object.defineProperty(typeof global === "object" ? global :
window, "EULER", {
  value:        2.71828,
  writable:     false,
  enumerable:   true,
  configurable: false
});
```

Taka deklaracja jest jednak uciążliwa w implementacji, szczególnie gdy mamy do czynienia z większą ilością deklarowanych stałych. Stała, zdefiniowana w ten sposób, będąca w istocie właściwością obiektu globalnego, sama ma zasięg globalny. Co więcej, próba nadania jej nowej wartości zakończy się niepowodzeniem, jednak żaden komunikat o błędzie nie zostanie wyświetlony i skrypt będzie kontynuował działanie. Wszystko to sprawia, że pisanie kodu z wykorzystaniem takich obiektów może prowadzić do konfliktów, a cały kod może działać nieprawidłowo. Dlatego też wraz z nadejściem nowego standardu wprowadzono obsługę stałych w sposób bardzo podobny jak ma to miejsce w innych językach:

Listing 12. Deklaracja stałej PI

```
function foo(){
  'use strict';
  const PI = 3.14;
  console.log( PI );
  //PI = 10
  // error
}
foo();
```

Jak widać powyżej, do deklaracji stałej użyjemy słowa kluczowego `const`. Tworzy ono typ danych, którego wartość jest chroniona przed zmianą. Deklaracja stałej może nastąpić w przestrzeni globalnej bądź w funkcji, a nowo utworzone dane mają zasięg blokowy. Podobnie jak w przypadku zmiennych deklarowanych za pomocą `let`, stałe są widoczne w zasięgach niższego stopnia, w drugą stronę już nie. Oznacza to, że stałe mogą być przesłaniane w zasięgach potomnych, np:

Listing 13. Przesłanianie zmiennych

```
function foo(){
  'use strict';

  const z1 = 20;
  console.log(z1);
  // 20

  {
    const z1 = 10;
    console.log(z1);
    //10
  }
  console.log(z1);
  // 20
}
```

Każda stała posiada swój typ. Do sprawdzenia typu używamy dobrze znanego operatora `typeof`. Stałe mogą być typu prostego, np. liczbą lub obiektem bądź funkcją. Wszystkie poniższe deklaracje zmiennych są poprawne.

Listing 14. Deklaracje różnych typów stałych

```
const ZM1 = 10
const ZM2 = "JAKIS TEKST";
const ZM3 = {
prop1 : 22,
prop2 : "ABC",
prop3 : {}
};
const ZM4 = function(){
  return "TO JEST STALA NR. " + 4;

};
const ZM5 = (function(){
  return 2 * 2;
});

console.log(typeof ZM1);   // “number”
console.log(typeof ZM2);   // “string”
console.log(typeof ZM3);   // “object”
console.log(typeof ZM4);   // “function”
console.log(typeof ZM4()); // “string”
console.log(typeof ZM5);   // “number”
```

Jak możemy zauważyć, kiedy do stałej przypiszemy funkcję, to jej typ – co jest logiczne – to "`function`". Jeśli jednak wywołamy stałą tak jak zwykłą funkcję, to jej typem będzie typ wartości zwracanej. Podobnie ma się sprawa podczas deklaracji stałej, której przypisujemy funkcję samowywołującą. Operator `typeof` zwróci nazwę typu wartości zwracanej przez funkcję jako typ stałej. Taka sytuacja może być myląca, szczególnie kiedy funkcja nie zwraca żadnej wartości. Wtedy też typem stałej będzie "`undefined`", pomimo że została ona zadeklarowana i ma przypisany do siebie wynik działania funkcji. Może tak się stać, jeśli nie przewidzimy wszystkich warunków wewnątrz funkcji, np:

Listing 15. Niespodziewana wartość stałej

```
const LICZBA = function( val ){
  if( val < 0 ) return "ujemna";
  if( val > 0 ) return "dodatnia";
}
console.log("Cyfra 0 jest " + LICZBA( 0 ) );
//Cyfra 0 jest undefined
```

Ostatnim przypadkiem szczególnym przy omawianiu stałych są stałe deklarowane jako obiekty. W Listingu 14 stała ZM3 była obiektem. Z definicji stałej wiemy, że nie może być do niej przypisana żadna inna wartość. Jeśli więc chcielibyśmy stałej ZM3 przypisać ciąg znaków „NOWA WARTOŚĆ ZMIENNEJ ZM3", zostanie zgłoszony błąd. Co jednak, jeśli chcielibyśmy zmienić wartość konkretnego pola albo dodać nowe? Okazuje się, że pola obiektu nie są chronione przed zmianą wartości. Nie możemy zmienić typu stałej z obiektu na np. liczbę, ale modyfikować wartości pól już tak. Nic więc nie stoi na przeszkodzie, aby nadać im nową, zupełnie inną wartość:

Listing 16. Nadanie nowych wartości atrybutom

```
const STALA = {
  prop1 : 'pierwsza wartość',
  prop2 : 'druga wartość'
};

console.log( STALA );
//Object { prop1="pierwsza wartość", prop2="druga wartość"}

STALA.prop1 = 10;

console.log( STALA );
//Object { prop1=10, prop2="druga wartość"}
```

Nie wszystkie przeglądarki jednak jednakowo interpretują modyfikowanie atrybutów. Przykładem może być dodanie nowych lub przypisanie do stałej innego obiektu. Chrome oraz Firefox zgłasza błąd, podczas gdy Safari nie.

Listing 17. Zmiana atrybutów w obiekcie stałym

```
const STALA = {
  prop1 : 'pierwsza wartość',
  prop2 : 'druga wartość'
};

STALA.prop3 = "trzecia wartość";
```

Kod z Listingu 17 zadziała w przeglądarce Safari. Przeglądarki Chrome i Firefox zgłoszą błąd.

Arrow Functions

Standard ES6 wprowadza nowy sposób pracy z funkcjami, jaki nie był dostępny w poprzednich wersjach, czyli tzw. funkcje strzałkowe. Czasami też można spotkać się z określeniem Bold Arrow. Jest to mechanizm szybkiego tworzenia funkcji anonimowych, który na pierwszy rzut oka może wydawać się mało intuicyjny. Z czasem jednak staje się mechanizmem ułatwiającym tworzenie kodu, a przy tym oferującym inne niż w przypadku standardowych funkcji możliwości.

Na poniższym listingu przedstawiono przykłady funkcji zwracających kwadrat podanej liczby. Pierwsza funkcja została stworzona przy użyciu standardowej składni definicji funkcji. Druga z użyciem nowego sposobu definiowania funkcji:

Listing 18. Dwa sposoby definiowania funkcji

