Java Script by BC

console.log('dana do wprowadzenia w konsole'); - wyrzuca nam 5 w konsoli

Komentarz:

// komentarz pojedynczy
/* długi na kilka linijek komentarz
*/ - tego mozna również w srodku kodu uzywać

Typy danych(7):

numbers - 1, 12, 23.45
strings - zbiór liter lub/z cyframi, spacjami itp otoczone '...' lub podwójnymi "..." moga byc tez zdania
boolean - true/false (mozna myslec tak lub nie)
null - null bez quotes- symbolizuje intencjonalny brak wartości
undefined - podobnie jak null, ale nie do konca
symbol - bardziej zlozone kodowanie, unikalne identyfikatory
object - zbiór danych

matma
+ - * / %(remainder)
console.log(15 - 30);
concatenation;
+ mozna tez łączyc stringi w zdania; w tym mozna tez wrzucać zmienne i laczyc ze stringami
ale wtedy zmienne bez '', (tylko nazwa zmiennej + 'string' + '.')

Operator:

.length (wlasciwosc wartosci zmiennej)
.rozpoczyna prace na wczesniejszej zmiennej
console.log('Hello'.toUpperCase()); - to juz jest komnda wykonania kodu,zmienia wartosc zmiennej/wyglad
.trim usuwa whitespace/puste przestrzenie w wartości zmiennej
. otwieramy mozliwosc na wlasniwosci wartosci zmiennej i pracy na niej

dodawające operatory do ustalonej wczesniej zmiennej
+= -= *= /=

np.
var/let/const number = 100
number = number + 10
number += 10
console.log(number); output 120

deklaracje zmiennych KEYWORD:
var - przed es6(-2015)
let - zmienne ktore mozna potem modyfikowac/lokalna zmienna
const - zmienne ktorych nie mozna modyfikować

&{zmienna} - sposób na wrzucenie zmiennej do stringa
let name = bartek
console.log(`Hello &{name}`); wyrzuca hello bartek
wazny tez jest `w nawiasach`.

wartosc infinity - nieskonczona
NaN - brak liczby

Operatory binarne (typ boolean)

operatory logiczne (true/false) OPERATORY PORÓWNANIA/comparison
<, >, <=, >=
== - porównianie dwoch wartosci (ale bez typu)
=== - porównanie dwoch wartosci ale tez ich typu (string/number);bez dokonywania konwersji typów
&& - operator 'i' - jesli oba warunki sa prawdziwe
|| - operator 'lub' - jeden z nich musi byc ok ->true
Boolean(0,-0,"",null,undefined,Nan,) - te wyrzucą nam false
Boolean ("Jan") - true - sprawdza nam czy obiekt jest true/false
zamiast boolean mozna użyc podwójnego zaprzeczenia !! np. !!"bartek" -> true

! odwraca logike z true-> false (i na odwrót) (BANG OPERATOR eng.)
!= - takie porównanie będzie odwrotne (tak samo w przypadku !==)
np; if (!(2==='2')) -> true

Instrukcje warunkowe IF/else (conditional statements)(binary decisions)

0np:
```
var test;
if (true) {
        test = "działa";
}
else {
        test = "brak pasujących wyników"};
}
```

0dodatkowo mozna miedzy if a else wstawiac wiele else if i dodawać mozliwości

let stopLight = 'yellow';

```
if (stopLight === 'red') {
  console.log('Stop!');
} else if (stopLight === 'yellow') {
  console.log('Slow down.');
} else if (stopLight === 'green') {
  console.log('Go!');
} else {
  console.log('Caution, unknown!');
}
```

OPERATOR SWITCH

```
let athleteFinalPosition = 'first place';
switch (athleteFinalPosition) {
  case 'first place' :
    console.log('You get the gold medal!');
      break;
  case 'second place' :
    console.log('You get the silver medal!');
      break;
  case 'third place' :
    console.log('You get the bronze medal!');
      break;
    default:
      console.log('No medal awarded.');
      break;

}
```

0np:

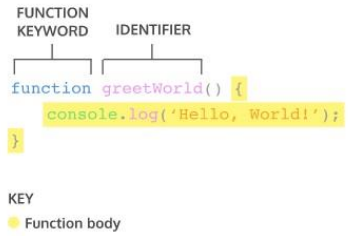
```
let defaultName;
if (username) {
  defaultName = username;
} else {
  defaultName = 'Stranger';
}
```

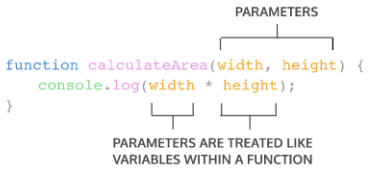
let defaultName = username || 'Stranger'; (JS bierze dane od lewej wiec username bedzie najpierw- jesli nie bedzie go to -> Stranger)

operator potrójny (trzy argumenty)
x ? y : v
warunek y-kod gdy prawdziwy v - kod do wykonania gdy jest false

```
name ? "witaj" + name : "witaj nieznajomy";
```

FUNCTION - mogący zostać użyty ponownie blok kodu (A *function* is a reusable block of code that groups together a sequence of statements to perform a specific task.);
deklaracja funkcji





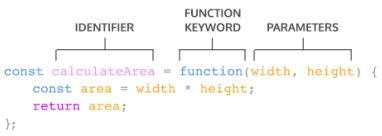
- To *call* a function in your code:

parametry w body działają jak zwykłe zmienne (ulegają np mnożeniu)

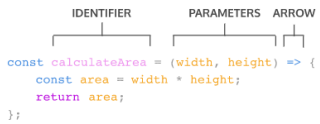
np; (default value function)
function greeting (name = 'stranger') {
  console.log(`Hello, ${name}!`)
}

greeting('Nick') // Output: Hello, Nick!
greeting() // Output: Hello, stranger!

return keyword - produkuje output, a więc wynik działania funkcji(do której był włożony jakis input); dzięki niemu widzimy wynik;



- To define a function using *arrow function notation*:



helper function - funkcja w funkcji; rozpatrywana od góry kodu.
np;
function monitorCount(rows, columns) {
  return rows * columns;
}

function costOfMonitors(rows, columns){
  return monitorCount(rows, columns) * 200;
}
 const totalCost = costOfMonitors(5, 4);
 console.log(totalCost); CONSOLE OUTPUT;4000

function expression

arrow function => (inny sposób utworzenia funkcji)

```
const plantNeedsWater = (day) => {
  if (day === 'Wednesday') {
    return true;
  } else {
    return false;
  }
};
```

Concise body (najbardziej zwięzła metoda tworzenia funkcji)

0 parameters

```
Const functionName = () => {};
```

1 parametr

```
Const functionName = paramOne => {};
```

2lub więcej parametrów

```
Const functionName = (paramOne, paramTwo) => {};
```

Body funkcji przy jednym bloku kodu można zamienic/skrócić w;

```
const squareNum = (num) => {

return num * num;

};
```

NA FORME KRÓTSZĄ -> const squareNum = num => num * num

HOISTING

Generalnie hoisting to taki mechanizm w języku JavaScript, który polega na tym, że silnik JS wyszukuje wszystkie deklaracje zmiennych (i funkcji) w danej funkcji i przenosi je na początek. Jeśli to zbyt zagmatwane to może przykład będzie bardziej pomocny:

```
function hoisting() { console.log(x); // undefined
  if (true) {
    var x = 1;
  }
```

SCOPE

(zasięg) idea dostępu/niedostępu zmiennych w kodzie.

