JavaScript

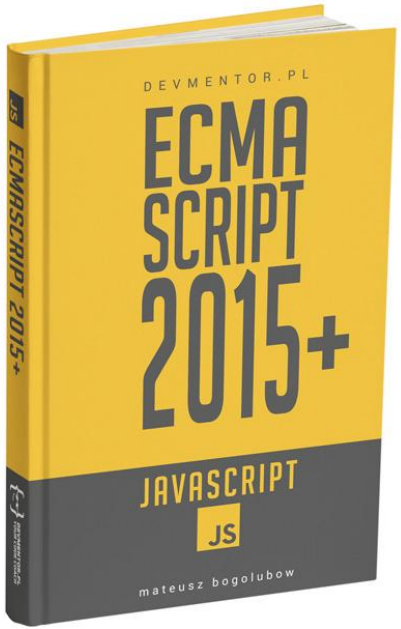
ECMAScript 2015+

Standard ECMAScript z 2015 r. (ES6) wniósł w świat JavaScript powiew świeżości.

Pojawiło się wiele przydatnych rozwiązań, które zwiększają wygodę pracy, jej efektywność i umożliwiają korzystanie z najnowszych wersji bibliotek, np. React.

Poznaj najważniejsze zmiany w języku JS od 2015 roku!

Korzystaj z webpacka - pisz kod zgodny z najnowszymi standardami i wspierany przez starsze przeglądarki.



Jak korzystać z materiałów?

Przedstawiony materiał to jedna, przemyślana, **spójna całość**. Dlatego nie zalecam skakania między stronami, nawet jeśli uważasz, że dany temat jest dla Ciebie jasny.

Zawsze znajdzie się jedna, drobna informacja, która może zmienić Twoją perspektywę, lub której zabraknie w najważniejszym momencie, tj. na rozmowie rekrutacyjnej.

Czytając daną stronę, **zawsze przepisuj przedstawiony kod** (nie kopiuj!) i testuj go w różnych konfiguracjach.

Nie bój się czegoś zmienić, dodać lub odjąć. Zamiast zastanawiać się “co by było gdyby”, po prostu to sprawdź!

Programowanie ma tą piękną cechę, że zawsze możemy cofnąć (`ctrl+z`) nasze zmiany bez większego wysiłku!

W materiałach będą pojawiać się odnośniki do **tematów pokrewnych lub rozszerzających** daną informację.

Zapoznaj się z nimi od razu. Następnie wróć do nich jeszcze raz po zakończeniu całego rozdziału.

Jeśli masz ograniczoną ilość czasu, to nie staraj się pogłębiać wiedzy z dodatkowych źródeł. Wystarczy, że się z nimi zapoznasz i możesz iść dalej.

Pamiętaj, że **temat główny** omawiany w materiale **jest niezbędny**. Tematy rozszerzające zagadnienie to dodatek, który warto znać, jeśli ma się na to czas.

Gdy czegoś nie rozumiesz, staraj się znaleźć dodatkowe informacje na ten temat. Czasami też warto wrócić do problematycznych zagadnień po przerwie.

Wyszukiwanie informacji jest częścią pracy programisty, dlatego już teraz zacznij rozwijać tę umiejętność. Samodzielne rozwiązywanie problemów jest **bardzo doceniane** w świecie IT.

Zadania do wykonania

Pod poniższym adresem znajdziesz zadania,
które obejmują materiał prezentowany w niniejszym ebooku.

https://github.com/devmentor-pl/practice-js-es2015plus

01. Wprowadzenie

ECMAScript (ES) to standard języka, który implementują producenci przeglądarek.

Jest to pewna koncepcja wdrażana do przeglądarek zgodnie z propozycją w specyfikacji. Czasem jest wdrażana trochę inaczej niż ta specyfikacja sugeruje, a czasami wogóle.

Stąd odwieczny problem: “u mnie działa”.

Aby sprawdzić, jak sobie radzą przeglądarki z obsługą danego rozwiązania, możemy posiłkować się tą tabelą lub serwisem caniuse.com.

Ecma International (organizacja standaryzująca systemy informatyczne) od 2015 r. publikuje co rok nową wersją standardu języka JavaScript pod nazwą ES2015 (ES6), ES2016 (ES7) itd.

Prawdziwy rozwój JS rozpoczął się w 2015 r., kiedy pojawiło się wiele nowości. Przeczytasz o nich w tym materiale.

Kolejne lata to mniejsze, lecz stabilne aktualizacje, które pozwalają na ciągły rozwój jednego z najpopularniejszych języków programowania.

Nie wszystkie nowe rozwiązania są implementowane do przeglądarek, dlatego będziemy potrzebować narzędzia, które pozwoli nie przejmować się wsparciem przez przeglądarki.

- 1997: ES1
- 1998: ES2
- 1999: ES3
- ????: Publikacja ES4 została wstrzymana z powodu braku wspólnego stanowiska członków zespołu TC39
- 2009: ES5 (rozszerzenie ES3)
- 2015: ES6 (rewolucja)
- 2016: ES7
- 2017: ES8

itd.

02. Webpack

Webpack to narzędzie, który działa na plikach zdefiniowanych na wejściu (zaraz to wyjaśnimy). Wykonuje na nich operacje i zwraca jako nowy plik (lub pliki) w nowej lokalizacji.

Jest on określany mianem “module budler” i pozwala zautomatyzować wiele procesów przeprowadzanych na naszym kodzie źródłowym (i nie tylko).

My skupimy się głównie na plikach JS, które będą transpilowane.

Transpilację należy rozumieć jako zmianę kodu napisanego przez nas zgodnie z nowym standardem (np. ES6) na kod starszy (np. ES5), ale wspierany przez większość używanych przeglądarek internetowych.

Do instalacji webpacka wykorzystamy menadżer pakietów JavaScript - npm - dlatego powinieneś mieć zainstalowany Node.js w wersji co najmniej 12.

Sprawdź wersję w terminalu: `node -v`.

#01 Podstawowa konfiguracja

02. Webpack

Konfigurację zaczniemy od utworzenia katalogu, w którym będziemy opracowywać nasz projekt. Powiedzmy, że będzie to `webpack-helloworld`.

Następnie uruchomimy terminal, w którym lokalizacja wskazuje na ten właśnie katalog.

Teraz możemy utworzyć plik, w którym będziemy przechowywać informacje o naszym projekcie, tj. `package.json`.

Aby to zrobić, wystarczy w terminalu wpisać: `npm init -y`.

To utworzy wspomniany plik z domyślnymi wartościami.

Od teraz będziemy tam przechowywać informacje m.in. o bibliotekach, jakie instalujemy przy pomocy npm.

Zawartość utworzonego pliku package.json powinna wyglądać bardzo podobnie do tego, co widać z prawej strony.

Opis poszczególnych właściwości znajdziesz w dokumentacji.

Format utworzonego pliku to plik JSON, który często jest wykorzystywany do wymiany informacji pomiędzy różnymi językami programowania czy aplikacjami.

```
{
  "name": "webpack-helloworld",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error:[...]\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Nadszedł czas na instalację Webpacka. Będziemy potrzebować dwóch paczek: `webpack` oraz `webpack-cli`.

Pierwsza to silnik naszego narzędzia, a druga to tzw. *command line interface* (CLI), czyli lista komend, dzięki którym zarządzamy webpackiem.

W terminalu, który wskazuje na ten sam katalog, co ostatnio, wpisujemy:

```
npm i webpack@4 webpack-cli@3 -D
```

Wyjaśnijmy sobie składniki zaprezentowanej komendy:

- `npm i` - skrót od *npm install* czyli zainstaluj paczki,
- `webpack@4` - paczka o nazwie *webpack* w wersji *4*,
- `webpack-cli@3` - paczka o nazwie *webpack-cli* w wersji *3*,
- `-D` - alternatywa do --save-dev, zapisz paczki w zależnościach deweloperskich.

Po uruchomieniu omówionej komendy, *npm* instaluje wskazane biblioteki.

Może to trochę potrwać - zależnie od szybkości łącza internetowego, ponieważ paczki są instalowane z zewnętrznego zasobu, tj. npmjs.com.

Od teraz package.json zawiera informacje o tym, jakie elementy zainstalowaliśmy. Zobacz, jak wygląda teraz nasz plik.