```
// standardowy zapis
var square = function ( val ) {
  return val * val;
};
console.log(square(2)); // 4
// nowy zapis
var square = val => val * val;
console.log(square(2)); //4
```

Nowy zapis, pomimo że jest krótszy i ma prostszą składnię, wydaje się mało czytelny. Aby zrozumieć ten skrócony zapis, należy wrócić do podstaw działania funkcji. Funkcja jest blokiem kodu, do którego w większości przypadków przesyłamy określone dane, następnie zostają one wykorzystane do np. obliczeń, po czym funkcja zwraca jakąś wartość wynikową. Mamy więc dwie porcje danych: argumenty i wyrażenie. Listing 19 przedstawia ten schemat za pomocą dwóch zapisów:

Listing 19. Schemat funkcji w standardowym i w nowym zapisie

```
function ( argumenty ){ return wrażenie; }
( argumenty ) => { return wyrażenie; }
```

Argumenty, jakie przekazujemy do funkcji, znajdują się w nawiasach okrągłych. Jeśli funkcja posiada jeden argument, nawiasy te można pominąć. Jeśli jednak funkcja posiada zero lub więcej niż jeden argument, nawiasy okrągłe są obowiązkowe. W związku z tym wszystkie przykłady definiowania funkcji przedstawione poniżej są prawidłowe.

Listing 20. Przykłady definicji funkcji strzałkowych

```
var foo = arg => { return arg * arg; }

var foo = ( arg ) => { return arg * arg; }

var foo = () => { return 10; }

var foo = ( arg1, arg2 ) => { return arg1 * arg2; }
```

Argumenty przekazywane są do wyrażenia. Jeśli wyrażenie zawiera się w jednej linii, to jego wartość zostanie zwrócona jako wynik działania funkcji. W takim przypadku można pominąć stosowanie słowa kluczowego `return` oraz stosowanie nawiasów klamrowych. Bardziej rozbudowane funkcje wymagają stosowania `return`, aby określić, jaką wartość zwraca funkcja. Jeśli nie określimy, co funkcja zwraca, to podobnie jak w standardowych funkcjach wartością zwracaną będzie undefined. Każda z definicji funkcji z Listingu 21 jest więc prawidłowa:

Listing 21. Różne ciała funkcji w nowym zapisie

```
var foo = arg => arg * arg;
var foo = () => { return 10 };
var foo = () => window.navigator ;
var foo = arg => { return arg * arg; }
var foo = arg => {
var tmp = arg + arg;
return tmp * arg;
}
var foo = arg => {
  window.params.a1 = arg;
}
```

Teraz przykład z Listingu 18 staje się jaśniejszy. Funkcja przyjmuje jeden argument. Pomijamy więc nawias okrągły. W bloku wyrażenia wykonujemy operację mnożenia. Wynik tej operacji zwracamy jako wynik działania funkcji. W związku z tym, że nasza operacja mieści się w jednej linii kodu, możemy pominąć słowo `return` oraz nawiasy klamrowe.

Jak więc widać, funkcje strzałkowe sprawdzają się w sytuacjach, kiedy mamy do czynienia z prostymi obliczeniami. Znakomicie sprawdzają się jako funkcje wywołania zwrotnego. Przykładem może być znalezienie liczb mniejszych niż 5:

Listing 22. Przykład wykorzystania funkcji strzałkowych

```
var numbers = [1,2,3,4,5,6,7,8,9,10];
var smallNumbers = numbers.filter( number => number < 5 );
```

Oprócz uproszczonej składni funkcje strzałkowe oferują jeszcze jeden niezwykle przydatny mechanizm, tzw. `lexical this` (*this leksykalny*) Oznacza to, że funkcja nie tworzy nowego zasięgu. Dzięki temu nie ma potrzeby definiowania dodatkowych zmiennych przechowujących aktualny `this`.

Wyobraźmy sobie sytuację, w której tworzymy klasę `Osoba`. Klasa ta posiada atrybut `wiek`. Atrybut ten jest zwiększany co sekundę za pomocą interwału czasowego `setInterval`, który jako parametr przyjmuje funkcję wywołania zwrotnego tworzącą swój własny zakres. Z tego też powodu, jeśli chcemy zmodyfikować wartość atrybutu `wiek`, musimy posłużyć się zmienną pomocniczą, przechowującą referencję do niego, czyli `this`. Taki kod przedstawiony jest na Listingu 23:

Listing 23. Zmienna przechowująca referencję do this

```
function Osoba( imie ){
  'use strict';
  var self = this;
  this.wiek = 1;
  this.imie = imie;
  this.zycie = setInterval( function(){
    self.wiek++;
  }, 1000);

  this.smierc = function(){
    clearInterval( self.zycie );
    console.log( self.imie + ' miał '+ self.wiek + ' lat. ' );
  }
}
var maciek = new Osoba('Maciek');

setTimeout(function() {
  maciek.smierc(); // Maciek miał 11 lat
},8000);
```

Analizując powyższy kod, napotykamy miejsca, w których konieczne było posłużenie się zmienną pomocniczą `self`. Interwał czasowy oraz funkcja `smierc` tworzą swoje własne zakresy, w których przesłaniają `this`. Aby móc dostać się do atrybutów obiektu, należy skorzystać ze zmiennej `self` przechowującej `this` obiektu. Problem ten nie występuje w przypadku funkcji strzałkowych:

Listing 24. Lexical this z użyciem funkcji strzałkowych

```
function Osoba( Imie ){
  'use strict';
  this.wiek = 1;
  this.imie = Imie;
  this.zycie = setInterval( () => this.wiek++ , 1000);
  this.smierc = () => {
    clearInterval( this.zycie );
    console.log( this.imie + ' miał(a) '+ this.wiek + ' lat. ' );
  }
}
var magda = new Osoba('Magda');
setTimeout(()=>{
  magda.smierc(); //Magda mial(a) 8 lat
},8000);
```

Jak widać, nowy sposób pracy z funkcjami może okazać się bardzo przydatny w sytuacjach, kiedy potrzebujemy użyć funkcji, których zawartość będzie zawierała prosty algorytm. Oczywiście, można tworzyć zaawansowane funkcje wykonujące skomplikowane operacje, jednak w tym wypadku bardziej przydatne będzie skorzystanie ze standardowej formy definicji funkcji. Należy pamiętać, że funkcje strzałkowe są funkcjami anonimowymi i powinny być stosowane w sytuacjach, w których dotychczas korzystaliśmy właśnie z funkcji anonimowych.

Parametry funkcji

Zmiany w funkcjach nie kończą się na wprowadzeniu funkcji z użyciem strzałek. Sposób przesyłania parametrów do funkcji również uległ modyfikacji. Sporą zmianą jest z pewnością wprowadzenie parametrów domyślnych funkcji. Brak tej funkcjonalności przez bardzo długi okres czasu irytował developerów. Pomimo używania prostego wzorca symulującego domyślne wartości, Java Script w tym kontekście pozostawał daleko za innymi językami programowania.

Aby symulować domyślną wartość przesyłanego parametru, musieliśmy sprawdzić, jaki jest typ przesyłanego argumentu. Jeśli był niezdefiniowany, oznaczało to, że nie został przesłany. Wtedy następowało nadanie mu wartości domyślnej:

Listing 25. Przykład definiowania wartości domyślnych parametrów