Global variables – zmienne spoza bloku kodu(wchodzą do *global namespace*)

Block scope/local scope – zmienne pobierane z bloku kodu i tylko w nim działające

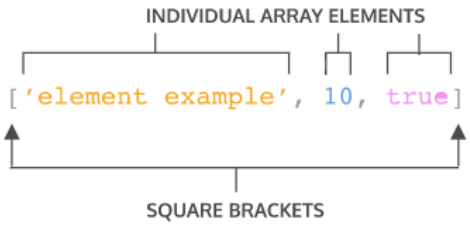
Jedne zmienne w bloku kodu, inne spoza bloku są brane.

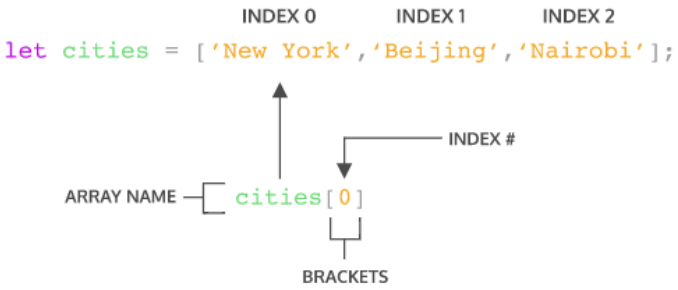
Scope pollution – przypadek gdy mamy zbyt wiele globalnych zmiennych lub gdy uzyjemy zmiennej z innego zasięgu(bloku)

Dobrą praktyką jest nie definiowanie zmiennych globalnie, poza blokiem kodu.

ARRAYS (tablice)

Tworzenie list w JS. Przechowują wszystkie typy danych. Kazda dana w tablicy ma przypisany numer porządkowy.





Update elementów tablicy:

```
let seasons = ['Winter', 'Spring', 'Summer', 'Fall'];

seasons[3] = 'Autumn';
console.log(seasons);
//Output: ['Winter', 'Spring', 'Summer', 'Autumn']
```

Wbudowana w array-je funkcja **.length**; podaje ilość elementów w tablicy.

```
const newYearsResolutions = ['Keep a journal', 'Take
a falconry class'];

console.log(newYearsResolutions.length);
// Output: 2
```

Funkcja **.push** – dodawanie do tablicy elementów(na jej końcu;

```
const itemTracker = ['item 0', 'item 1', 'item 2'];

itemTracker.push('item 3', 'item 4');

console.log(itemTracker);
// Output: ['item 0', 'item 1', 'item 2', 'item 3',
'item 4'];
```

.pop () – usuwa ostatni element tablicy.
.unshift() – dodaje nowy pierwszy element.
.indexOf() – szuka wartości w tablicy. The index to start the search at. If the index is greater than or equal to the array's length, -1 is returned, which means the array will not be searched. If the provided index value is a negative number, it is taken as the offset from the end of the array. Note: if the provided index is negative, the array is still searched from front to back. If the provided index is 0, then the whole array will be searched. Default: 0 (entire array is searched).

Tablice mogą być modyfikowane np. w funkcjach, które zmieniają te tablice (mutated!).
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array
Zagnieżdżone tablice (**nested**), kiedy tablica przechowuje inną tablicę.

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

To access the nested arrays we can use bracket notation with the index value, just like we did to access any other element:

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

console.log(nestedArr[1]); // Output: [2, 3]
```

Notice that `nestedArr[1]` will grab the element in index 1 which is the array `[2, 3]`. Then, if we wanted to access the elements within the nested array we can *chain*, or add on, more bracket notation with index values.

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

console.log(nestedArr[1]); // Output: [2, 3]
console.log(nestedArr[1][0]); // Output: 2
```

Zmienne zadeklarowane jako **const oraz zawierające tablice mogą być edytowane**(dane w tablicy). Tablica, której dane uległy zmianie w wyniku działania jakiejś funkcji stale zmienia te dane.

LOOP

Pętle! Iterate – powtarzać

A `for` loop contains three expressions separated by `;` inside the parentheses:

1. an *initialization* starts the loop and can also be used to declare the iterator variable.
2. a *stopping condition* is the condition that the iterator variable is evaluated against— if the condition evaluates to `true` the code block will run, and if it evaluates to `false` the code will stop.
3. an *iteration statement* is used to update the iterator variable on each loop.

The `for` loop syntax looks like this:

```
for (let counter = 0; counter < 4; counter++) {
  console.log(counter);
}
```

Do odejmowania używamy -- zamiast ++.
Wypluwanie po kolei każdego elementu tablicy:
const vacationSpots = ['Bali', 'Paris', 'Tulum'];

*for (let i = 0; i < vacationSpots.**length**; i++){*
console.log('I would love to visit ' + vacationSpots[i]);
}

Nested Loops

Zagnieżdżone pętle (nested loops): porównanie i wyszukiwanie w tablicach dwóch identycznych elementów oraz przeniesienie do nowej tablicy.

let bobsFollowers = ['Joe', 'Marta', 'Sam', 'Erin'];
let tinasFollowers = ['Sam', 'Marta', 'Elle'];
let mutualFollowers = [];

for (let i = 0; i < bobsFollowers.length; i++) {
for (let j = 0; j < tinasFollowers.length; j++) {
if (bobsFollowers[i] === tinasFollowers[j]) {
mutualFollowers.push(bobsFollowers[i]);
}
}
};

While Loops

While – podczas(gdy)

Increments – przyrost (++)

Losowanie parametru z tablicy:

const cards = ['diamond', 'spade', 'heart', 'club'];
let currentCard;

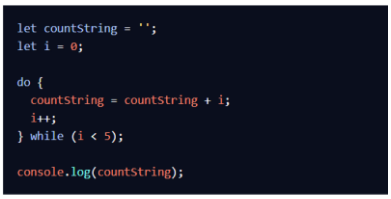
while (currentCard != 'spade') {

```
  currentCard = cards[Math.floor(Math.random() * 4)];
        console.log(currentCard);
}
```

Wyrzuca parametry tablicy, póki nie wyrzuci ‘spade’. Wtedy pętla się kończy. Funkcja math.random losuje za każdym razem jeden z elementów tablicy.

Do…While Loops

Wykonaj jakies zadanie raz a następnie idz dalej, póki wymóg nie zostanie spełniony.



```
let countString = '';
let i = 0;

do {
  countString = countString + i;
  i++;
} while (i < 5);

console.log(countString);
```

In this example, the code block makes changes to the `countString` variable by appending the string form of the `i` variable to it. First, the code block after the `do` keyword is executed once. Then the condition is evaluated. If the condition evaluates to `true`, the block will execute again. The looping stops when the condition evaluates to `false`.

Note that the `while` and `do...while` loop are different! Unlike the `while` loop, `do...while` will run at least once whether or not the condition evaluates to `true`.

Pętla wykonywana póki wymaganie jest true. Kiedy wymaganie(condition) zmienia się na false, pętla się zatrzymuje. W odróżnieniu do ***while loops***, ta pętla uruchomi się chociaż raz.

Do jest rozpatrywane najpierw, potem JS czyta stan wymagania ***while***, jeśli jest *false* – pętla zamyka się. Tym różni się ***do…while*** od zwykłej ***while loops*** – pętla chociaż raz zostanie wykonana.