```
{
  "name": "webpack-helloworld",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: [...]\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "webpack": "^4.41.2",
    "webpack-cli": "^3.3.9"
  }
}
```

Dodatkowo w katalogu webpack-helloworld, w którym uruchomiliśmy instalację, pojawił się nowy element - katalog node_modules.

To tam zostały zainstalowane wszystkie niezbędne pliki, które są potrzebne do uruchamiania *webpack* i *webpack-cli*.

Katalog node_modules nigdy nie powinien być dodawany do *gita* czy *GitHuba* (trzeba użyć pliku .gitignore).

Powodem tego jest fakt, że instalacja paczek zależy od systemu operacyjnego i jego wersji, więc każdy użytkownik (czy programista) powinien wykonać taką instalację na własnym komputerze samodzielnie.

Gdy posiadamy plik package.json i mamy tam zapisane zależności, to wystarczy wykonać komendę: npm i.

Menadżer pakietów sprawdzi, co mamy zapisane w zależnościach i zainstaluje brakujące elementy.

Mając już zainstalowane niezbędne paczki, możemy utworzyć pierwszy plik konfiguracyjny dla webpacka.

Plik o nazwie webpack.config.js musimy utworzyć w katalogu głównym (u nas to webpack-helloworld).

Obok znajdziesz minimalną konfigurację, która wczyta plik ./src/app.js i na jego podstawie utworzy plik ./build/app.min.js.

```
const path = require('path');
// importuję bibliotekę [path] z [node.js]
module.exports = {
    entry: './src/app.js',
    // definiuję plik wejściowy
    output: {
        path: path.resolve(__dirname, 'build'),
        // definiuję ścieżkę wyjściową
        filename: 'app.min.js',
        // definiuję nazwę pliku wyjściowego
    },
    module: {
        rules: []
        // obecnie brak dodatkowych ustawień
    },
}
// eksportuję ustawienia dla webpacka
```

Aby uruchomić *webpacka* wystarczy w terminalu uruchomić komendę: `npx webpack --mode production`

Aby zobaczyć efekt działania *webpacka*, najpierw powinniśmy utworzyć katalog `./src`, a potem plik `./src/app.js`.

Możemy tam wpisać dowolny kod JS, np. `console.log('webpack')`.

Na następnym slajdzie poznasz działanie komendy `--mode production`.

```
const path = require('path');
// importuję bibliotekę [path] z [node.js]
module.exports = {
    entry: './src/app.js',
    // definiuję plik wejściowy
    output: {
        path: path.resolve(__dirname, 'build'),
        // definiuję ścieżkę wyjściową
        filename: 'app.min.js',
        // definiuję nazwę pliku wyjściowego
    },
    module: {
        rules: []
        // obecnie brak dodatkowych ustawień
    },
}
// eksportuję ustawienia dla webpacka
```

Wspomniana komenda optymalizuje nasz kod pod wersję produkcyjną, czyli taką, która jest udostępniana użytkownikowi końcowemu.

Zależy nam wtedy na kodzie, który będzie szybko uruchamiany, co *webpack* stara się zrealizować poprzez kod podobny do tego obok.

Jest on nieczytelny, co stanowi jego zaletę, ponieważ trudno go odtworzyć (co nie oznacza, że jest to niemożliwe).

```
!function(e){var t={};function n(r){if(t[r])return
t[r].exports;var o=t[r]={i:r,l:!1,exports:{}};return
e[r].call(o.exports,o,o.exports,n),o.l=!0,o.exports}n.
m=e,n.c=t,n.d=function(e,t,r){n.o(e,t)||Object.defineP
roperty(e,t,{enumerable:!0,get:r})},n.r=function(e){"u
ndefined"!=typeof
Symbol&&Symbol.toStringTag&&Object.defineProperty(e,Sy
mbol.toStringTag,{value:"Module"}),Object.defineProper
ty(e,"__esModule",{value:!0})},n.t=function(e,t){if(1&
t&&(e=n(e)),8&t)return e;if(4&t&&"object"==typeof
e&&e&&e.__esModule)return e;var
r=Object.create(null);if(n.r(r),Object.defineProperty(
r,"default",{enumerable:!0,value:e}),2&t&&"string"!=ty
peof e)for(var o in e)n.d(r,o,function(t){return
e[t]}.bind(null,o));return r},n.n=function(e){var
t=e&&e.__esModule?function(){return e.default} /* ...
*/
```

Teraz, aby zobaczyć, czy plik utworzony przez *webpacka* działa poprawnie, musimy utworzyć plik HTML, który uruchomi JS.

Robimy to, jak do tej pory, z tym że wskazujemy ścieżkę do nowego pliku, tj. `./build/app.min.js`.

Kod takiego HTML-a mamy obok.

Pamiętaj, aby ten plik również znajdował się w katalogu głównym - tak, jak plik `webpack.config.js`.

```
<!DOCTYPE html>
<html lang="pl">
<head>
    <meta charset="UTF-8">
    <meta name="viewport"
        content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible"
        content="ie=edge">
    <title>devmentor.pl - webpack</title>
</head>
<body>

    <script src="./build/app.min.js"></script>
</body>
</html>
```

#02 Production vs Development

02. Webpack

Wiemy już, że wersja produkcyjna ma działać w taki sposób, aby ułatwić życie użytkownikowi końcowemu.

Wersja deweloperska (a w zasadzie ten tryb) ma za zadanie ułatwić działania programiście.

Zanim dowiesz się, co to znaczy, to uruchomimy *webpacka* we wspomnianym trybie za pomocą komendy:
`npx webpack --mode development`.

```
/******/ (function(modules) { // webpackBootstrap
/******/ 	// The module cache
/******/ 	var installedModules = {};
/******/
/******/ 	// The require function
/******/ 	function __webpack_require__(moduleId) {
/******/
/******/

// ...

eval("console.log('webpack');\n\n//#
sourceURL=webpack:///./src/app.js?");

/***/ })

/******/ });
```

Tym razem nasz plik wynikowy wygląda bardziej czytelniej (obok jego urywek).

Jak odpalimy przeglądarkę, to okaże się, że informacje o wywołaniach w konsoli dotyczą pliku wejściowego (app.js), a nie wyjściowego (app.min.js).

To zdecydowanie ułatwi nam debugowanie! Nosi to nazwę source maps i jest automatycznie uruchamiane w trybie developerskim.

```
/******/ (function(modules) { // webpackBootstrap
/******/ 	// The module cache
/******/ 	var installedModules = {};
/******/
/******/ 	// The require function
/******/ 	function __webpack_require__(moduleId) {
/******/
/******/

// ...

eval("console.log('webpack');\n\n//#
sourceURL=webpack:///./src/app.js?");

/***/ })

/******/ });
```

Do trybu deweloperskiego będziemy używać specjalnej paczki o nazwie `webpack-dev-server`.

Dzięki niemu uruchomimy lokalny serwer, który ułatwi nam pracę - choćby przez automatyczne odświeżanie strony za każdym razem, gdy zapiszemy zmiany dokonane w pliku wejściowym.

Oczywiście zanim zaczniemy z niego korzystać, musimy go zainstalować:

```
npm i webpack-dev-server@3 -D
```

Dodatkowo wykorzystamy HtmlWebpackPlugin, który pozwoli nam utworzyć kopię naszego źródłowego pliku HTML i dołączyć do niego pliki JS. Instalujemy:

```
npm i html-webpack-plugin@3 -D
```

Musimy teraz wprowadzić zmianę w konfiguracji *webpacka*. Modyfikujemy webpack.config.js.

Na początku importujemy zainstalowaną przed chwilę paczkę (plugin).

Następnie określamy jej wykorzystanie wraz z odpowiednimi ustawieniami.