```
function sumowanie( arg1, arg2 ){
  if( typeof arg1 === 'undefined' ){
    arg1 = 1;
  }
  if( arg2 === 'undefined'){
    arg2 = 1;
  }

  return arg1 + arg2;
}
```

W nowym standardzie nie musimy zaśmiecać sobie ciała funkcji zbędnymi blokami warunkowymi. Podobnie jak w innych językach programowania, określamy domyślną wartość w miejscu definiowania argumentów podczas deklarowania funkcji:

Listing 26. Domyślne wartości parametrów w ECMAScript 6

```
function sumowanie( arg1 = 1, arg2 = 1 ){
  return arg1 + arg2;
}
```

Wartości domyślne nie są jedyną zmianą wprowadzoną w parametrach funkcji. Nowością jest wprowadzenie parametru `rest`. Takie rozwiązanie zostało wprowadzone m.in. w języku PHP w wersji 5.6+. Służy ono do zebrania wszystkich nadmiarowych argumentów przekazanych do funkcji i udostępnienia ich w formie tablicy. Dzięki temu uzyskujemy dostęp do wszystkich argumentów przekazanych do funkcji:

Listing 27. Parametr rest w praktyce

```
function foo( a, b, ...restArgs ){
  console.log('Ilość parametrów: ' + (2 + restArgs.length) );
}
foo( 10,20,30 ); // Ilość parametrów: 3
foo( 10,20 ); // Ilość parametrów: 2
foo( 10,20, 'paramter', { arg : 'val' } ); // Ilość parametrów: 4
```

Do tej pory w języku JavaScript funkcjonowała tablica argumentów funkcji `arguments`. W niej znajdowały się wszystkie argumenty przekazane do funkcji. W czym więc tkwi nowość takiego rozwiązania? Przede wszystkim `arguments` nie jest prawdziwą tablicą, a jedynie ją imituje. Obiekt `rest` przekazany do funkcji tworzy obiekt będący prawdziwą tablicą. Dzięki temu możemy na jego rzecz wywoływać funkcje takie jak `forEach`, `map`, `filter` itp. Przykładem może być przekazanie do funkcji wielu parametrów, spośród których kilka pełni rolę funkcji wywołania zwrotnego. Wykorzystując obiekt `rest`, jesteśmy w stanie wyłuskać te funkcje i wywołać je, korzystając z jednego przebiegu pętli `forEach`. Ten sam algorytm napisany z wykorzystaniem obiektu `arguments` nie zadziała poprawnie.

Listing 28. Różnice pomiędzy obiektem rest i arguments

```
function call1( arg1, arg2 ){

  console.log( 'Suma: ' + (arg1 + arg2) );
}

function call2( arg1, arg2 ){

  console.log( 'Iloczyn: ' + (arg1 * arg2) );
}
```

```
function foo( a, b, ...restArgs ){

  restArgs.forEach( rest => {
    if( typeof rest === 'function' ){
      rest.call(this, a, b );
    }
  });

}

// ponizszy kod nie zadziala
function foo2( a, b ){

  arguments.forEach( rest =>{
    if( typeof rest === 'function' ){
      rest.call(this, a, b );
    }
  });

}

foo( 2, 3, 4, call1, "Jakiś paramnetr", call2);
foo2( 5, 6, 'xx', call1, "Jakiś paramnetr", call2); //blad -
forEach nie jest funkcja
```

W powyższym przykładzie w funkcji `foo()` wykorzystaliśmy obiekt `rest` oraz znane nam już funkcje strzałkowe. Za pomocą pętli przeszliśmy po wszystkich nadmiarowych argumentach funkcji. Argumenty będące funkcjami zostały wywołane na rzecz funkcji `foo()`, z jej dwoma pierwszymi argumentami. Jest to prosty przykład demonstrujący różnicę pomiędzy obiektami `rest` a `arguments`. Co prawda w funkcji `foo2()` można skorzystać z innej pętli i osiągnąć ten sam efekt, jednak przy bardziej skomplikowanym filtrowaniu lub mapowaniu tablicy nowy zapis jest łatwiejszy.

Tworzenie obiektu `rest` w deklaracji funkcji poprzedzone jest trzykropkiem (...). Zapis ten ma też inne zastosowanie. Jeśli zostanie użyty w chwili wywołania funkcji, pełni wówczas rolę odwrotną do obiektu `rest`. Rozdziela on tablicę lub ciąg znaków na osobne argumenty funkcji. Ułatwia to proces wysyłania i odczytywania argumentów funkcji. Dla przykładu stworzymy obiekt użytkownika, który podczas inicjacji będzie wypełniał pola takie jak imię, nazwisko i wiek. Do inicjalizacji obiektu użyjemy tablicy zawierającej te dane.

Listing 29. Przekazywanie elementów tablicy do funkcji

```
var user_info = ['Jan', 'Nowak', 45];

function User(name, surname, age ){
  this.name = name;
  this.surname = surname;
  this.age = age;
}

var jan = new User( user_info[0], user_info[1], user_info[2] );
```

Wprowadzony w ECMAScript 6 nowy operator rozdzielenia upraszcza taki proces. W połączeniu z obiektem `rest` daje on bardzo szerokie pole do stworzenia prostego i wydajnego mechanizmu przekazywania parametrów. Zaprezentowany przykład zapisany z wykorzystaniem operatora `spread` oraz obiektu `rest` przedstawia się następująco:

Listing 30. Wykorzystanie operatora spread wraz z obiektem rest przy przekazywaniu argumentów do funkcji

```
var user_info = ['Jan', 'Nowak', 45, {'www': 'www.adres.pl',
'email' : 'email@domena.pl'}];

function User(name, surname, age, ...rest ){
  this.name = name;
  this.surname = surname;
  this.age = age;

  for( let key in rest[0] ){
    this[key] = rest[0][key];
  }
}

var jan = new User( ...user_info );
```

Operator `spread` ma jeszcze dwie funkcjonalności. Pierwszą z nich jest rozbicie na tablicę ciągu znaków. Po takiej operacji każdy znak znajdzie się w osobnym elemencie tablicy:

Listing 31. Rozbicie ciągu znaków

```
// listing 31
var str = "ABCDE";
var arr = [...str]; // [A], [B], [C], [D], [E]
```

Trzykropek może służyć też do łączenia ze sobą tablic:

Listing 32. Łączenie tablic z użyciem operatora…