Break Keyword

Break keyword powoduje zakończenie pętli nawet jeśli wymaganie(condition) nie zostało spełnione.

```
const rapperArray = ["Lil' Kim", "Jay-Z", "Notorious B.I.G.", "Tupac"];

for (let i = 0; i < rapperArray.length; i++){
  console.log(rapperArray[i]);
  if (rapperArray[i] === 'Notorious B.I.G.'){
    break;
  }
}

console.log("And if you don't know, now you know.");
```

Jeśli na konsoli wyskoczy natorius big to nastepuje break i koniec pętli. Następnie dodawany jest strinh *„and if you dont…”*

Higher-order Function

Funkcje które akceptują/rozumieją inne funkcje jako argumenty lub/oraz wyrzucają jako wynik funkcje. Budowanie abstrakcji na innej abstrakcji. Cos rozumie się przez coś innego. Np.; urządzamy urodziny -> upiekliśmy ciasto. (cos na zasadzie skojarzeń). Używanie ***higher-order function*** powoduje że kod jest prosty w czytaniu oraz debugowaniu.

Funkcje jako dane

Możemy funkcji, która już została opisana zmienną, nadać inną zmienną (np. jeśli nazwa pierwotnej zmiennej opisującej funkcje jest za długa).

W **JS funkcje to obiekty klasy pierwszej**. Mogą mieć metodę(np.: *.toString())* i właściwości(np. *.length, .name*). **Funkcje są obiektami!**

Właściwość .name podaje nam wartość zmiennej(w tym wypadku „prawdziwą” nazwę zmiennej).

Funkcje jako parametry

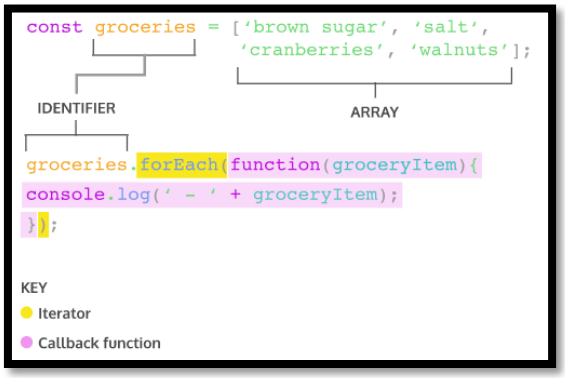
Higher-order function to funkcje które akceptują funkcje jako parametry(w sobie) oraz zwracają wynik działania funkcji jako funkcja lub parametr.

Iterators

Iterator można rozumieć jako **rodzaj wskaźnika udostępniającego dwie podstawowe operacje**: **odwołanie się do konkretnego elementu** w tablicy (dostęp do elementu) oraz **modyfikację samego iteratora** tak, by **wskazywał na kolejny element** (sekwencyjne przeglądanie elementów). Musi także istnieć sposób utworzenia iteratora tak, by wskazywał na pierwszy element, oraz sposób określenia, kiedy iterator wyczerpał wszystkie elementy w kolekcji. W zależności od języka i zamierzonego zastosowania iteratory mogą dostarczać dodatkowych operacji lub posiadać różne zachowania.

Podstawowym celem iteratora jest pozwolić użytkownikowi przetworzyć każdy element w kolekcji bez konieczności zagłębiania się w jej wewnętrzną strukturę. Pozwala to kolekcji przechowywać elementy w dowolny sposób, podczas gdy użytkownik może traktować ją jak zwykłą sekwencję lub listę. Klasa iteratora jest zwykle projektowana wraz z klasą odpowiadającej mu kolekcji i jest z nią ściśle powiązana. Zwykle to kolekcja dostarcza metod tworzących iteratory.

Iteration method(iterators) – metody na powtarzanie przeszukiwania tablic, manipulowania jej elementami oraz zwracania wartości wykonanej metody(w funkcji). Przykładowe metody:



.forEach()

```
const artists = ['Picasso', 'Kahlo', 'Matisse', 'Utamaro'];

artists.forEach(artist => {
  console.log(artist + ' is one of my favorite artists.');
});
```

-zza każdy element w tablicy artist, wykonaj działanie artist+string.

.map()

```
const numbers = [1, 2, 3, 4, 5];
const squareNumbers = numbers.map(number => {
  return number * number;
});
console.log(squareNumbers);
```

*- wykonuje przeszukanie tablicy, wykonuje na niej działanie i **zwraca nową tablicę**! Nie zmienia ‘starej’ tablicy.*

.filter()

```
const things = ['desk', 'chair', 5, 'backpack', 3.14, 100];
const onlyNumbers = things.filter(thing => {
  return typeof thing === 'number';
});
console.log(onlyNumbers);
```

*-filtruje po danym typie danych w tablicy i je wyrzuca, tutaj należy np.;również użyć w bloku kodu **typeOf/ ifelse/ <.Jakieś zależności filtrującej tablice za zasadzie (true/false). Również zwraca nową tablice**.*

.join() – zmienia w łańcuch znaków wszystkie elementy tablicy i łączy je w jeden łańcuch znaków bez przecinków czy ‘.

.findIndex()

Przeszukiwanie tablicy(array) w poszukiwaniu wartości która zwroci nam *true* w funkcji callback.

Jeśli funkcja findIndex nie znalazła szukanego argumentu, zwraca -1. Jeśli znajdzie zwrac w formie cyfry, która jest numerem w indeksie tablicy. Np.; zwraca 2, oznacza to, że szukany argument jest na miejscu 3 w tablicy.

Jeśli szukam w tablicy nazwy która zaczyna się np. na s; nawiasy kwadratowe do oznaczenia czego metoda .findIndex ma szukać. Tutaj szuka stringa który zaczyna się[0] na literę s.

```
const startsWithS = animals.findIndex(sname => {
  return sname[0]=== 's';
});
```

Reduce()

Zwraca pojedynczą wartość po iteracji tablicy. Wymagane są argumenty *accumulator* oraz *currentValue* do zliczania parametrów tablicy. 100 to drugi argument, który jest brany przez JS jako pierwsza wartość w tablicy.

```
const numbers = [1, 2, 4, 10];

const summedNums = numbers.reduce((accumulator,
currentValue) => {
  return accumulator + currentValue
}, 100)  // <- Second argument for .reduce()

console.log(summedNums); // Output: 117
```

Here's an updated chart that accounts for the second argument of `100`:

Iteration #	accumulator	currentValue	return value
First	100	1	101
Second	101	2	103
Third	103	4	107
Fourth	107	10	117

const cities = ['Orlando', 'Dubai', 'Edinburgh', 'Chennai', 'Accra', 'Denver', 'Eskisehir', 'Medellin', 'Yokohama'];

```
const word = cities.reduce((acc, currVal) => {
  return acc + currVal[0]
}, "C");
console.log(word)
//output: CODECADEMY
```

Pozostałe metody iteracji tablic:

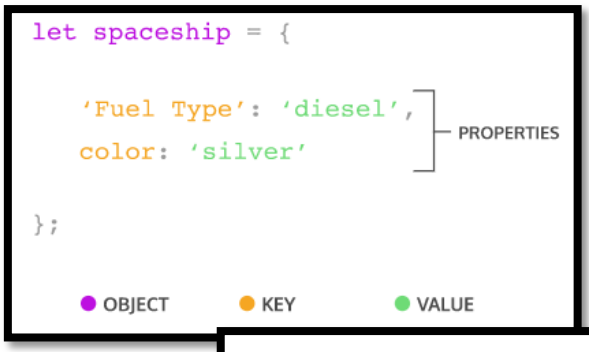
W linku znajdują się dokumentacje opisujące te metody, składnie metod oraz pozostałe szczegóły.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array#Iteration_methods

.some() –czy jakiś element tablicy spełnia argumenty funkcji. Zwracany jest **boolean true/false.**

.every() - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/every

Sprawdza czy każdy element tablicy spełnia argument(żądanie).