```
const path = require('path');
// importuję bibliotękę [path] z [node.js]
const HtmlWebpackPlugin = require('html-webpack-plugin');
// importuję odpowiedni plugin
module.exports = {
    // ...

    plugins: [
        new HtmlWebpackPlugin({
            template: './src/index.html',
            // wskazuję plik źródłowy
            filename: 'index.html'
            // określam nazwę dla pliku
        })
    ]
}
```

Ponieważ określiliśmy, że plik HTML znajduje się w katalogu ./src, przenosimy go tam i usuwamy element <script>.

Od teraz to nasz plugin będzie tworzył kopię pliku ./src/index.html i dołączał do niego <script>.

Przetestujmy:
npx webpack --mode production

Czy w lokalizacji ./build znajduje się plik index.html ze znacznikiem <script>?

```
const path = require('path');
// importuję bibliotekę [path] z [node.js]
const HtmlWebpackPlugin = require('html-webpack-plugin');
// importuję odpowiedni plugin
module.exports = {
    // ...

    plugins: [
        new HtmlWebpackPlugin({
            template: './src/index.html',
            // wskazuję plik źródłowy
            filename: 'index.html'
            // określam nazwę dla pliku
        })
    ]
}
```

Sprawdźmy teraz, czy nasz *webpack-dev-server* działa prawidłowo:
`npx webpack-dev-server --open`

Dzięki opcji `--open` powinna się pojawić nowa zakładka w przeglądarce.

Zmień coś w pliku `./src/app.js` i zapisz. Na stronie od razu powinna być widoczna zmiana. Super!

Tutaj nie musimy definiować trybu deweloperskiego. Jest on automatycznie uruchamiany przez *webpack-dev-server*.

Zauważ, że w terminalu ciągle trwa nasłuchiwanie na zmiany.

Jeśli chcesz przerwać ten proces to wystarczy skrót klawiszowy `ctrl + c`.

Pamiętaj również, że przy każdej zmianie konfiguracji dla *webpacka* należy ponownie go uruchomić (dotyczy to również *webpack-dev-server*).

Na koniec pewne ułatwienie. Do tej pory za każdym razem wpisywaliśmy całą komendę uruchamiającą *webpacka* lub *webpack-dev-server*.

Utwórzmy skróty w naszym pliku `package.json`, które będą uruchamiały wcześniej używane komendy.

Teraz wystarczy wpisać w konsolę: `npm run start` lub `npm run build`.

Uwaga. *Jeśli używasz Windowsa, to należy wykonać dodatkowe działania - o tym za chwilę.*

```
{
  "name": "webpack-helloworld",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: (...) \" && exit 1",
    "start": "webpack-dev-server --open",
    "build": "webpack --mode production"
  },
  "devDependencies": {
    "html-webpack-plugin": "^3.2.0",
    "webpack": "^4.41.2",
    "webpack-cli": "^3.3.9",
    "webpack-dev-server": "^3.9.0"
  }
}
```

Podsumowując: `npm run build` uruchamiamy, kiedy chcemy udostępnić pliki użytkownikowi końcowemu - znajdują się one w katalogu `./build`.

Natomiast `npm run start` odpalamy, gdy chcemy rozwijać naszą aplikację.

Dodatkowo warto wiedzieć, że `webpack-dev-server` nie tworzy plików, tylko zapisuje je w pamięci RAM. To przyśpiesza cały proces i nie obciąża tak dysku twardego, co ma znaczenie dla jego żywotności (szczególnie dla dysków SSD).

```
{
  "name": "webpack-helloworld",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: (...) \" && exit 1",
    "start": "webpack-dev-server --open",
    "build": "webpack --mode production"
  },
  "devDependencies": {
    "html-webpack-plugin": "^3.2.0",
    "webpack": "^4.41.2",
    "webpack-cli": "^3.3.9",
    "webpack-dev-server": "^3.9.0"
  }
}
```

Microsoft postanowił w swoim systemie ograniczyć możliwość uruchamiania skryptów z poziomu wiersza poleceń.

Aby to zmienić, należy zgodnie z oficjalną instrukcją ustawić odpowiednią flagę w PowerShell. Pamiętaj, aby uruchomić go w trybie administratora.

Następnie poniższą wprowadź komendę i potwierdź ją enterem:

```
Set-ExecutionPolicy -ExecutionPolicy RemoteSigned
```

```
{
  "name": "webpack-helloworld",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: (...) \" && exit 1",
    "start": "webpack-dev-server --open",
    "build": "webpack --mode production"
  },
  "devDependencies": {
    "html-webpack-plugin": "^3.2.0",
    "webpack": "^4.41.2",
    "webpack-cli": "^3.3.9",
    "webpack-dev-server": "^3.9.0"
  }
}
```

#03 Babel

02. Webpack

Babel jest tzw. transpilatorem, który w naszym przypadku tłumaczy kod zapisany za pomocą nowszej składni na kod zgodny ze starszym standardem języka JS.

Aby móc z niego korzystać, musimy zainstalować kilka dodatkowych paczek:

```
npm i @babel/core@7 -D
npm i @babel/preset-env@7 -D
npm i babel-loader@8 -D
```

Pierwsza z nich to silnik *Babela*.

Druga to zestaw zasad jakimi ma się kierować *Babel* przy transpilacji.

Ostatnia to zestaw reguł, które określają, w jaki sposób *webpack* ma współdziałać z *Babelem*. *Loader* pozwala zwiększać możliwości *webpacka* dzięki dodatkowym modułom.

Teraz musimy skonfigurować naszego *webpacka,* aby współpracował z *Babelem*.

Zaczynamy od pliku webpack.config.js.

Dodajemy informacje o nowym module, który będzie uruchamiany za pomocą babel-loader.

W naszym przypadku transpilowane będą wszystkie pliki wejściowe z rozszerzeniem .js z wyłączeniem tych z katalogu node_moduels.

```
module.exports = {
    // ...
    module: {
        rules: [
            {
                test: /\.js$/,
                // określam, jakie pliki
                // będą brane pod uwagę
                exclude: /node_modules/,
                // określam wykluczenia
                use: 'babel-loader',
                // określam, jaki [loader]
                // ma być wykorzystany
            }
        ]
    // ...
    },
}
```

Musimy jeszcze utworzyć plik `.babelrc` w katalogu głównym (obok *webpack.config.js*), w którym zdefiniujemy, jakimi zasadami ma się kierować *Babel*.

Dodatkowo utworzymy w tej samej lokalizacji plik `.browserlistrc`, który pozwoli nam zdefiniować, jakie przeglądarki mają wspierać nasz transpilowany kod.

```
{
    "presets": ["@babel/preset-env"]
}
```

```
last 2 years
> 2%
not dead
```

W naszym przypadku zdefiniowaliśmy wszystkie przeglądarki, których wersje zostały wypuszczone w ostatnich 2 latach.

Dodatkowo ich udział w rynku jest większy niż 2% i posiadają one wsparcie producentów.

Więcej na temat możliwości definiowania wsparcia znajdziesz na GitHubie.

```
{
    "presets": ["@babel/preset-env"]
}
```

```
last 2 years
> 2%
not dead
```

Od teraz możemy pisać kod JS, korzystając z najnowszych rozwiązań i nie musimy się martwić wsparciem przez przeglądarki.

Babel sprawdzi, jakie przeglądarki, wśród tych, które określiliśmy, wspierają dany kod i, jeśli zajdzie taka potrzeba, to go zamieni na starszy odpowiednik.

Wyobraź sobie, ile trzeba by było poświęcić czasu na wykonanie czegoś takiego ręcznie. Praktycznie nie do zrobienia w rozsądnym czasie!

Całą strukturę kodu, jaką omawialiśmy, znajdziesz na GitHubie.

Wystarczy, że pobierzesz repozytorium i uruchomisz instalacją paczek poprzez komendę: `npm i`.

Ponieważ w `package.json` mamy zapisane zależności, to zostaną one zainstalowane.

03. Łańcuchy szablonowe

#01 Template strings

03. Łańcuchy szablonowe

Łańcuchy szablonowe (ang. *template strings*) to nowa forma zapisu ciągów znaków.

Pozwalają one tworzyć wielowierszowe ciągi znaków oraz wygodniej osadzać w nich zmienne, a nawet wywoływać funkcje.

Taki ciąg definiujemy za pomocą backticka (`), który możemy znaleźć na klawiaturze zazwyczaj w okolicy tyldy (~). *Backtick* czasem nazywany jest grawisem.