```
var arr1 = [4, "string", 55, true];
var arr2 = [1,2,3, ...arr1];
console.log(arr2);
//[1, 2, 3, 4, "string", 55, true]

var arr3 = [...arr1, ...arr2];
console.log(arr3);
//[4, "string", 55, true, 1, 2, 3, 4, "string", 55, true]
```

Szablony

Z tej funkcjonalności ucieszą się chyba wszyscy developerzy, niezależnie od poziomu zaawansowania. Nawet osoby zaczynające programować w JavaScript spotkały się z koniecznością łączenia ciągów znakowych ze zmiennymi. Obojętnie, czy jest to zliczenie ilości elementów w drzewie DOM, czy wartości zwrócone asynchronicznie, brak mechanizmu szablonów był naprawdę uciążliwy. W międzyczasie powstało kilka naprawdę dobrych bibliotek, jak np. Handlebars.js, oferujących możliwość tworzenia szablonów w JavaScript. Sprawdzają się one znakomicie przy większych treściach i zaawansowanych operacjach na drzewie DOM.

Dotychczas praca z ciągami znaków, które musieliśmy połączyć z danymi generowanymi dynamicznie, polegała na ręcznej konkatenacji znaków z wartościami:

Listing 33. Przykład łączenia ciągów znaków z wartościami dynamicznymi i wyrażeniami

```
function showPerson(imie, nazwisko, wiek){
  console.log('Osoba ' + imie + ' ' + nazwisko + ' ma ' + wiek +
  ' lat, czyli ' + ( wiek * 12 ) + ' miesięcy.' );
}
```

W najnowszej wersji standardu został wprowadzony mechanizm szablonów. Pozwala on na wstawienie zmiennych (i nie tylko) wewnątrz ciągu znakowego, bez potrzeby jego ręcznego łączenia. Zawartość szablonu mieści się w odwróconych apostrofach, a wstawiane wartości umieszczamy pomiędzy nawiasami klamrowymi poprzedzonymi znakiem dolara ${....}. W związku z powyższym kod funkcji z Listingu 26 będzie przedstawiał się następująco:

Listing 34. Przykład zastosowania szablonów

```
function showPerson( imie, nazwisko, wiek){
  console.log(`Osoba ${imie} ${nazwisko} ma ${wiek} lat, czyli
  ${wiek * 12} miesięcy`);
}
```

Tak stworzony kod jest zdecydowanie bardziej czytelny. Co więcej, wartości podane w nawiasach klamrowych nie ograniczają się jedynie do wartości zmiennej. Nic nie stoi na przeszkodzie, żeby, jak w przykładzie powyżej, w nawiasach znajdowało się wyrażenie, którego wynik będzie umieszczony w szablonie. Wyrażenie może mieć postać prostego lub złożonego rachunku arytmetycznego:

Listing 35. Przykład złożonego wyrażenia arytmetycznego w szablonach

```
function delta(a ,b, c){
console.log( `Delta wynosi: ${Math.power(b,2) - ( 4 * a* c)}`);
}
```

Mechanizm szablonów jest bardzo elastyczny, oprócz zmiennych typu prostego oraz wyrażeń bez problemu radzi sobie z ciągami znaków, obiektami, funkcjami, a nawet instrukcjami warunkowymi. Jeśli istnieje konieczność użycia odwróconego apostrofa w treści szablonu, należy poprzedzić go ukośnikiem \.

Listing 36. Przykłady użycia szablonów z różnymi typami danych

```
var user = {
  name : 'Jan',
  surname : 'Kowalski',
  age : 25,
  role : 'admin'
};

var lastLogin = function(){
  'use strict';
  let d = new Date();
return d.toDateString();
}

function userInfo(user){
  console.log(`Witaj to: ${user.name} ${user.surname},
Twoje logowanie nastąpiło ${lastLogin()}
Aktualnie masz ${ user.age + "lat" }
Typ twojego konta to: ${ user.role === 'admin' ? "administrator"
: "użytkownik"}
  A to są znaki specjalne: \` \${ }`);
}
userInfo(user);
```

Powyższy listing zawiera kilka sposobów definiowania wartości wyrażenia znajdującego się w szablonie. Jak można zauważyć, szablon zajmuje kilka linii kodu. W tym przykładzie ma to duże znaczenie, ponieważ szablony nie pomijają białych znaków. Oznacza to, że ciąg znaków przeniesiony do nowej linii, w rzeczywistości również będzie wyświetlony poniżej. To samo dotyczy spacji oraz wszystkich innych białych znaków, jakie mogą pojawić się w tekście. Na przykład ostatni wiersz jest przesunięty o 10 spacji. Tak samo sformatowany i wyświetlony zostanie w konsoli oraz w przeglądarce internetowej.

Istnieje też bardziej zaawansowany sposób tworzenia szablonów. Można stworzyć specjalnie przygotowaną funkcję oznaczającą. Pierwszym parametrem takiej funkcji jest tablica ciągów znaków, a kolejnymi są przekazane parametry. Wywołanie takiej funkcji następuje poprzez podanie jej nazwy przed znakiem `, czyli początkiem szablonu.

Listing 37. Funkcja obsługi szablonów

```
function tag(strings, ...args){
  console.log( strings );
  //[ "Nazywam się ", " i mam ", " lat, czyli ", " miesięcy." ]

  console.log( args );
  //[ "Jan Kowalski", "60", 720 ]

  return `Jestem ${ args[0].toUpperCase() } i mam ${args[1]} lat
  (${args[2]} mc-y)`;
}

var osoba = {
  nazwa : "Jan Kowalski",
  wiek : '60'
}
tag`Nazywam się ${osoba.nazwa} i mam ${osoba.wiek} lat, czyli
${osoba.wiek * 12} miesięcy.`;
```

Tablica ciągów znaków przekazana w pierwszym parametrze powstaje z podzielenia treści szablonu na mniejsze części. Elementem dzielącym jest wyrażenie zawarte w znaczniku ${...}. Proces ten bardzo przypomina rezultat działania metody `split()`. Drugim argumentem jest obiekt `rest` zawierający wszystkie parametry przekazane do szablonu. Jeśli parametrem jest wyrażenie, wtedy w tablicy znajdzie się jego wynik, tak samo w przypadku użycia funkcji lub instrukcji warunkowej.

Istotne jest, aby funkcja tagująca zwracała wartość, gdyż to właśnie ona będzie ostateczną postacią szablonu. I tak w przykładzie powyżej zmieniliśmy treść szablonu, który po wykonaniu funkcji będzie miał postać: *"Jestem JAN KOWALSKI i mam 60 lat (720 mc-y)"*.

Obiekt Promise

Standard ECMAScript 2015 wprowadził natywną obsługę mechanizmu `Promise`. Do tej pory wykorzystanie tego mechanizmu w JavaScript odbywało się najczęściej za pomocą bibliotek takich jak Q, when, WinJS, RSVP.js czy też jQuery (począwszy od wersji 1.5). Jest to narzędzie do pracy z funkcjami asynchronicznymi, które chroni te funkcje przed ingerencją zewnętrznego kodu podczas ich wykonywania. Celem wprowadzenia w zagadnienia asynchroniczne przyjrzyjmy się Listingowi 38. Zawiera on przykład funkcji asynchronicznej:

Listing 38. Wykorzystanie funkcji asynchronicznej