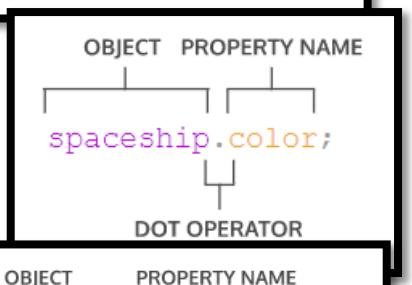


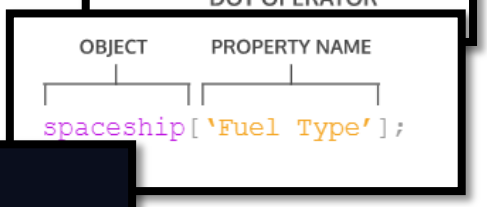
Objects

Wstęp do obiektów.

By wydobyć(access) wartość z obiektu:

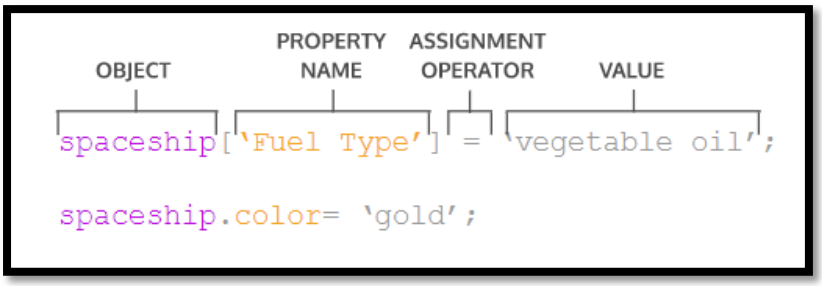
1. Metoda .dot (**dot notation**)
   a. Jeśli nie ma danej wartości, konsola zwraca undefined.
   b. **Bracket notation** – []. **Musimy używać tej notacji przy numerach, spacjach oraz symbolach w kluczach obiektu.**
   c. **Jako wyciągnie danych w funkcji:**






```
let returnAnyProp = (objectName, propName) =>
objectName[propName];

returnAnyProp(spaceship, 'homePlanet'); // Returns
'Earth'
```

Dodawanie 'properties' do obiektu lub ich edycja:



Jeśli była już taka wartość w obiekcie, zostanie nadpisana. Jeśli jej nie było, zostanie utworzona. Można używać notacji . lub []. Nie można zmieniac const, lecz można zmienić jego wartość.

Usuwanie properties z obiektu operatorem delete:

```
const spaceship = {
  'Fuel Type': 'Turbo Fuel',
  homePlanet: 'Earth',
  mission: 'Explore the universe'
};

delete spaceship.mission;  // Removes the mission
property
```

Metody: (.log i inne)

let retreatMessage = 'We no longer wish to conquer your planet. It is full of dogs, which we do not care for.';

```
let alienShip = {
  retreat() {
    console.log(retreatMessage)
  },
  takeOff() {
    console.log('Spim... Borp... Glix... Blastoff!')
  }
};
alienShip.retreat(); //inwokacja funkcji z obiektem output;tekst z retreatmessage
alienShip.takeOff(); //inwokacja funkcji z obiektem spim borp..
```

Nested objects: oczywiście można w obiektach zawierac inne obiekty.

Wyciąganie properties z obiektu: JS czyta od lewej do prawej wiec idziemy od ogółu do szczegółu.

```
spaceship.nanoelectronics['back-up'].battery; //
Returns 'Lithium'
```

Oraz trochę inna metoda odczytywania danych:
let capFave = spaceship.crew.captain['favorite foods'][0];

In the preceding code:

- First the computer evaluates `spaceship.nanoelectronics`, which results in an object containing the `back-up` and `computer` objects.
- We accessed the `back-up` object by appending `['back-up']`.
- The `back-up` object has a `battery` property, accessed with `.battery` which returned the value stored there: `'Lithium'`

Pass by reference: Objects are passed by reference. This means when we pass a variable assigned to an object into a function as an argument, the computer interprets the parameter name as pointing to the space in memory holding that object. As a result, functions which change object properties actually mutate the object permanently (even when the object is assigned to a const variable).

Funkcje zmieniające properties w obiekcie properties=key+value. Np. name: ‘John Malkovic’,

```
let spaceship = {
  'Fuel Type' : 'Turbo Fuel',
  homePlanet : 'Earth'
};

let greenEnergy = (obj) => {
  obj['Fuel Type'] = 'avocado oil'
}; //js zmienia fuel type na avocado oil.
```

Wywołanie funkcji na obiekcie:
greenEnergy(spaceship) //wtedy dopiero funkcja jest wykonywana!
Następnie jeśli chcemy wrzucić w koncole to console.log(spaceship);

Looping through objects (for…in):

Pętle to narzedzie programistyczne, które powtarza blok kodu póki wymaganie(condition) nie zostanie spełnione. Jednak key-value w obiektach nie są numerowane tak jak tablice wiec dlatego mamy składnie **js for…in**.

```
let spaceship = {
   crew: {
   captain: {
      name: 'Lily',
      degree: 'Computer Engineering',
      cheerTeam() { console.log('You got this!') }
      },
   'chief officer': {
      name: 'Dan',
      degree: 'Aerospace Engineering',
      agree() { console.log('I agree, captain!') }
      },
   medic: {
      name: 'Clementine',
      degree: 'Physics',
      announce() { console.log(`Jets on!`) } },
   translator: {
      name: 'Shauna',
      degree: 'Conservation Science',
      powerFuel() { console.log('The tank is full!') }
      }
   }
};

for (let crewMember in spaceship.crew) {
  console.log(`${crewMember}: ${spaceship.crew[crewMember].name}`)
};

for (let crewMember in spaceship.crew) {
  console.log(`${spaceship.crew[crewMember].name}: ${spaceship.crew[crewMember].degree}`)
};

//Output:
captain: Lily
chief officer: Dan
medic: Clementine
translator: Shauna
Lily: Computer Engineering
Dan: Aerospace Engineering
Clementine: Physics
Shauna: Conservation Science
```

Metoda .this(keyword)

Odnosi się do obiektu w którym wykonywana jest funkcja, wtedy konsola nie wyrzuca referenceError.

```
const robot = {
  model: '1E78V2',
  energyLevel: 100,
 provideInfo() {
   return `I am ${this.model} and my current energy level is ${this.energyLevel}.`
 }
};
console.log(robot.provideInfo());
```

Przy arrow function nie używamy .this! wtedy funkcja szuka globalnej zmiennej, która w tym przypadku jest lokalna(tylko w danym obiekcie). Wiec zwraca undefined.

Privacy

W ***js*** nie można blokować properties obiektu aby nie mogły być edytowalne. Można za do dac znac innym czytelnikom kodu, ze obiekt nie powinien być manipulowany.

key : value – przy takim oznaczeniu mamy informacje ze ta wlasciwosc nie powinna być zmieniana, lecz jest to możliwe.