```
const num = 1;
const text1 = `string with value from num: ${ num }`;
// zawartość: string with value from num: 1
const getNum = function() {
    return num;
}
const text2 = `string with value
from function: ${ getNum() }`;
// zawartość (dwie linie): string with value
// from function: 1

console.log( text1 );
console.log( text2 );
```

Aby utworzyć taki ciąg znaków, wystarczy interesującą nas zawartość umieścić między dwoma *backtickami*.

Jeśli chcemy osadzić lub wywołać funkcję wewnątrz łańcucha szablonowego, to wystarczy użyć znaku dolara (`$`) oraz nawiasów klamrowych (`{}`) wewnątrz których wstawimy dowolne wyrażenie.

Może to być również działanie matematyczne czy nazwa zmiennej.

```
const user = {
    firstName: 'Łukasz',
    lastName: 'Nowak',
    birthYear: 1990,
};

const createMessage = function(u) {
    const currYear = (new Date()).getFullYear();
    const message = `${u.firstName} ${u.lastName},
obchodzi w tym roku
${currYear - u.birthYear} urodziny!`;

    return message;
}

const message = createMessage(user);
console.log(message);
```

Może się to wydawać niewielkim usprawnieniem, jednak czytelność kodu bardzo się poprawił. Możesz to porównać z przykładem obok.

Prezentuje on użycie konkatenacji, czyli operatora `+` dla połączenia ciągów znaków.

Znak `\n` oznacza znak nowej linii, którego nie musieliśmy stosować używając *backticka*.

```
const user = {
    firstName: 'Łukasz',
    lastName: 'Nowak',
    birthYear: 1990,
};

const createMessage = function(u) {
    const currYear = (new Date()).getFullYear();
    const message = u.firstName +' '+ u.lastName +'\n'+
'obchodzi w tym roku \n' +
(currYear-u.birthYear) + ' urodziny!';

    return message;
}

const message = createMessage(user);
console.log(message);
```

#02 Tagged template strings

03. Łańcuchy szablonowe

Oznaczone łańcuchy szablonowe to specjalna konstrukcja, która pozwala wywołać funkcję w dość specyficzny sposób.

Na zdefiniowanym łańcuchu szablonowym wykonujemy dowolną operację i zwracamy zmieniony ciąg znaków.

Przyjrzyjmy się przykładowi obok. Deklarujemy funkcję fn() z *4 parametrami*, która zwraca ciąg znaków ‘abc’.

```
const fn = function(arr, arg1, arg2, arg3) {
    console.log(
        arr,
        arg1,
        arg2,
        arg3
    );
    return 'abc';
}

const v1 = 111;
const v2 = 222;

const text = fn`Ciąg znaków z ${v1} oraz ${v2} i tyle!`;
console.log(text); // abc
```

Następnie wywołujemy tę funkcję, wykorzystując jej nazwę tj. fn, a następnie od razu wprowadzamy łańcuch szablonowy (bez nawiasów okrągłych).

Wywołanie w ten sposób powoduje, że do parametru arr jest przekazywana tablica, która posiada 3 wartości:

- ‘Ciąg znaków z ‘
- ‘ oraz ‘
- ‘ i tyle!’

Czyli stringi rozdzielone wcześniej wyrażeniami.

```
const fn = function(arr, arg1, arg2, arg3) {
    console.log(
        arr,
        arg1,
        arg2,
        arg3
    );
    return 'abc';
}

const v1 = 111;
const v2 = 222;

const text = fn`Ciąg znaków z ${v1} oraz ${v2} i tyle!`;
console.log(text); // abc
```

Pozostałe parametry przyjmują wartości:

- **arg1** -> 111
- **arg2** -> 222
- **arg3** -> undefined

Trzeci parametr jest `undefined`, ponieważ w łańcuchu nie było trzech, a jedynie dwa wyrażenia.

Natomiast do zmiennej `text` została przypisana wartość `'abc'` ponieważ tą wartość zwraca funkcja `fn`.

```
const fn = function(arr, arg1, arg2, arg3) {
    console.log(
        arr,
        arg1,
        arg2,
        arg3
    );
    return 'abc';
}

const v1 = 111;
const v2 = 222;

const text = fn`Ciąg znaków z ${v1} oraz ${v2} i tyle!`;
console.log(text); // abc
```

Skoro znamy już zasadę działania, to przyjrzyjmy się bardziej praktycznemu przykładowi.

Mamy dwie funkcje, które przyjmują ten sam łańcuch szablonowy.

Dzięki różnej implementacji jedna zwróci kod *HTML*, a druga *Markdown*.

Wrócimy jeszcze do tego tematu, jak tylko poznamy kolejne usprawnienia wprowadzone w *ES6*.

```
function useHtml(str, val) {
    return `${ str[0] } <b>${ val }</b> ${ str[1] }`;
}

function useMarkdown(str, val) {
    return `${ str[0] } *${ val }* ${ str[1] }`;
}

const name = 'Łukasz';
console.log(
    useHtml`Super programista ten ${name}!`,
    // Super programista ten <b>Łukasz</b>!
    useMarkdown`Super programista ten ${name}!`,
    // Super programista ten *Łukasz*!
);
```

04. Funkcje strzałkowe

Funkcja strzałkowa (ang. *arrow function*, *fat arrow*) to specjalny zapis, który przyspiesza tworzenie funkcji.

Funkcja strzałkowa nie posiada nazwy, dlatego musimy ją przypisać do zmiennej.

Zamiast pisać słowo kluczowe *function*, jak do tej pory, pomijamy je, a między nawiasami okrągłymi oraz klamrowymi wstawimy znak strzałki (`=>`).

To wszystko! Wywołanie funkcji strzałkowej pozostaje takie samo jak do tej pory.

```
const sumFn = function(a, b) {
    return a + b;
}

const sumFnArrow = (a, b) => {
    return a + b;
}

console.log(
    sumFn(3, 5),
    sumFnArrow(4, 5),
);
```

Ponieważ przy pomocy funkcji strzałkowej tworzymy wyrażenie funkcyjne, to hoisting nie ma tutaj zastosowania.

Należy najpierw zdefiniować funkcję, a potem ją wykorzystać.

Często funkcje strzałkowe są wykorzystywane przy bezpośrednim przekazywaniu funkcji jako parametr.

```
const arr = [1, 2, 3, 4];

arr.forEach( (num, index) => {
    console.log(num, '=>', index);
});

const newArr = arr.map( (item, i) => {
    return item * i;
});

console.log(newArr);
// [0, 2, 6, 12]
```

#01 Zapis skrótowy

04. Funkcje strzałkowe

W szczególnych przypadkach możemy skrócić zapis funkcji strzałkowej.

Ma to miejsce, gdy funkcja posiada tylko jedną instrukcję.

Wtedy pomijamy nawiasy klamrowe i słowo kluczowe `return`.

Funkcja zwraca to, co jest po prawej stronie strzałki (`=>`). W naszym przypadku jest to wynik działania `a + b`.

```
const sumFn = function(a, b) {
    return a + b;
}

const sumFnArrow = (a, b) => a + b;

console.log(
    sumFn(3, 5),
    sumFnArrow(4, 5),
);
```

Dodatkowo, jeśli funkcja posiada tylko jeden parametr, możemy zrezygnować z nawiasów okrągłych.

Wtedy z lewej strony strzałki wstawiamy jedynie nazwę parametru.

Należy uważać, aby nazwa parametru oraz strzałka była zapisane w tej samej linii. Inaczej pojawi się błąd składni.