```
var message = "Hello ";

function _get( url ){
  var req = new XMLHttpRequest();

  req.open('GET', url, true);

  req.onreadystatechange = function () {
    if (req.readyState == 4) {
      if(req.status == 200){
        message += req.responseText; //Hello John
      }
      else{
        message = "Error!";
      }
    }
  };
  req.send(null);
console.log( message ); // "Hello "
}

_get('/get-name');
```

Jak wiadomo, wywołanie asynchroniczne nie blokuje kolejki wywołań. Oznacza to, że w powyższym przykładzie wyświetlenie w konsoli wartości zmiennej `message` będzie miało postać: *"Hello "*. Dzieje się tak, ponieważ przeglądarka nie czeka z wykonywaniem kodu do czasu otrzymania odpowiedzi z adresu, pod który wysłała żądanie. Rozwiązaniem tego problemu jest wywołanie procedur w chwili otrzymania odpowiedzi asynchronicznej. W tym miejscu pojawia się możliwość skorzystania z mechanizmu `Promise`.

Czym tak naprawdę jest mechanizm `Promise`? Jest to obiekt, którego konstruktor jako argument przyjmuje dwuargumentową funkcję. Jej argumentami są funkcje wywołania zwrotnego, uruchamiane w chwili pomyślnego zakończenia procesu asynchronicznego, jak i w chwili wystąpienia błędu. Brzmi to nieco skomplikowanie, ale w rzeczywistości takie nie jest. Przejdźmy do przykładu. Na początku stworzymy nowy obiekt `Promise`:

Listing 39. Nowy obiekt Promise

```
var promise = new Promise(
  function (resolve, reject) {
    ...
    if ( success ) {
      resolve(value);
    } else {
      reject(reason);
    }
  });
```

W tym momencie należy wspomnieć o trzech stanach, w jakich może znajdować się obiekt `Promise`:

- Oczekiwanie – stan, podczas którego wykonywane są operacje, np. wysłanie asynchronicznego żądania.

- Spełnienie – stan, w którym funkcja z sukcesem zakończyła swoje działanie, np. otrzymanie poprawnego obiektu JSON w odpowiedzi na żądanie.
- Odrzucenie – moment pojawienia się błędu, np. komunikat 404, lub niepoprawnie sformatowany obiekt JSON.

W przykładzie z Listingu 39 obiekt znajduje się w fazie oczekiwania. Po ukończeniu zadań następuje sprawdzenie wartości zmiennej `success`. Jeśli ma wartość `true`, wywoływana jest funkcja `resolve()`, a obiekt przechodzi w stan spełnienia (Fulfillment). W przeciwnym wypadku wywoływana jest funkcja `reject()`, a obiekt znajduje się w stanie odrzucenia (Rejected).

Mając już wiedzę teoretyczną, stworzymy mechanizm asynchroniczny oparty na przykładzie z Listingu 38.

Listing 40. Obsługa żądania asynchronicznego za pomocą mechanizmu Promise

```
var message = "Hello ";

function onSuccess( msg ){
  console.log( message + '' + msg );
};

function onError( err ){
  console.log( err );
};

function _get( url ){
  return new Promise(function( resolve, reject ){

    var req = new XMLHttpRequest();

    req.open('GET', url, true);
    req.onreadystatechange = function () {
      if (req.readyState == 4) {
        if(req.status == 200){
          resolve(req.responseText)
        }
        else{
          reject( new Error(this.statusText) );
        }
      }
    };

    req.onerror = function () {
      reject( new Error('XMLHttpRequest Error: '+ this.statusText)
      );
    };

    req.open('GET', url);
    req.send();

  });
}

_get('http://127.0.0.1/es6/file.txt').then(onSuccess, onError );
```

Jak widać, zmianie uległa funkcja `_get()`. Od teraz zwraca ona obiekt typu `Promise`, który wykonuje operację asynchroniczną. Wywołanie funkcji `_get()` z podanym adresem jako parametrem powoduje rozpoczęcie działania asynchronicznego. Zwrócony obiekt przekazywany jest do funkcji obsługi `then()` przyjmującej jako parametry dwie funkcje, wywoływane odpowiednio w chwili powodzenia operacji, jak i błędu. Oba te parametry są opcjonalne, przy czym należy pamiętać, że kiedy chcemy pominąć funkcję stanu spełnienia, a jedynie użyć funkcji odrzucenia, należy zastąpić ją wartością `null`. Istnieje też inny sposób wywoływania funkcji zwrotnych. Do tego celu możemy użyć dwóch funkcji obiektu `Promise`, tj. `then()` oraz `catch()`. W takiej sytuacji w funkcji `then()` pomijamy parametr wykonywany w momencie pojawienia się błędu i przenosimy go do funkcji `catch()`. Przykłady wywołania zamieszczono na Listingu 41.

Listing 41. Różne sposoby pracy z obiektem Promise

```
_get('http://127.0.0.1/es6/file.txt').then(onSuccess, onError );
_get('http://127.0.0.1/es6/file.txt').then(onSuccess);
_get('http://127.0.0.1/es6/file.txt').then(null, onError );
_get('http://127.0.0.1/es6/file.txt').then( onSuccess ).catch( onError );
```

Pracując z obiektami `Promise`, bardzo często będziemy używać ich do obsługi akcji wywołań asynchronicznych. Zdarza się, że zachodzi konieczność wywołania procedur w ściśle określonej kolejności. Można to osiągnąć poprzez zastosowanie łańcuchów wywołań. W poniższym przykładzie wykorzystamy funkcję `_get()` z Listingu 40. Tym razem będziemy pracować na łańcuchach wywołań.

Listing 42. Łańcuch wywołań then()

```
_get('http://127.0.0.1/es6/file.txt')
  .then(function( msg ){
    msg = msg + ' ' + '. Nice to meet you!';
    return msg;
  })
  .then(onSuccess)
  .catch(onError);
```

Tworzenie łańcucha wywołań daje gwarancję wykonania poprzedniej operacji. W przykładzie na Listingu 42 wywołanie `.then(onSuccess)` nastąpi dopiero po zakończeniu żądania asynchronicznego z funkcji `_get()` oraz po wykonaniu operacji zawartych w pierwszym wywołaniu `then()`. Mechanizm ten gwarantuje kolejność wywoływania operacji. Ma to szczególne znaczenie w sytuacji następujących po sobie wywołań asynchronicznych, w których każde kolejne wywołanie jest uzależnione od wyniku poprzedniego. W chwili wystąpienia błędu następuje wywołanie pierwszego napotkanego w kolejce `catch()`. Dobrze zaprojektowany łańcuch wywołań pozwala na obsłużenie błędu poprzez np. wstawienie wartości zastępczej. Korzystając ze znanej nam już funkcji `_get()`, wyślemy żądanie nieistniejącego pliku. Błąd, jaki się pojawi, obsłużymy za pomocą prostego łańcucha wywołań `then()` oraz `catch()`. W tym celu zmodyfikujemy nieco funkcję obsługi błędu oraz dodamy nową, wywoływaną po wystąpieniu błędu. Sama funkcja `_get()` pozostanie bez zmian.

Listing 43. Łańcuch wywołań then() oraz catch()