Getters & Setters

- Getters can perform an action on the data when getting a property.
- Getters can return different values using conditionals.
- In a getter, we can access the properties of the calling object using this.
- The functionality of our code is easier for other developers to understand.

```
const robot = {
 _model: '1E78V2',
 _energyLevel: 100,
 get energyLevel(){
   if(typeof this._energyLevel === 'number') {
     return 'My current energy level is '
+ this._energyLevel
   } else {
     return "System malfunction:
cannot retrieve energy level"
   }
 }
};
console.log(robot.energyLevel);
output; My current energy level is 100
```

Setters

Możemy sprawić żeby do danego key była przypisywana wartość, lub typ danej jaki chcemy. W tym przypadku musi to być liczba. Pozwala oczywiście również na zmienianie tej wartości.

```
const person = {
  _age: 37,
  set age(newAge){
    if (typeof newAge === 'number'){
      this._age = newAge;
    } else {
      console.log('You must assign a number to age');
    }
  }
};
```

Notice that in the example above:

- We can perform a check for what value is being assigned to `this._age`.
- When we use the setter method, only values that are numbers will reassign `this._age`
- There are different outputs depending on what values are used to reassign `this._age`.

Można dodać kolejny warunek w body if;a pisząc && oraz kolejny wymóg. Przykład użycia w jednym obiekcie *setters* oraz *getters*. Pomagają nam przy zmienianiu danych w obiekcie, wyrzucając np. wymog wpisania wartości danego typu np. numer.

```
const robot = {
  _model: '1E78V2',
  _energyLevel: 100,
  _numOfSensors: 15,
  get numOfSensors(){
    if(typeof this._numOfSensors === 'number'){
      return this._numOfSensors;
    } else {
      return 'Sensors are currently down.'
    }
  },
  set numOfSensors(num){
    if(typeof num === 'number' && num >= 0)
      return this._numOfSensors = num;
    else 'Pass in a number that is greater than or equal to 0';
  }
};

robot.numOfSensors = 100;
console.log(robot.numOfSensors);
robot._numOfSensors = 'broke';
console.log(robot.numOfSensors);

//output:
100
Sensors are currently down.
```

Factory function

Funkcje fabryczne ;)

```
const monsterFactory = (name, age, energySource,
catchPhrase) => {
  return {
    name: name,
    age: age,
    energySource: energySource,
    scare() {
      console.log(catchPhrase);
    }
  }
};
```

```
const ghost = monsterFactory('Ghouly', 251,
'ectoplasm', 'BOO!');
ghost.scare(); // 'BOO!'
```

Destructuring (property value shorthand)

Wprowadzone w es6 przypisywanie właściwości do zmiennych w skróconej formie:

```
const monsterFactory = (name, age) => {
  return {
    name,
    age
  }
};
```

Destructured Assignment

Wyciąganie property z obiektu jako zmienna oraz wywoływanie funkcji z obiektu do którego jest przypisana dana wartość(property):

W tym przypadku ustawiliśmy zmienna functionality oraz wywolalismy jej funkcje beep, który konsola wyrzucila nam jako *beep boop.*

```
const robot = {
  model: '1E78V2',
  energyLevel: 100,
  functionality: {
    beep() {
      console.log('Beep Boop');
    },
    fireLaser() {
      console.log('Pew Pew');
    },
  }
};

const { functionality } = robot;
functionality.beep();
```

Możemy tez używać wbudowanych metod na obiektach(nie tylko tych które sa napisane w obiekcie jak właśnie *beep*()).

Object.keys(tutaj nazwa obiektu) *//ta metoda wyrzuca nam w konsoli wszystkie names/keys (np.; model, EnergyLevel jak nazdjeciu obok).Wyrzuca dane jako tablica*

Object.entries(tutaj nazwa obiektu) *//metoda do wyrzucania w konsoli tablicy z key-value.*

Object.assign() ***//dodawanie do istniejącego obiektu nowych elementów.***

Podsumowanie obiektów:

Calling object to metoda przypisana w obiekcie. Keyword this odnosi się do calling object. Metody nie maja automatycznego dostępu do wewnętrznych danych w calling object.

```
const robot = {
  model: 'SAL-1000',
  mobile: true,
  sentient: false,
  armor: 'Steel-plated',
  energyLevel: 75
};

// What is missing in the following method call?
const robotKeys = Object.keys(robot);

console.log();

// Declare robotEntries below this line:
const robotEntries = Object.entries(robot)
console.log();

// Declare newRobot below this line:
const newRobot = Object.assign({laserBlaster: true, voiceRecognition: true}, robot);

console.log(newRobot);
```

Classes

Narzedzie do produkowania na szybko podobnych względem siebie obiektów. Mogą sluzyc jako swoiste templates(podkładki).

Do tworzenia klas potrzebna jest metoda **constructor(name).**

```
class Dog {
  constructor(name) {
    this.name = name;
    this.behavior = 0;
  }
}
```

- `Dog` is the name of our class. By convention, we capitalize and CamelCase class names.
- JavaScript will invoke the `constructor()` method every time we create a new instance of our `Dog` class.
- This `constructor()` method accepts one argument, `name`.
- Inside of the `constructor()` method, we use the `this` keyword. In the context of a class, `this` refers to an instance of that class. In the `Dog` class, we use `this` to set the value of the Dog instance's `name` property to the `name` argument.
- Under `this.name`, we create a property called `behavior`, which will keep track of the number of times a dog misbehaves. The `behavior` property is always initialized to zero.

Klasa dog z jednym parametrem do uzupełnienia(name). Poniżej klasa surgeon z dwoma parametrami do uzupełniania przy tworzeniu nowego obiektu(name, department).

```
class Surgeon {
  constructor(name, department) {
    this._name = name;
    this._department = department;
  }
}
```

Dodawanie instancji – tworzenie dzięki klasie nowego obiektu przy użyciu **keyword new**.

```
class Dog {
  constructor(name) {
    this.name = name;
    this.behavior = 0;
  }
}

const halley = new Dog('Halley'); // Create new Dog instance
console.log(halley.name); // Log the name value saved to halley
// Output: 'Halley'
```

Getters i Setters w klasach można używać tylko bez przecinków, przy budowaniu składni! Przy gettersach dodac **return**.

Przykład budowania klasy z gettersami i jedną metodą:

```
class HospitalEmployee {
  constructor (name) {
    this._name = name;
    this._remainingVacationDays = 20;
  }
  get name() {
    return this._name;
  }
  get remainingVacationDays() {
    return this._remainingVacationDays;
  }
  takeVacationDays(daysOff) {
    this._remainingVacationDays -= daysOff;
  }
}
```

Przy metodach w klasach nie dodajemy return.

Inheritance – dziedziczenie parent/child classes

Dzielenie się properties (key-value) z dziecmi -> **shared properties** -> ***keyword extends oraz super***

Tworzenie podklasy(child) cat w klasie(parents) animal:

```
class Cat extends Animal {
  constructor(name, usesLitter) {
    super(name);
    this._usesLitter = usesLitter;
  }
}
```

Super keyword wywoluje klasę rodziców, wydaje im informacje o zmianach w podklasie Cat.

Extends keyword wrzyca metody metod animal class do podklasy cat oraz oczywiście properties(key-value).

Child classes mogą mieć również swoje własne,nowe properties(właściwości).

Do podklas można tez **dodawać nowe wartości** przy pomocy metody **.push**(opisana w innym rozdziale):

addCertification(newCertification) {

this._certifications.push(newCertification);

w tym wypadku dodajemy nowe certyfikaty pielęgniarce ;)

nurseOlynyk.addCertification('Genetics');

console.log(nurseOlynyk.certifications);

```
class Nurse extends HospitalEmployee {
  constructor(name, certifications) {
    super(name);
    this._certifications = certifications;
  }
  get certifications() {
    return this._certifications;
  }
  addCertification(newCertification) {
    this._certifications.push(newCertification);
  }
}

const nurseOlynyk = new Nurse('Olynyk',
['Trauma','Pediatrics']);
nurseOlynyk.takeVacationDays(5);
console.log(nurseOlynyk.remainingVacationDays);

nurseOlynyk.addCertification('Genetics');
console.log(nurseOlynyk.certifications);
```

```
class Animal {
  constructor(name) {
    this._name = name;
    this._behavior = 0;
  }

  static generateName() {
    const names = ['Angel', 'Spike', 'Buffy',
'Willow', 'Tara'];
    const randomNumber = Math.floor(Math.random()*5);
    return names[randomNumber];
  }
}
```

Static Method (metody statyczne)

.generateName() //metoda statyczna. Działa jednorazowo przy wywolaniu, generuje losowe imie. Nie można ich wywolywac np. przy console.log. może się laczyc tylko z daną klasą np.: Animal(parent class).