```
const showInfoFn = function(info) {
    console.log(info);
}

const showInfoFnArrow = info => console.log(info);

showInfoFn('eeee');
showInfoFnArrow('wow!');
```

Teraz możemy zobaczyć pełną użyteczność funkcji strzałkowej.

W jednej linii możemy zapisać operację, która do tej pory wymagała od nas większej ilości kodu.

Zauważ również, że nie dzieje się nic złego, gdy funkcja niczego nie zwraca - tak jak w przypadku wykorzystania `console.log()`.

```
const arr = [1, 2, 3, 4];

arr.forEach( num => console.log(num) );
// 1
// 2
// 3
// 4

const newArr = arr.map( item => item * 2 );
console.log(newArr);
// [2, 4, 6, 8]
```

#02 Kontekst dla this

04. Funkcje strzałkowe

Do tej pory każda funkcja tworzyła kontekst dla this.

Upraszczając, można powiedzieć, że w zależności od miejsca wywołania funkcji, this mógł wskazywać na inną wartość.

W przykładzie obok funkcja showText() jest wywoływana wewnątrz innej funkcji, dlatego w tym wypadku this nie wskazuje już na item, lecz na obiekt window, który nie posiada właściwości .innerText.

```
const liList = document.querySelectorAll('li');
liList.forEach( item => {
    // [item] wskazuje na konkretne <li/>
    item.addEventListener('click', function() {
        // [this] wskazuje na <li/>
        function showText() {
            // kontekst dla [this]
            // został zmieniony
            console.log(
                this.innerText
            );
            // [undefined] w konsoli
        }
        showText();
    });
});
```

Jeśli funkcję showText() zamienimy na funkcję strzałkową, to this ciągle będzie wskazywał na tą samą wartość, co poza deklaracją funkcji.

Nasz przykład zadziała tak, jak byśmy tego oczekiwali. Kontekst dla this się nie zmienił!

```
const liList = document.querySelectorAll('li');
liList.forEach( item => {
    // [item] wskazuje na konkretne <li/>
    item.addEventListener('click', function()  {
        // [this] wskazuje na <li/>
        const showText = () => {
            // kontekst dla [this]
            // nie został zmieniony
            console.log(
                this.innerText
            );
            // w konsoli zobaczymy tekst
            // klikniętego <li/>
        }
        showText();
    });
});
```

05. Destrukturyzacja

Destrukturyzacja (ang. *destructuring*) nazywana jest również dekompozycją lub przypisaniem destrukturyzacyjnym (ang. *destructuring assignment*).

Polega na pobraniu danych z tablic lub obiektów i przypisaniu ich do zmiennych.

```
const arr = [0, 1, 2, 3];
const [a, b] = arr;
console.log(a, b);
// 0 1

const obj = {
    firstName: 'Łukasz',
    lastName: 'Nowak'
}
const { firstName } = obj;
console.log(firstName);
// Łukasz
```

#01 Tablice

05. Destrukturyzacja

Destrukturyzacji tablic możemy dokonać poprzez wstawienie z lewej strony przypisania nawiasów kwadratowych, a wewnątrz nich nazw zmiennych, do których mają być przypisane wartości.

W przypadku tablic bardzo ważna jest kolejność. Pierwsza nazwa zmiennej otrzyma pierwszą wartość z tablicy.

Jeśli chcemy pominąć którąś z wartości, to wystarczy wstawić dodatkowy przecinek.

```
const arr = [0, 1, 2, 3];
const [a, b] = arr;
console.log(a, b);
// 0 1

let c = 0;
[, , c] = arr;
console.log(c);
// 2
```

Przedstawione wcześniej zasady możemy wykorzystać podczas przekazywania parametrów do funkcji.

W przykładach obok pobierzemy jedynie pierwszą i trzecią wartość z tablicy przekazanej jako parametr.

W przypadku użycia destrukturyzacji w funkcji strzałkowej nie możemy pominąć nawiasów okrągłych.

```
const arr = [0, 1, 2, 3];

const fn1 = function([a , , c]) {
    console.log(a, c);
}

const fn2 = ([a , , c]) => {
    console.log(a, c);
}

fn1(arr); // 0 2
fn2(arr); // 0 2
```

#02 Obiekty

05. Destrukturyzacja

Destrukturyzacja obiektów przebiega na podobnej zasadzie z tą różnicą, że używamy nawiasów klamrowych - co jest oczywiste, ponieważ to z ich pomocą tworzymy obiekty.

Do utworzenia nazw zmiennych używamy nazw właściwości obiektu, co pokazuje przykład obok.

```
const obj = {
    firstName: 'Łukasz',
    lastName: 'Nowak',
    age: 22,
}

const {
    firstName,
    lastName,
} = obj;
// można to napisać też
// w jednej linii

console.log(
    firstName,
    lastName
); // Łukasz Nowak
```

Możemy również tworzyć nazwy zmiennych inne niż nazwy właściwości obiektu.

Musimy tylko pozostawić powiązanie, a po znaku dwukropka wpisać nową nazwę zmiennej, której chcemy użyć.

```
const obj = {
    firstName: 'Łukasz',
    lastName: 'Nowak',
    age: 22,
}

const {
    firstName: name,
    lastName: secondName,
} = obj;
// można to napisać też
// w jednej linii

console.log(
    name,
    secondName
); // Łukasz Nowak
```

Podobnie jak w destrukturyzacji tablic, możemy użyć destrukturyzacji obiektów przy przekazywaniu argumentów do funkcji.

Definiujemy tylko interesujące nas właściwości obiektu, a ich wartości zostają przypisane do zmiennych o tych samych nazwach.

Teraz możemy je wygodnie wykorzystać we wnętrzu naszych funkcji.

W przypadku funkcji strzałkowej, tak jak wcześniej, musimy zachować nawiasy okrągłe.

```
const user = {
    firstName: 'Łukasz',
    lastName: 'Nowak',
    age: 22,
}

const sayHelloFn = function( {firstName, lastName} ) {
    console.log(firstName, lastName);
}

const sayHelloFnArrow = ( {firstName, lastName} ) => {
    console.log(firstName, lastName);
}

sayHelloFn(user); // Łukasz Nowak
sayHelloFnArrow(user); // Łukasz Nowak
```

#03 Podsumowanie

05. Destrukturyzacja

Destrukturyzacja jest bardzo wygodnym sposobem na utrzymanie czytelności kodu.

W szczególności, gdy mamy bardzo rozbudowane struktury i chcemy wyciągnąć dane z ich głębi.

W takim przypadku możemy wykorzystać oba rodzaje destrukturyzacji - jak w przykładzie obok.

```
const place = {
    name: {
        pl: 'Kraków',
        la: 'Cracovia',
        de: 'Krakau',
    },
    position: [50.061389, 19.938333],
}

const {
    name: {pl: plName},
    position: [lat, lng],
} = place;

console.log(`${plName} => ${lat} ${lng}`);
// Kraków => 50.061389 19.938333
```

06. Operator ...

W zależności od miejsca użycia, operator `...` (trzy kropki) może nosić nazwę operatora rozproszenia (ang. *spread operator*) lub operatora reszty (ang. *rest operator*).

```
const arrNums = [1, 2, 3];
const arrBools = [true, false];

console.log( [...arrNums, ...arrBools]);
// [1, 2, 3, true, false]

const calcSum = (...nums) => {
    return nums.reduce( (acc, num) => acc + num, 0 );
}

console.log( calcSum(1, 2, 3, 4, 5) );
// 15
```

#01 Rozproszenie

06. Operator ...

Operator rozproszenia (ang. *spread operator*) występuje wtedy, gdy chcemy większą grupę elementów rozdzielić na poszczególne wartości.

Jeśli mielibyśmy być bardziej dokładni, to powiedzieliśmy, że te grupy elementów implementują interfejs *Iterable*.

Oznacza to, że możemy po nich iterować - tak jak to ma miejsce np. w przypadku tablic.