```
function onSuccess( msg ){
  console.log( 'Hello ' + msg );
};

function onSuccessAfterError( msg ){
  console.log( msg );
};

function onError( err ){
  console.log( err );
  return 'Connection error, please try again letter';
};

_get('http://127.0.0.1/es6/bad_fle.txt')
  .then(onSuccess)
  .catch(onError)
  .then(onSuccessAfterError);
```

Generatory

Generatory stanowią specyficzny rodzaj funkcji, umożliwiają bowiem zatrzymanie wykonywania działań funkcji oraz jej wznowienie od miejsca, w którym została wstrzymana, zachowując przy tym wewnętrzne wartości danych. Deklaracja generatora jest zbliżona do typowej deklaracji funkcji, jednak w tym przypadku po słowie kluczowym `function` występuje znak gwiazdki *, a w ciele funkcji występuje słowo `yield`. Na Listingu 44 przedstawiono bardzo prosty generator zwracający kolejne wartości ciągu Fibonacciego.

Listing 44. Obliczanie ciągu Fibonacciego z użyciem generatorów

```
function* fibonacci() {
  var pre = 0, cur = 1;
  while ( true ) {
    [ pre, cur ] = [ cur, pre + cur ]
    yield cur;
  }
}

var fib = fibonacci();
```

```
console.log(fib.next().value); //1
console.log(fib.next().value); //2
console.log(fib.next().value); //3
console.log(fib.next().value); //5
```

Przejdźmy więc do analizy powyższego kodu. Znak * informuje nas, że mamy do czynienia z generatorem, a nie zwykłą funkcją. Następnie definiujemy zmienne pomocnicze, przechowujące dwie wartości niezbędne do obliczenia kolejnej wartości ciągu, oraz tworzymy nieskończoną pętlę `while`. Pierwsza linijka w pętli wydaje się niezrozumiała. Skorzystaliśmy tutaj z nowego mechanizmu definiowania zmiennych w nawiasach prostokątnych. Zapis ten jest równoznaczny z zapisem `var tmp = pre; pre = cur; cur = cur + tmp`. Jest to kolejna nowość w ES6 ułatwiająca życie programistom. Kwintesencją generatora jest użycie `yield cur;`. Powoduje to wstrzymanie wykonywania funkcji i zwrócenie aktualnej wartości zmiennej `cur`. Różnica między `yield` a `return` jest taka, że `return` kończy działanie funkcji. Ponowne jej wywołanie będzie skutkowało rozpoczęciem wykonywania wszystkich operacji w niej od początku. W generatorze natomiast wartości zmiennych zostają zachowane. W naszym przykładzie generator przypisujemy do zmiennej fib i na jej rzecz wykonujemy operacje. Wewnętrzne zmienne `prev` i `cur` zachowują swoją wartość przy kolejnych wywołaniach. Dostęp do zwracanej wartości otrzymujemy poprzez zapis `fib.next().value`. Z każdym następnym odwołaniem generator wznawia działanie od pierwszej instrukcji występującej po instrukcji `yield`. W Listingu 44 będzie to zakończenie pętli i ponowne sprawdzenie warunku.

Przeanalizujmy teraz, w jaki sposób używać generatorów. W Listingu 44 do konsoli wrzucaliśmy wynik działania funkcji `next()`. Funkcja ta wywołana na rzecz obiektu generatora powoduje rozpoczęcie lub wznowienie jego działania. Jako wynik zwraca obiekt zawierający dwa pola: `value` oraz done. Obiekt ten, jak już wiemy, jest zwracany przez słowo kluczowe `yield`. Pole done jest typu boolowskiego i określa ono, czy generator zakończył swoją pracę. Zakończenie pracy generatora będzie skutkowało zwracaniem wartości `true` w polu done oraz undefined jako wartość. Taką sytuację przedstawia Listing 45.

Listing 45. Praca z generatorem kończącym

```
function* generator(){

  yield 1;
  yield 2;
  yield 3;

  return 4;
}
var myGen = generator();

console.log( myGen.next() ); // Object { value=1, done=false}
console.log( myGen.next() ); // Object { value=2, done=false}
console.log( myGen.next() ); // Object { value=3, done=false}
console.log( myGen.next() ); // Object { value=4, done=true}
console.log( myGen.next() ); // Object { done=true,
value=undefined}
```

Za pomocą funkcji `next()` można również przekazywać parametry do generatora. Jest to mechanizm wielokrotnego wejścia. Pozwala on na przekazanie parametrów, które mogą nie być jeszcze zdefiniowane podczas pierwszego wywołania. Poniższy przykład nie oferuje zbyt użytecznej funkcjonalności, ale znakomicie sprawdza się jako przykład działania generatora, który na przemian zwraca i przyjmuje określone wartości.

Listing 46. Generator zwracający i przyjmujący wartości

```
function* power(){
  var v1 = yield "a";
  var v2 = yield "b";

  return Math.pow(v1, v2);
}

var pow = power();

console.log( pow.next() ); //Object { value="a", done=false}
console.log( pow.next(3) ); //Object { value="b", done=false}
console.log( pow.next(4) ); //Object { value=81, done=true}
console.log( pow.next() ); //Object { done=true, value=undefined}
```

Przesłane do generatora wartości zastępują w nim instrukcję `yield`... . W rzeczywistości więc deklaracje zmiennych w generatorze mają postać `v1 = 3;, v2 = 4;`. Ciekawostką jest fakt, że według większości materiałów dostępnych w sieci ostatnie wywołanie `pow.next()` powinno spowodować wystąpienie błędu. Jednak zarówno przeglądarka Firefox oraz Chrome, jak i serwer Node każde kolejne wywołanie funkcji `next()` interpretują w odmienny sposób. Zamiast generować błąd, zwracają poprawny obiekt, w którym pole `value` ma wartość undefined, a pole done – `true`.

Wiemy już, jak konstruować i obsługiwać generatory. Przyszła pora na bardziej zaawansowane użycie tego mechanizmu. Do tej pory za pomocą `yield` zwracaliśmy wartości typów podstawowych. Nic nie stoi na przeszkodzie, aby zamiast określonej wartości wywołać funkcję, nawet funkcję obsługującą wywołania asynchroniczne. Co więcej, mechanizm generatora wykryje taki proces i wstrzyma się z dalszym wykonywaniem do czasu otrzymania odpowiedzi. Oznacza to, że żądania asynchroniczne zaczynają mieć postać kodu synchronicznego, przez co odzyskujemy kontrolę nad całym procesem. Używając poznany wcześniej obiekt `Promise`, można stworzyć przejrzystą i w pełni funkcjonalną strukturę obsługującą metody asynchroniczne. Taka struktura została przedstawiona na Listingu 47. Oprócz mechanizmu generatorów i obiektu `Promise` w przykładzie tym wykorzystano nowe elementy standardu ECMAScript 6, takie jak zakres blokowy czy funkcje strzałkowe. Kod ten został przetestowany w przeglądarce Firefox 40.0.3 oraz w serwerze Node v4.1

Listing 47. Mechanizm obsługi żądań asynchronicznych wykorzystujący generator i obiekty Promise