Keyword STATIC //mówimy *JS* ze odpalamy metode statyczną, w tym wypadku generateName().

.generatePassword() – generuje haslo. Jeśli ma być losowe to używamy metody ze zmiennej randomNumber powyżej (losowa liczba do 5 w tym wypadku).

Jeśli ma być tylko sama losowa liczba to usuwamy zmienną *names* oraz zwiększamy liczbę możliwych opcji(zamiast 5 -> 10000np). następnie nie robimy z tego zmiennej tylko od razu return i skladnie math.floor….:

```
static generatePassword() {

   return Math.floor(Math.random() * 10000);

 }

}
```

Podsumowanie:

Klasy są podkładkami dla obiektów(szkieletem). Czyms co można uzupelnic danymi, ozywic przy tworzeniu obiektów, które będą miały utworzony od razu szkielet. Kiedy chcemy stworzyć klase używamy *constructora*(metody).

Dziedziczenie wystepuje gdy tworzymy klasę rodzica z metodami oraz wlasciwosciami(properties) które następnie dziedziczą podklasy(child classes) z identycznymi wlasciwosciami oraz metodami. Aby utworzyć podklasę(child) używamy keyword extends. Keyword super wywoluje konstruktora klasy parent aby można było korzystać z jego danych. Metody statyczne używamy na klasie tylko i wyłącznie.

Browser compatibility & trasnpilers (ES5/6)

Transpilers to transpilatory(przetwarzacze), narzędzia które czytają kod zrodlowy w danym jezyku a następnie zmieniają na inny język. Mogą zmieniac tez kod js na starszą wersję tego jezyka przy pomocy np. biblioteki babel.

ECMAScript2015 wprowadzila wiele poprawek oraz nowych składni, które mogą wyprzeć stare:

Const, **let** oraz nowy styl wprowadzania stringów(**${zmienna}**). Ecma international odpowiada za standaryzację JS. Nowości podnosza czytelność kodu oraz latwosc pisania, przy czym sa po prostu krótsze skladnie. Nowe skladnie eliminują wystepujace w ES5 bugi. Nowości przyblizyly JS do innych zorientowanych obiektowo jezyków – latwiej jest wejść w ten jezyk z innego. Oczywiście wystepuje wsteczna kompatybilność.

Dwa ważne narzędzia do sprawdzania problemów z kompatybilnością strony – stara wersja JS ES5 vs. ES6:

Caniuse.com – do sprawdzenia jakie przeglądarki np. rozpoznają ‘let’.

Babel – biblioteka JS do konwertowania nowego Js do wersji starszej es5(która jest rozpoznawalna przez większość przeglądarek.

Istalacja środowiska uruchomieniowego:

https://kursjs.pl/kurs/es6/webpack.php

oraz **Node.js**

Modules (moduły)

Kawałki kodu, schowane w pliku, które mogą być wykorzystywane wielokrotnie oraz można je eksportować z jednego programu do drugiego. Pomagają przy:

- Szukaniu, naprawianiu i debugowaniu kodu;
- Ponowne użycie oraz cykl zdefiniowanego zachowania logicznego(fragment kodu) w różnych częściach aplikacji;
- Trzymają bezpieczeństwo i chronią przed innymi modułami;
- Zapobiegają *global namespace pollution*(za dużo zdefiniowanych zmiennych) oraz potencjalnej kolizji nazewnictwa(zmienne);

Tworzenie modułu przy użyciu składni node.js – module.exports.

Tworzenie obiektu, a następnie eksportowanie go tworząc z niego moduł(cześć kodu która może być importowana w dowolnej chwili, w potrzebie:

```
let Menu = {};
Menu.specialty = "Roasted Beet Burger with Mint Sauce";

module.exports = Menu;
```

Tworzymy klasyczny obiekt, przypisujemy do niego properties(key-value).

Nastepnie używamy zmiennej module oraz export który tworzy z obiektu moduł.

Require() function – importowanie modułu do innego pliku JS:

```
const Menu = require('./menu.js');

function placeOrder() {
  console.log('My order is: ' + Menu.specialty);
}

placeOrder();
```

tworzymy dowolnie nazwana zmienna, a następnie wywołujemy funkcje **require** oraz podajemy ścieżkę dostępu do modułu który chcemy zaimportować. Teraz możemy utworzyć np. nowa funkcje która korzysta z właściwości(menu.speciality) importowanego modułu. Następnie odpalamy funkcje aby wyrzuciło nam stringa z funkcji.

Można tez od razu w console.log zawrzeć wywołanie funkcji:

```
console.log(Airplane.displayAirplane());
```

Dzięki es6 JS wspiera nowe składnie importujące/exksportujace moduły – **export default** oraz **named exports.**

export Default – eksportuje jeden moduł na plik.

```
let Menu = {};

export default Menu;
```

ES6 posiada również **Keyword import.**

```
import Menu from './menu';
```

missionControl.js

```
let Airplane = {
  availableAirplanes: []

};

Airplane.availableAirplanes = [
  {
    name: 'AeroJet',
    fuelCapacity: 800
  },
  {
    name: 'SkyJet',
    fuelCapacity: 500
  }
];

export default Airplane;
```

airplane.js

```
import Airplane from './airplane';

function displayFuelCapacity() {

 Airplane.availableAirplanes.forEach(function(element) {
    console.log('Fuel Capacity of ' + element.name + ': ' + element.fuelCapacity);
  });
}


displayFuelCapacity();
```

//Output:

Fuel Capacity of AeroJet: 800

Fuel Capacity of SkyJet: 500

Udało się importować dane z innego folderu. Dzięki funkcji forEach JS przejrzal tablice i utworzyl z nich stringa dla każdego elementu tablicy – dla aero jeta utworzyl string oraz skyjeta.

```
let specialty = '';
function isVegetarian() {
};
function isLowSodium() {
};

export { specialty, isVegetarian };
```

Named export

Named export może eksportować nawet pojedyncze funkcje jako zmienne.

Tutaj eksportował zmienna stringa jako obiekt oraz funkcje jako obiekt – w rzeczywistości przy importowaniu sa de facto ***function object.***

Może być także eksportowany od razu po zadeklarowaniu, jeśli na początku dodamy export:

Te eksportowane możemy w prosty sposób importować jako zmienne:

```
import { specialty, isVegetarian } from 'menu';
```

```
export let specialty = '';
export function isVegetarian() {
};
function isLowSodium() {
};
```

Named Import

```
import { specialty, isVegetarian } from './menu';

console.log(specialty);
```

Aby importować, wybieramy w body importu nazwy zmiennych(pod którymi kryja się również funkcje oraz properties). Następnie wskazujemy na zrodlo z którego pliku maja zostać pobrane.

Export As/ Import As

```
let specialty = '';
let isVegetarian = function() {
};
let isLowSodium = function() {
};

export { specialty as chefsSpecial, isVegetarian as
isVeg, isLowSodium };
```

pozwala zmienić przy eksporcie nazwę zmiennej. (łał) używając keyword'a **as**.