```
const arr = [1, 2, 3];

const calcSum = function(a, b, c) {
    return a + b + c;
}

console.log( calcSum(arr[0], arr[1], arr[2]) );
// 6

console.log( calcSum(...arr) );
// 6
```

Skoro wiemy, że tablica implementuje interfejs *Iterable,* to oznacza, że możemy rozdzielić jej elementy na poszczególne wartości. To właśnie robimy w przykładzie obok.

Mamy funkcję, która przyjmuje 3 argumenty.

Możemy przekazać do niej wartości tablicy, odwołując się do ich indeksów.

Możemy również wykorzystać operator rozproszenia, który przekaże do funkcji poszczególne wartości tablicy.

```
const arr = [1, 2, 3];

const calcSum = function(a, b, c) {
    return a + b + c;
}

console.log( calcSum(arr[0], arr[1], arr[2]) );
// 6

console.log( calcSum(...arr) );
// 6
```

To rozwiązanie często wykorzystujemy również wtedy, gdy nie wiemy, ile argumentów będziemy chcieli przekazać.

Przykładowo obiekt `Math` posiada metodę `.max()`, do której można przekazać dowolną ilość parametrów.

Na podstawie tych parametrów zostanie zwrócona największa wartość.

Dzięki rozproszeniu możemy od razu przekazać wszystkie wartości z tablicy.

```
const arr = [111, 72, 23, 51];

console.log(
    Math.max( ...arr ), // ES6
    Math.max.apply(null, arr), // ES5
    Math.max(arr[0], arr[1], arr[2], arr[3]),
);
```

Rozproszenia możemy również użyć do utworzenia kopii naszej tablicy czy obiektu.

Często nie chcemy operować na tych samych danych, lecz na ich kopii.

Jest to o tyle ważne, że typy złożone w JavaScript sa [kopiowane przez referencję](#).

To czasem utrudnia debugowanie naszego kodu. O niezmienności (ang. *immutability*) jeszcze będziemy mówić.

Tymczasem utwórzmy kopię tablicy.

```
const arr = [12, 14, 16, 18];
const newArr = [ ...arr ];

console.log( newArr )
// [12, 14, 16, 18]

console.log( arr === newArr );
// false
// mimo że zawartość tablicy jest taka sama,
// to operator porównania ocenia,
// czy elementy wskazują
// na ten sam adres w pamięci RAM

console.log( arr.slice() );
// [12, 14, 16, 18]
// .slice() też tworzy kopię tablicy
```

To rozwiązanie pozwala nam również zamienić obiekt *Iterable* na tablicę, dzięki czemu będziemy mogli skorzystać z metod dostępnych dla tablic.

Możemy np. zmienić obiekt *NodeList* (zwracany przez `.querySelectorAll()`) na tablicę i potem wykorzystać `.reduce()`, aby obliczyć sumę wartości przechowywanych w treści znaczników.

```
const liList = document.querySelectorAll('li');
// przyjmujemy że istnieją 4 elementy <li/>
// i zawierają odpowiednio: 1, 2, 3, 4
const liListArr = [ ...liList ];
// teraz jest to tablica, której elementami są
// poszczególne znaczniki <li/>

const sum = liListArr.reduce( (acc, item) => {
    return acc + parseInt(item.innerText);
}, 0);
// ponieważ to tablica, to mogę
// wykorzystać reduce do obliczenia sumy

console.log(sum);
```

Wraz z nadejście *ES2018* możemy wykorzystać rozproszenie do utworzenia kopii obiektów lub ich łączenia.

W podobny sposób można połączyć tablice, co było już możliwe od wersji *ES2015*.

```
const basicData = {
    firstName: 'Łukasz',
    lastName: 'Nowak',
}

const extendedData = {
    weight: '82kg',
    height: '187cm',
}

const person = { ...basicData, ...extendedData };
console.log(person);
// {
//    firstName: 'Łukasz', lastName: 'Nowak',
//    weight: '82kg', height: '187cm',
// }
```

Należy pamiętać, że kopiowanie elementów, które odbywa się za pomocą rozproszenia, to tzw. kopiowanie płytkie (ang. *shallow copy*).

Oznacza to, że kopiowanie realizowane jest tylko na pierwszym poziomie zagnieżdżenia. Potem znów mamy referencję.

Niestety na chwilę obecną JS nie oferuje idealnego rozwiązania do kopiowanie głębokiego (ang. *deep copy*).

```
const placeA = {
    name: {
        pl: 'Kraków',
        la: 'Cracovia',
        de: 'Krakau',
    },
    position: [50.061389, 19.938333],
}

const placeB = { ...placeA }
// tworzymy kopie płytką
placeB.name.pl = 'Warszawa';
// zmieniamy wartość dla nowego obiektu

console.log(placeA.name.pl);
// Warszawa
// okazuje się, że zmodyfikowaliśmy też [placeA]
```

#02 Reszta

06. Operator ...

Operatora reszty wykorzystujemy wtedy, kiedy chcemy zabrać pozostałe elementy w jedną grupę.

Sama definicja już sugeruje, że jest to odwrotność rozproszenia i faktycznie tak można powiedzieć.

W przykładzie obok zbieramy wszystkie argumenty, jakie zostały przekazane do funkcji.

```
const calcSum1 = function(...args) {
    return args.reduce( (acc, num) => {
        return acc + num;
    }, 0);
}
const calcSum2 = (...params) => {
    let sum = 0;
    params.forEach( num => {
        sum += num;
    });

    return sum;
}
console.log(
    calcSum1(1, 2, 3, 4, 5), //15
    calcSum2(3, 5, 6, 7), //21
);
```

Jak pokazano na przykładzie obok, sposób napisania funkcji jest bez znaczenia.

Operator reszty zbiera wszystkie elementy w tablicę, więc możemy się do nich odwołać za pomocą metod tablicowych.

Nic nie stoi na przeszkodzie, aby odwoływać się do nich po indeksie.

Kolejność wartości w tablicy zależy od kolejności przekazania.

```
const assignPrice = (name, ...products) => {
    let price = 0;
    products.forEach(item => price += item.price);

    return {
        name,
        price: price,
        products: products,
    }
}
console.log(
    assignPrice(
        'Łukasz Nowak',
        {name: 'Karty', price: 20},
        {name: 'książka', price: 49},
    )
);
```

Wracając jeszcze do samego przykładu - przekazywanie wartości do obiektu możemy skrócić, jeśli nazwa właściwości ma taką samą nazwę jak zmienna.

Tak jak w przykładzie obok, tj.

- price: price => **price**
- products: products => **products**

```
const assignPrice = (name, ...products) => {
    let price = 0;
    products.forEach(item => price += item.price);

    return {
        name,
        price, // price: price
        products, // products: products
    }
}
console.log(
    assignPrice(
        'Łukasz Nowak',
        {name: 'Karty', price: 20},
        {name: 'książka', price: 49},
    )
);
```

Operatora reszty możemy używać również podczas destrukturyzacji.

Jak widzisz na przykładzie obok, dotyczy to zarówno tablic, jak i obiektów.

Ważne przy wykorzystaniu operatora reszty jest to, by był on ostatnim elementem destrukturyzacji, inaczej pojawi się błąd.

```
const arr = [1, 2, 3, 4];

const [ first, ...rest ] = arr;
console.log(first, rest);
// 1 [2, 3, 4]

const obj = {
    name: 'Marek',
    weight: '66kg',
    height: '182cm',
}

const { name, ...other } = obj;
console.log(name, other);
// Marek {weight: '66kg', height: '182cm'}
```

05. Wartości domyślne

Wartości domyślne to wartości przypisywane do zmiennych, których wartość nie została określona.

Wartości takie możemy definiować w parametrach funkcji oraz podczas destrukturyzacji.

Niestety nie możemy ich łączyć z operatorem reszty.

```
const fn1 = function(a, b = 'bbb', c = 12 ) {
    console.log(a, b, c);
}

const fn2 = (a, b, c = 12) => {
    console.log(a, b, c);
}

fn1(); // undefined 'bbb' 12
fn2(47); // 47 undefined 12

const obj = {name: 'Łukasz'}
const {name, age = 12} = obj;
console.log(name, age);
// Łukasz 12
```

Oba rozwiązania (parametry funkcji i destrukturyzację) możemy łączyć z wartościami domyślnymi.

W przykładzie obok w obiekcie `name` nie występuje właściwość `en`, dlatego do `en` została przypisana wartość domyślna.

W przypadku zmiennych `lat` i `lng` wartości w tablicy istnieją, dlatego to one zostały przypisane do zmiennych, a nie zdefiniowane wartości domyślne.