```
"use strict";

// Generator
function* myGen(){
  let v1 = yield request('first');
  let v2 = yield request('second');
  let v3 = yield request('third');

  return [v1, v2, v3];
}

//Funkcja wykonująca żądania asynchroniczne
function request( txt ){
  return new Promise( function(resolve, reject){
    setTimeout( () => {
      console.log(txt);
      resolve( txt );
    } , 1000);
  });
}

// Funkcja uruchamiająca generator
function runner(generator){
  let gen = generator();
  return new Promise( (resolve, reject) => {

    let loop = (val) => {
      let result;
      try {
        result = gen.next(val);
      } catch (err) {
        reject(err);
      }

      if( result.done){
        resolve( result.value );
      } else {
        result.value.then( (val) => loop(val) )
          .catch( ( err ) => reject(err) );
      }
    }
    loop();
  });
}

// uruchomienie mechanizmu
runner( myGen )
  .then( msg => console.log(msg) )
```

```
  .catch( err => console.log(err) );

//first
//second
//third
//["first", "second", "third"]
```

W powyższym przykładzie wyjątkowo odeszliśmy od przyjętego założenia krótkich listingów, jednak w tym przypadku nie było innego sposobu, aby zaprezentować cały mechanizm. Ale po kolei. Przejdźmy do analizy powyższego kodu. Najpierw stworzyliśmy nowy generator `myGen()`. Posiada on trzy punkty wstrzymania zwracające wynik funkcji `request()`, a na końcu zwraca wartości zmiennych `v1`, `v2`, `v3` w postaci tablicy. Następnie definiujemy funkcję `request()`. Jest to funkcja wykonująca żądanie asynchroniczne. W naszym przykładzie uruchamiana jest metoda `setTimeout()`. Przy czym nie ma znaczenia, jakiej funkcji użyjemy w tym przykładzie. Równie dobrze moglibyśmy wysłać żądanie AJAXowe lub jakiekolwiek inne żądanie asynchroniczne. Ważne jest, że zostało ono zdefiniowane wewnątrz obiektu `Promise`, który jest zwracany przez funkcję `request()`. Dla zachowania przejrzystości pominięto obsługę błędów. W rzeczywistości należy zadbać, aby obiekt `Promise` wywoływał metodę `reject()` w chwili pojawienia się błędów, jak np. błąd w dostępie do zasobów itp.

Kolejnym krokiem jest stworzenie funkcji uruchamiającej generator. W Listingu 47 nosi ona nazwę `runner()`. Jako argument przyjmuje ona referencję do generatora. Działanie funkcji rozpoczyna się od stworzenia obiektu generatora i zwrócenia obiektu `Promise`. Wewnątrz tego obiektu definiujemy strukturę przypominającą pętlę. Do zmiennej `loop` przypisano funkcję uruchamiającą generator. Umieszczenie tej instrukcji w bloku `try... catch` daje nam kontrolę nad wystąpieniem błędów. Jeśli takie by się pojawiły, obiekt `Promise` wejdzie w stan. Jeśli generator wykona poprawnie swoje zadanie, sprawdzamy wartość logiczną pola `result.done`. Logiczna prawda (`true`) oznacza, że generator skończył swoje działanie. Możemy więc wywołać funkcję `resolve()` i zmienić stan zwracanego obiektu `Promise` na spełniony. W sytuacji, kiedy generator nie skończył działać rekurencyjnie, wywołujemy funkcję `loop`. Jednak to wywołanie musi odbywać się w ściśle określony sposób. Zmienna `result` zawiera obiekt `Promise` zwrócony przez generator (stworzony w funkcji `request()`). Żeby zachować asynchroniczność, nowe wywołanie `loop` musi nastąpić wewnątrz metody `then()` zwracanego obiektu. Co więcej, teraz parametr `val` będzie miał już swoją wartość. Zostanie on przekazany do generatora i przypisany zmiennej `v1`. Taka sytuacja powtarza się do chwili, kiedy generator nie zakończy swojego działania.

Aby cały mechanizm zaczął funkcjonować, potrzeba go jeszcze uruchomić. Wywołujemy więc funkcję `runner()`, przekazując do niej referencję do generatora. Wiedząc, że pracujemy na mechanizmie Promise, obsłużymy je za pomocą metod `then()` oraz `catch()`.

Cały ten przykład wymaga głębszego przeanalizowania i zapoznania się z nową formą pracy z wywołaniami asynchronicznymi. Przykład ten można rozbudowywać i łączyć w łańcuchy z innymi wywołaniami. Co więcej, pozwala on na pracę asynchroniczną w podobny sposób jak ze standardowymi funkcjami synchronicznymi.

Klasy i obiekty

Ostatnim zagadnieniem omawianym w tym artykule będą nowe metody tworzenia klas i obiektów. Nowa składnia jest tzw. lukrem składniowym, czyli sposobem bardziej intuicyjnego i łatwiejszego zapisu dotychczas stosowanych mechanizmów. Zachowując kompatybilność wsteczną, klasy i obiekty w ECMAScript 6 nadal opierają się na dziedziczeniu prototypowym. Nowa składnia przybliża jednak język JavaScript do standardów programowania obiektowego znanych z innych języków programowania. Rozwiązanie może nie idealne, ale oparte na rozsądnym kompromisie pomiędzy zgodnością ze starszymi wersjami standardu a nowoczesną składnią obiektową. Warto wspomnieć, że w chwili pisania niniejszego artykułu wsparcie dla nowej składni obiektowej jest praktycznie znikome. Decydując się na użycie nowej składni, wręcz koniecznym jest skorzystanie z narzędzi kompilujących kod do starszych wersji standardu.

Poniższy przykład przedstawia dotychczasowy sposób tworzenia obiektów wraz z mechanizmem dziedziczenia.

Listing 48. Przykład dziedziczenia prototypowego

```
function Family( name ){
  this.name = name;
}

Family.prototype.myself = function(){
  console.log('I am ' + this.name );
}

function Father(){
  Family.call(this, 'Father');
}
Father.prototype = Object.create( Family.prototype );

var father = new Father();
father.myself(); // I am Father
```

Dla osób mających doświadczenie w programowaniu obiektowym w JavaScript ten kod jest w pełni zrozumiały. Developerzy na co dzień programujący w innych językach z pewnością poświęcą dłuższą chwilę na jego analizę. Nowy sposób zapisu wprowadza słowa kluczowe, takie jak `class`, `extends` oraz `super`. Ich funkcjonalność jest dokładnie taka sama jak w przypadku składni z takich języków jak chociażby PHP czy Java. Najprościej będzie przedstawić to na przykładzie. Stwórzmy więc kod, którego funkcjonalność będzie identyczna z tym z Listingu 48. Teraz jednak skorzystamy z nowej składni.

Listing 49. Przykład użycia nowej składni