Importowanie wymaga podania zmienionych nazw zmiennych!

```
import { chefsSpecial, isVeg } from './menu';
```

Podsumowanie: funkcje **export** czy **export default** można łączyć, używają ich pod sobą w razie konieczności. Nie jest to jednak dobrą praktyką. To samo tyczy się importów. Moduły to mogące być ponownie użyte kawałki kodu zaimportowane z innego pliku:

Moduły z ES5:

- **Module.exports** eksportuje moduł do użycia w innym programie;
- **Require**() importuje moduł do użycia w bieżącym programie;

Moduły z Es6 to:

- **Export default** – eksportuje obiekty, funkcje czy podstawowe dane w JS;
- **Export** – eksportuje dane jako zmienna;
- Można tez zmieniać nazwy eksportowanych zmiennych używając keyword;a as.
- **Import** – importuje obiekty, funkcje i inne typy danych.

Promises

Temat związany jest z asynchronicznym kodowaniem – program wykonując jakies duże zadanie pozwala nam w międzyczasie na mniejsze działania/procesy w kodzie.

Obietnice. Sa to obiekty reprezentujące ewentualny wynik asynchronicznej operacji. Wprowadzone w tej postaci w ES6. Nie wiemy czy obietnica zostanie spełniona, ale na pewno dowiemy się kiedyś czy uda się ja wykonać czy tez nie. Obietnice maja trzy stany:

- Pending – oczekujące na rozwiązanie;
- Resolve – obietnica spelniona;
- Rejected – cos poszlo nie tak

Wszystkie obietnice pozwalają na wykonanie ***logic*** czyli napisana co dalej jeśli się udało oraz jeśli się nie udało.

```
const inventory = {
  sunglasses: 1900,
  pants: 1088,
  bags: 1344,
  vodka: 5
};

// Write your code below:
const myExecutor = (resolve, reject) => {
  if(inventory.vodka > 0) {
    resolve('Vodka has been drank.');
  } else {
    reject('There is no vodka in the pocket.');
  }
}

function drinkVodka () {
  return new Promise(myExecutor);
}

const orderPromise = drinkVodka();
console.log(orderPromise);
```

tworzenenie obiektu pocket. Następnie funkcji myExecutor która jako parametry posiada **resolve** oraz **reject** – w przypadku niewykonania(lub nie spełnieniu argumentu).

Resolve Reject zawsze jako parametry!

Następnie utworzenie funkcji drinkVodka w celu wywołania obietnicy która używa my Executor.

Następnie utworzenie zmiennej orderPromise która jest defacto funkcja drinkVodka.

Następnie wywołanie w konsoli tej zmiennej, która wykorzystuje kod napisany wyżej.

Jeśli jest ok, to mamy resolve i w consoli wyskakuje ze vodka has been drank ;)

setTimeout(callback, delay in miliseconds) – jest to API node. Dwa parametry callback oraz delay zawsze musza być. Call back jest funkcją która zostanie włożona do kolejki po upływie delay, który sami ustalamy.

.then() - I have a promise, when it settles, **then** here's what I want to happen…

Początkowy stan asynchronicznej obietnicy jest pending(wyslane), następnie będzie albo rejected albo fullfilled. Następnie do gry wchodzi wyzsza funkcja .THEN: obojętnie jak zostanie rozpatrzona obietnica, then kontynuuje działanie w określony przez nas sposób. Jak wyżej, ma dwie callback function jako argumenty:

Wtedy gdy jest rozpatrzona pozytywnie(mowi się **onFulfilled**) – success handler i ustawiamy logike działania na pozytywną.

Oraz rozpatrzona obietnica negatywnie (**onRejected**) – failure handler; wtedy ustawiamy logike kodu jako negatywną. Then zawsze zwraca obietnice. Składnia:

```
const prom = new Promise((resolve, reject) => {
  resolve('Yay!');
});

const handleSuccess = (resolvedValue) => {
  console.log(resolvedValue);
};

prom.then(handleSuccess); // Prints: 'Yay!'
```

```
let prom = new Promise((resolve, reject) => {
  let num = Math.random();
  if (num < .5 ){
    resolve('Yay!');
  } else {
    reject('Ohhh noooo!');
  }
});

const handleSuccess = (resolvedValue) => {
  console.log(resolvedValue);
};

const handleFailure = (rejectionReason) => {
  console.log(rejectionReason);
};

prom.then(handleSuccess, handleFailure);
```

U góry promise bez callback reject funkcji. Po boku pełny promise z funkcja wyzsza then oraz callback resolve oraz reject.

metoda math.random wywoluje nam losową liczbę ;)

Catch() - funkcja *catch* bierze tylko jeden argument – onRejected. A wiec odrzucona obietnice. Wiec w przypadku odrzucenia wykonuje się kod z catch'a. używanie catcha jest prawie tym samym co używanie then.

```
prom
  .then((resolvedValue) => {
    console.log(resolvedValue);
  })
  .catch((rejectionReason) => {
    console.log(rejectionReason);
  });
```

Composition – łączenie obietnic; jedna po drugiej, zależne od siebie. Nie wolno robić nested promises, zagnieżdżonych w sobie, powinny się łączyć pod sobą a nie „wchodzić" w siebie. Należy pamiętać o keyword return, aby wywołana obietnica zwracała wartość na której będzie bazowała następna obietnica.

```
const {checkInventory, processPayment, shipOrder} =
require('./library.js');

const order = {
  items: [['sunglasses', 1], ['bags', 2]],
  giftcardBalance: 79.82
};

// Refactor the code below:

checkInventory(order)
.then((resolvedValueArray) => {
  return processPayment(resolvedValueArray);
})
.then((resolvedValueArray) => {
  return shipOrder(resolvedValueArray);
})
.then((successMessage) => {
  console.log(successMessage);
});
```

Wzorcowa kompozycja połączonych (chain) obietnic, które oddziałują na siebie.

```
let myPromises = Promise.all([returnsPromOne(),
returnsPromTwo(), returnsPromThree()]);

myPromises
  .then((arrayOfValues) => {
    console.log(arrayOfValues);
  })
  .catch((rejectionReason) => {
    console.log(rejectionReason);
  });
```

Promise.all() – funkcja nakazującą wykonać wszystkie obietnice w tym samym momencie (w losowej kolejności).

```
const {checkAvailability} = require('./library.js');

const onFulfill = (itemsArray) => {
  console.log(`Items checked: ${itemsArray}`);
  console.log(`Every item was available from the distributor. Placing order now.`);
};

const onReject = (rejectionReason) => {
  console.log(rejectionReason);
};

// Write your code below:

const checkSunglasses = checkAvailability('sunglasses', 'Favorite Supply Co.');
const checkPants = checkAvailability('pants', 'Favorite Supply Co.');
const  checkBags = checkAvailability('bags', 'Favorite Supply Co.');

Promise.all([checkSunglasses, checkPants, checkBags])
  .then(onFulfill)
  .catch(onReject);
```

Podsumowanie:

- Promises to obiekty JS które reprezentują rezultat działań asynchronicznych;
- Promises mogą być w trzech stanach: pending, resolved lub rejected;
- Promise jest settled jeśli została wypełniona lub odrzucona
- Aby utworzyć korzystamy z keyword **new Promise**;
- **setTimeout()** jest funkcja Node która opóźnia wykonanie funkcji callback;
- używamy **.then()** jako success handler callback, który zawiera dalsze instrukcja w skutek wykonania obietnicy;
- używamy .catch() dla failure handler callback, który zawiera dalsze instrukcje w skutek niewykonania obietnicy;
- możemy łączyć (chain) obietnice w **composition**, tworząc łańcuch działań asynchronicznych. Używamy do tego .then oraz .catch, poprzez wielorazowe użycie;
- powinniśmy chain'ować zamiast zagnieżdżać obietnice;
- możemy poprzez **Promise.all()** odpalić wszystkie obietnice w kodzie w losowej kolejności(ten sam moment wykonywania).