```
const place = {
    name: {
        pl: 'Kraków',
        la: 'Cracovia',
        de: 'Krakau',
    },
    position: [50.061389, 19.938333],
}

const {
    name: {en = 'Cracow'},
    position: [lat = 0, lng = 0],
} = place;

console.log(`${en} => ${lat} ${lng}`);
// Cracow => 50.061389 19.938333
```

06. Klasy

Standard *ES2015* wprowadził do składni JavaScript słowo kluczowe `class`.

Już na tym etapie warto wspomnieć, że jest to pewnego rodzaju usprawnienie w zapisie.

W JavaScript cały czas nie ma klas znanych z *Javy*, *C++* czy *PHP*.

Nadal obowiązują zasady związane z prototypami.

```
const Car = function() {

}
// Konstruktorów nie tworzymy
// za pomocą funkcji strzałkowych

class Person {

}
// nowy zapis w ES6

console.log(
    typeof Car, // function
    typeof Person, // function
);

// Dowód, że to ciągle to samo
```

Słowo kluczowe class pozwala nam tworzyć grupę właściwości i metod, które dotyczą tego samego elementu.

Można powiedzieć, że jest to pewnego rodzaju szablon, który potem wypełniamy danymi.

Aby wykorzystać ów szablon, musimy użyć słowa kluczowego new, które utworzy nowy obiekt na podstawie określonej klasy.

```
class Person {

}

const person1 = new Person();
const person2 = new Person();
const person3 = new Person();

console.log(
    typeof person1, // object
    person1 === person2, // false
)
```

Taki obiekt może posiadać metody i właściwości, które zostały zdefiniowane w klasie.

Definiowanie właściwości z przypisanymi wartościami początkowymi odbywa się za pomocą metody `constructor()`.

Nazwa ta jest zawsze stała i nie zależy od nazwy klasy.

```
class Person {
    constructor() {
        this.firstName = 'Łukasz';
        this.lastName = 'Nowak';
    }
}

const person1 = new Person();
const person2 = new Person();
const person3 = new Person();

console.log(
    person1, // {firstName: "Łukasz", lastName: "Nowak"}
    person2, // {firstName: "Łukasz", lastName: "Nowak"}
    person1 === person2, // false
);
```

#01 Konstruktor

06. Klasy

W konstruktorze możemy definiować właściwości powstającego obiektu za pomocą zmiennej this.

Jak już wiemy, this zależy od kontekstu.

W przykładzie kontekstem tym będą nasze utworzone obiekty, np. person1.

Podczas tworzenia obiektu za pomocą new Person(), this wskazuje właśnie na ten obiekt.

Dzięki temu tworzone są wartości początkowe dla obiektu.

```
class Person {
    constructor() {
        this.firstName = 'Łukasz';
        this.lastName = 'Nowak';
    }
}

const person1 = new Person();
const person2 = new Person();
const person3 = new Person();

console.log(
    person1, // {firstName: "Łukasz", lastName: "Nowak"}
    person2, // {firstName: "Łukasz", lastName: "Nowak"}
    person1 === person2, // false
);
```

Rozwiązanie to jest niedoskonałe, ponieważ teraz wszystkie obiekty posiadają te same wartości.

Możemy temu zaradzić, wykorzystując parametry w konstruktorze, a wartości dla naszych właściwości przekazać w postaci argumentów przy tworzeniu obiektów.

```
class Person {
    constructor(firstN, lastN) {
        this.firstName = firstN;
        this.lastName = lastN;
    }
}

const person1 = new Person('Łukasz', 'Nowak');
const person2 = new Person('Jan', 'Kowalski');
const person3 = new Person('Anna', 'Polak');

console.log(
    person1, // {firstName: "Łukasz", lastName: "Nowak"}
    person2, // {firstName: "Jan", lastName: "Kowalski"}
    person1 === person2, // false
);
```

Wykorzystywane nazwy są dowolne.

Przyjęło się jednak, że nazwy parametrów są takie same, jak nazwy właściwości, chociaż nie jest to regułą.

`constructor()`, jak i inne funkcje wewnątrz klasy, mogą wykorzystywać *destrukturyzajcę*, *operator reszty* czy *wartości domyślne* - tak jak normalne funkcje.

```
class Person {
    constructor({name, age = 0}) {
        this.name = name;
        this.age = age;
    }
}

const person1 = new Person({name: 'Łukasz Nowak'});
const person2 = new Person({
    name: 'Jan Kowalski', age: 19
});

console.log(
    person1, // {name: "Łukasz Nowak", age: 0}
    person2, // {name: "Jan Kowalski", age: 19}
    person1 === person2, // false
);
```

Do wartości konkretnych właściwości w obiekcie możemy się odwołać poprzez kropkę lub nawiasy kwadratowe.

Działa to tak samo, jak do tej pory w obiektach.

```
class Person {
    constructor({name, age = 0}) {
        this.name = name;
        this.age = age;
    }
}

const person1 = new Person({name: 'Łukasz Nowak'});
const person2 = new Person({
    name: 'Jan Kowalski', age: 19
});

console.log(
    person1.age, // 0
    person1['age'], // 19
);
```

#02 Metody

06. Klasy

Jak już wcześniej wspominaliśmy, metody to funkcje wewnątrz klas.

`constructor()` jest specjalną wersją metody, ponieważ ma ściśle określoną nazwę i jest wywoływany podczas tworzenia obiektu.

Możemy również tworzyć inne, dowolne funkcje i wywoływać je wtedy, kiedy tego potrzebujemy.

```
class Person {
    constructor({name, age = 0}) {
        this.name = name;
        this.age = age;
    }
    getName() {
        return this.name;
    }
    setName(name) {
        this.name = name;
    }
}

const person = new Person({name: 'Łukasz Nowak'});
person.setName('Jan Kowalski');
console.log( person.getName() ); // Jan Kowalski
```

Metody te mogą przyjmować parametry, ale nie muszą.

Mogą zwracać jakąś wartość, ale nie muszą.

Mogą operować na właściwościach danego obiektu za pomocą zmiennej this.

Ich wywołanie odbywa się za pomocą nazwy zmiennej, która przechowuje obiekt oraz kropki, po której występuje nazwa funkcji.

```
class Person {
    constructor({name, age = 0}) {
        this.name = name;
        this.age = age;
    }
    getName() {
        return this.name;
    }
    setName(name) {
        this.name = name;
    }
}

const person = new Person({name: 'Łukasz Nowak'});
person.setName('Jan Kowalski');
console.log( person.getName() ); // Jan Kowalski
```

#03 Dziedziczenie

06. Klasy

Dziedziczenie to sposób na skopiowanie rozwiązań z jednej klasy do drugiej bez powielania kodu.

Przykładowo: jest wielu ludzi na świecie, a wśród nich jest grupa, która mówi w języku polskim.

Mamy już zbudowaną klasę Person, więc skorzystamy z niej, tworząc klasę Polish.

```
class Person {
    constructor(name) {
        this.name = name;
        this.lang = null;
    }
    getName() {
        return this.name;
    }
}
class Polish extends Person {
    constructor(name) {
        super(name);
        this.lang = 'pl';
    }
}
const polish = new Polish('Łukasz Nowak');
console.log(polish.getName()); // "Łukasz Nowak"
```

W kodzie obok pojawiło się kilka nowych rozwiązań.

Zaczynamy od słowa kluczowego `extends`, które oznacza dziedziczenie, czyli “skopiuj właściwości i metody z klasy nadrzędnej, tj. rodzica (`Person`)”.

Dzięki temu zapisowi w klasie `Polish` mamy dostęp m.in. do metody `.getName()`.