```
class Family {

  constructor( name) {
    this.name = name;
  }

  myself(){
    console.log(`I am + ${this.name}` );
  }
}

class Father extends Family {
  constructor(){
    super('Father');
  }
}

var father = new Father();
console.log( father.myself() );
```

Nowy zapis jest zdecydowanie łatwiejszy do zrozumienia. Konstruowanie klas odbywa się w podobny sposób jak ma to miejsce w innych językach programowania. Rolę konstruktora obiektu pełni metoda `constructor()`. Proces dziedziczenia odbywa się przy użyciu słowa kluczowego `extends` umieszczonego pomiędzy nazwami klas dziedziczącej i bazowej. Dostęp do konstruktora klasy bazowej odbywa się poprzez metodę `super()`. Nowa składnia narzuca również zachowanie kolejności. Klasa musi być najpierw zdefiniowana, zanim zostanie utworzona jej instancja. Przy programowaniu obiektowym z użyciem funkcji zachowanie kolejności nie jest konieczne:

Listing 50. Różnice w definiowaniu klas z użyciem nowego i starego zapisu

```
var k1 = new Klasa1()
var k2 = new Klasa2() // błąd

function Klasa1(){}

class Klasa2{
  constructor(){}
}
```

Nowa składnia oferuje również bardziej intuicyjny zapis metod statycznych, będących w rzeczywistości funkcjami składowymi niezależnymi od kontekstu, w jakim zostały zdefiniowane. Oto przykład zdefiniowania metody statycznej w ES6 wraz z przykładem wykorzystującym dotychczasowe sposoby tworzenia takich metod. W tym celu rozbudujemy trochę naszą klasę `Family`.

Listing 51. Metody statyczne z użyciem nowej i dotychczasowej składni

```
class Family {

  constructor( name) {
    this.name = name;
  }

  myself(){
    console.log(`I am + ${this.name}` );
  }

  static info(){
    console.log('Typical family information class');
  }
}

Family.info(); //Typical family information class
Family.myself() // error

// Stary sposób
function Family( name ){
  this.name = name;
}

Family.prototype.myself = function(){
  console.log('I am ' + this.name );
}

Family.info = function(){
  console.log('Typical family information class');
}

Family.info(); //Typical family information class
Family.myself();// error
```

Warto pamiętać, że metody statyczne podlegają dziedziczeniu, w przeciwieństwie do metod będących zdefiniowanymi poza prototypem klasy bazowej.

Listing 52. Dziedziczenie metod statycznych

```
function Father(){
  Family.call(this, 'Father');
}
Father.prototype = Object.create( Family.prototype );

Father.info(); //error

class Father extends Family {
  constructor(){
    super();
  }
}
Father.info(); //Typical family information class
```

Uproszczenia składni doczekały się również literały obiektowe. Wciąż obowiązujący zapis bazujący na JSON został zmodyfikowany. Od teraz przypisując funkcję do pola obiektu, możemy zrezygnować z zapisu bazującego na parach klucz : wartość, co zostało przedstawione na Listingu 53.

Listing 53. Uproszczona składnia literałów obiektowych

```
var literal = {
  foo : function(){
    console.log('Correct function foo()');
  },

  foo2(){
    console.log('Correct function foo2()');
  }
};

literal.foo();
literal.foo2();
```

PODSUMOWANIE

ECMAScript 6 jest przyszłością języka JavaScript. Co prawda upłynie jeszcze sporo czasu, zanim przeglądarki internetowe zaczną w szeroki sposób wspierać nowe mechanizmy. Można spodziewać się, że rozwiązania serwerowe takie jak Node będę w stanie szybciej zaimplementować obsługę standardu. Biorąc pod uwagę proces aktualizacji tych narzędzi, ES6 będzie szybciej stosowane w aplikacjach backendowych. Nie oznacza to, że frontendowcy będą spokojnie czekać na rozpowszechnienie się nowego standardu. Wręcz przeciwnie. Stosując kompilatory do ECMAScript 5, będą powstawać nowe wersje bibliotek, oparte o nowszą wersję. Przykładem może być biblioteka Bootstrap 4, której komponenty skryptowe JS zostały w całości napisane w ECMAScript 6.

Analizując nowy standard, nie sposób odnieść wrażenie, że pewne rozwiązania zostały wprowadzone za późno. Programiści przyzwyczaili się do specyficznych właściwości języka, a wprowadzane zmiany są ukierunkowane raczej na uporządkowanie składni. Takimi przykładami są mechanizmy definiowania zmiennych za pomocą `let` czy leksykalny `this` w funkcjach strzałkowych. Kontrowersyjne jest również wprowadzenie nowej metody deklaracji klas i pracy z obiektami. Standard nie wprowadza nowych rozwiązań, a bazuje na znanym mechanizmie prototypowym.

Trzeba jednak uczciwie powiedzieć, że nowy standard jest dużym krokiem w przyszłość. Pozwala zrezygnować z dołączania do projektu zewnętrznych bibliotek, oferując natywne wsparcie dla rozwiązań takich jak szablony, obiekty `Promise` czy mechanizm modułów. Wprowadzenie generatorów, deklaracje stałych, argumenty domyślne, operatory `...rest` otwierają nowe drogi rozwoju aplikacji opartych o JavaScript. Pierwsza styczność ze wspomnianymi zapisami może przyprawić o ból głowy.

Przy ocenie należy też wziąć pod uwagę, że na tym wydaniu nie skończy się rozwój standardu. Szósta wersja ma przygotować developerów i środowiska do nowego stylu programowania aplikacji webowych. Wedle zapowiedzi komitetu ECMA na siódmą wersję standardu przyjdzie nam czekać krócej niż na szóstą. Lista życzeń z tygodnia na tydzień się wydłuża. Aktualne wydanie jest ważnym etapem przejściowym. Powołując się na słowa Jafar'a Husain'a, ECMAScript 6 jest bliższy idei JavaScript niż jego dotychczasowe implementacje. Można więc przypuszczać, że rozwój, po burzliwych czasach, wkroczył w fazę dobrze przemyślanego planu wydawniczego. Jak będzie naprawdę, zweryfikuje życie i developerzy piszący kod w nowych standardach.

Grzegorz Dąbrowski

https://www.linkedin.com/in/dabrowskigrzegorz

Z technologiami webowymi związany od ponad 15 lat. Obecnie pracuje jako Front-end Developer w firmie Nomino sp. z.o.o., gdzie współtworzy aplikacje typu e-Commerce dla zagranicznych klientów.