```
class Person {
    constructor(name) {
        this.name = name;
        this.lang = null;
    }
    getName() {
        return this.name;
    }
}
class Polish extends Person {
    constructor(name) {
        super(name);
        this.lang = 'pl';
    }
}
const polish = new Polish('Łukasz Nowak');
console.log(polish.getName()); // "Łukasz Nowak"
```

Mamy również wywołanie funkcji `super()`, która oznacza wywołanie konstruktora rodzica.

Jej nazwa jest zawsze taka sama i pojawia się jako pierwsza instrukcja w nadpisanym konstruktorze.

To rozwiązanie pozwala nam ustawić właściwość `name` zgodnie z tym, co jest zdefiniowane w `constructor()` dla `Person` i jednocześnie dokonać własnych operacji.

```
class Person {
    constructor(name) {
        this.name = name;
        this.lang = null;
    }
    getName() {
        return this.name;
    }
}
class Polish extends Person {
    constructor(name) {
        super(name);
        this.lang = 'pl';
    }
}
const polish = new Polish('Łukasz Nowak');
console.log(polish.getName()); // "Łukasz Nowak"
```

Słowo kluczowe `super` może być również wykorzystywane jako odniesienie do rodzica w celu wywołania jego metody.

W przykładzie obok wykorzystaliśmy metodę rodzica, tj. `super.getArea()`, i otrzymany wynik pomnożyliśmy przez `this.c`.

Dzięki temu rozwiązaniu wykorzystaliśmy wcześniej napisany kod.

```
class Square {
    constructor(a, b) {
        this.a = a;
        this.b = b;
    }
    getArea() { return this.a * this.b }
}
class Cube extends Square {
    constructor(a, b, c) {
        super(a, b);
        this.c = c;
    }
    getArea() { return super.getArea() * this.c }
}
const cube1 = new Cube(1, 2, 3);
console.log( cube1.getArea() ); // 6
```

07. Moduły

Dzielenie kodu na mniejsze części (moduły) jest bardzo dobrą praktyką, która pozwala utrzymać go w lepszej organizacji.

Każdy moduł powinien realizować swoje zadanie i być jak najbardziej niezależny od reszty naszej aplikacji.

To zapewnia większą elastyczność dla naszego oprogramowania.

Reprezentacją modułu może być klasa (lub ich zestaw), która realizuje konkretne zadanie.

Skoro moduł ma być niezależny, przydałoby się przechowywać go w osobnej lokalizacji.

Może być to plik, który potem dołączamy do naszej strony za pomocą `<script>`.

Niestety sposób ten powoduje, że mamy wiele zmiennych globalnych. Nie jest więc dobrym podejściem.

Na szczęście możemy wykorzystać rozwiązania, które enkapsulują utworzone moduły i pozwalają na wskazanie elementów, które mogą być importowane i eksportowane.

#01 CommonJS

07. Moduły

CommonJS, który wywodzi się ze środowiska *node.js*, możemy dzięki *webpackowi* zastosować w JS.

Webpack połączy wszystkie eksporty i importy w jeden plik, który potem dołączymy do naszej strony.

Pliki źródłowe, na których pracujemy, pozostaną rozdzielone na moduły, natomiast kod produkcyjny będzie połączony i zoptymalizowany przez *webpacka*.

Z *CommonJS* będziemy korzystać do eksportowania danych za pomocą wyrażenia `module.exports = {}`, do którego przypisujemy to, co chcemy eksportować.

Natomiast przy imporcie będziemy używać składni `require("./adres/do/pliku.js")`.

Tworzymy nasz moduł w osobnym pliku, np. `getDate.js` i definiujemy, co chcemy eksportować, tj. udostępnić na zewnątrz.

W naszym przykładzie udostępniamy funkcję `getDate()`.

W pliku, w którym chcemy skorzystać z tego rozwiązania, musimy zaimportować moduł.

Zauważ, że możemy użyć całkowicie innej nazwy zmiennej.

```
const getDate = () => {
    return new Date();
}

module.exports = getDate;
```

```
const getCurrentDate = require('./getDate.js');

const data = getCurrentDate();
console.log( data.getFullYear() );
```

Podobnie działamy w przypadku, gdy mamy importować i eksportować klasę.

Wówczas nazwa pliku, jaki przechowuje klasę, zaczyna się wielką literą, co jest zgodne z konwencją nazewniczą. Jej autorem jest Douglas Crockford.

```
class Point {
    constructor(x, y) {
        this.x = x;
        this.y = y;
    }
}
module.exports = Point;
```

```
const Point = require('./Point.js');

const p1 = new Point(1, 2);
console.log( p1 ); // {x: 1, y: 2}
```

Oczywiście możemy eksportować więcej elementów i jednocześnie importować tylko te, które nas interesują.

W tym przypadku eksportujemy po prostu obiekt, który zawiera elementy do eksportu.

Natomiast przy imporcie wykorzystujemy zapis jak w destrukturyzacji.

```
const calcSum = (...args) => {
    return args.reduce( (acc, num) => acc + num, 0);
}
const getMax = (...args) => {
    return Math.max(...args);
}
module.exports = { calcSum, getMax }
```

```
const { calcSum } = require('./math.js');

const sum = calcSum(1, 2, 3, 4);
console.log( sum ); // 10
```

#02 ES6 modules

07. Moduły

Kolejny raz standard *ES2015* wprowadził do JavaScript rewolucyjne rozwiązanie, jakim jest import i export.

Obecnie większość nowoczesnych przeglądarek wspiera to rozwiązanie bez pomocy ze strony *webpacka*! O ile *webpack* umożliwia rezygnację z rozszerzenia importowanego pliku, o tyle, przy stosowaniu wsparcia przeglądarki, musimy o nim pamiętać.

My jednak pozostaniemy przy *webpacku*, by móc korzystać z innych jego dobrodziejstw.

```
const getDate = () => {
    return new Date();
}

export default getDate;
```

```
import getCurrenDate from './getDate.js';
// gdy import jest realizowany przez przeglądarkę,
// to rozszerzenie pliku [.js] musi pozostać

const data = getCurrenDate();
console.log( data.getFullYear() );
```

W przypadku eksportu używamy słów kluczowych `export default` jeśli chcemy wskazać domyślnie importowany element.

Taki element możemy importować do dowolnej zmiennej, która znajduje się pomiędzy słowami kluczowymi `import` oraz `from`, po którym wprowadzamy ścieżkę do pliku, z którego importujemy.

```
const getDate = () => {
    return new Date();
}

export default getDate;
```

```
import getCurrenDate from './getDate.js';

const data = getCurrenDate();
console.log( data.getFullYear() );
```

Podobnie wygląda eksport i import klas.

Czasami informacja o eksporcie jest definiowana od razu przy deklaracji klasy.

Nie ma reguły, jeśli chodzi o to rozwiązanie. Powinniśmy jednak trzymać się zasad, które panują w naszym zespole programistów.

```
export default class Point {
    constructor(x, y) {
        this.x = x;
        this.y = y;
    }
}
```

```
import Point from './Point.js';

const p1 = new Point(1, 2);
console.log( p1 ); // {x: 1, y: 2}
```

Jeśli nie potrzebujemy definiować domyślnego modułu, możemy pominąć słowo default i osobno wskazać elementy, które mają być eksportowane za pomocą samego słowa export.

Przy imporcie wykorzystujemy nawiasy klamrowe i wskazujemy na interesujące nas elementy.

Tutaj możemy użyć słowa kluczowego as, aby zmienić nazwę zmiennej, do której przypisujemy importowany element.

```
export const calcSum = (...args) => {
    return args.reduce( (acc, num) => acc + num, 0);
}
export const getMax = (...args) => {
    return Math.max(...args);
}
```

```
import { calcSum as getSum } from './math.js';

const sum = getSum(1, 2, 3, 5);
console.log( sum ); // 11
```

Zadania do wykonania

Pod poniższym adresem znajdziesz zadania,
które obejmują materiał prezentowany w niniejszym ebooku.

https://github.com/devmentor-pl/practice-js-es2015plus