Async Await

Skladnia okreslana jako syntatic sugar – nie zmienia generalnego działania, lecz zostaje wprowadzona aby zwiększyć czytelność kodu oraz uprościć składnie oraz przede wszystkim *promises*.

Async używamy przed zdefiniowaniem funkcji lub jako ekspression która zwraca nam zawsze promise. Wtedy możemy uzywac tradycyjnej składni z .then oraz .catch. funkcja asynchroniczna **Async** działa na trzy sposoby:

- Jeśli nic nie zwroci nam funkcja, zwraca nam resolved value jako undefined;
- Jeśli wartość nie promisowa zostaje zwrocona z funkcji, wtedy zwraca nam promise wykonane dla tej wartości;
- Jeśli promise zostanie zwrócone z funkcji, po prostu zwraca ta promise

```
async function fivePromise() {
  return 5;
}

fivePromise()
.then(resolvedValue => {
    console.log(resolvedValue);
  })  // Prints 5
```

```
11  withConstructor(0)
12    .then((resolveValue) => {
13    console.log(` withConstructor(0) returned a promise which
    resolved to: ${resolveValue}.`);
14  })
15
16  // Write your code below:
17  async function withAsync(num){
18    if (num === 0){
19      return 'zero';
20    } else {
21      return 'not zero';
22    }
23  };
24
```

Await – można uzywac tylko z async razem. Umieszcza się go w body function(tylko i wyłącznie). Jest to operator który zwraca wartość obietnicy. **Async pauzuje wykonywanie funkcji póki obietnica nie zostanie spelniona.**

Async await pozwala nam praktycznie na synchroniczne wykonywanie kodu. Wymuszanie wykonania promise, a następnie nawet zatrzymanie wykonywania kodu aby spełnić obietnice.

```
async function asyncFuncExample(){
  let resolvedValue = await myPromise();
  console.log(resolvedValue);
}

asyncFuncExample(); // Prints: I am resolved now!
```

await zatrzymuje wykonywanie async function póki obietnica nie zostanie spelniona(może został spelniona lub odrzucona-to już inna sprawa).

```
// async/await version:
async function announceDinner() {
  // Write your code below:
  await brainstormDinner().then((meal) => {
    console.log(`I'm going to make ${meal} for dinner.`)
  })

}

announceDinner();
```

Lapanie bledów **try..catch** methods;

Funkcja Waiting() pauzuje wykonywanie funkcji póki pierwsza obietnica nie zostanie wykonana, wtedy wykonujemy drugą obietnice. Dopiero gdy obydwie zostaną wykonanie, razem wrzucane są do konsoli.

Funkcja concurrent() nie potrzebuje await do wykonania – obydwie obietnice mogą zostać wykonywane jednocześnie.

Jeśli mamy wiele obietnic, niezależnych od siebie, warto używać ,then() zamiast await.

Promise.all() – szczególnie przydatne kiedy potrzebujemy wszystkich pozytywnych promises ponieważ dzięki promise.all sprawdza nam wszystkie promises włożone w tablice jednocześnie i wyrzuca jeden wynik – jeśli chociaż jedna z nich jest negatywna, od razu kończy analizę i wyrzuca błąd. Przydatne żeby w razie błędów szybko kończyło analizę.

We can pass an array of promises as the argument to `Promise.all()`, and it will return a single promise. This promise will resolve when all of the promises in the argument array have resolved. This promise's resolve value will be an array containing the resolved values of each promise from the argument array.

```
async function asyncPromAll() {
  const resultArray = await
Promise.all([asyncTask1(), asyncTask2(),
asyncTask3(), asyncTask4()]);
  for (let i = 0; i<resultArray.length; i++){
    console.log(resultArray[i]);
  }
}
```

Podsumowanie:

Awesome work getting the hang of the **async...await syntax**! Let's review what you've learned:

- async...await is syntactic sugar built on native JavaScript promises and generators.
- We declare an async function with the keyword async.
- Inside an async function we use the await operator to pause execution of our function until an asynchronous action completes and the awaited promise is no longer pending .
- await returns the resolved value of the awaited promise.
- We can write multiple await statements to produce code that reads like synchronous code.
- We use try...catch statements within our async functions for error handling.
- We should still take advantage of concurrency by writing async functions that allow asynchronous actions to happen in concurrently whenever possible.

Requests (żądania)

https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods

http requests; 4 najbardziej popularne to:

- ***Get*** - dostajemy informacje(np. ze strony)
- ***Post*** – wysyłamy informacje do zeodla, która je przyjmuje i wysyla informacje zwrotną.
- put
- delete

Aby utworzyć takie metody potrzebujemy np. XHR object:

- dla get – datamuse API
- dla post – url shortener API;

JSON – java Script object notation;

Event loop: https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop

XML - https://developer.mozilla.org/en-US/docs/Web/XML/XML_introduction extensive markup lang do tworzenia różnych żądań

AJAX – wcześniej był używany tylko dla danych z rozszerzeniem XML, teraz może być tez używany w innych formatach;

Żądanie GET

w XHR (XMLHttprequest API).

W responseType podajemy json aby odpowiedz była sformatowana w języku java Script.

If w funkcji sprawdza czy żądanie zostało ukończone.

Pod if dodajemy return xhr.response aby otrzymać spowrotem żądanie(jego zwrotne dane).

Metoda **.open** tworzy nowe żądanie a jako argumenty ustawiamy typ żądania oraz URL tego żądania.

```
// XMLHttpRequest GET
                creates new object
const xhr = new XMLHttpRequest();
const url = 'http://api-to-call.com/endpoint';

xhr.responseType = 'json';
xhr.onreadystatechange = () => {
  if (xhr.readyState === XMLHttpRequest.DONE) {
      // Code to execute with response
    }
};                                          handles response

xhr.open('GET', url);
xhr.send();                    opens  request and sends object
```

Jest to tzw. Boilerplate, a więc ogólny blok kodu w tym przypadku tworzący ogólne żadanie get.
You've written the boilerplate code for an AJAX GET request using an XMLHttpRequest object.

Const url = **'rel_rhy='** is the start of a parameter for the query string. This parameter will narrow your search to words that rhyme.

Const wordQuery = inputField.value grabs what is in the inputField and assigns it to the variable wordQuery.

https://en.wikipedia.org/wiki/Query_string

dodawanie do url dodatkowej informacji przy wysyłaniu żądania. Przykład tworzenia na obrazku obok:

const queryParams = 'rel_jjb='; //szuka słowa opisującego inne(które podamy w wyszukiwarce np.);

const additionalParams = '&topics='; *//znak & oddziela nam parametry. Znak równości na końcu przypisze nam wlasnie w tym miejscu value do tego klucza jakim jest topics.*

A query string contains additional information to be sent with a request. The Datamuse API allows us to retrieve more specific data with query strings attached to the request URL.

- Wiki: query string

A query string is separated from the URL using a `?` character. After `?`, you can then create a parameter which is a key value pair joined by a `=`. Examine the example below:

```
'https://api.datamuse.com/words?key=value'
```

If you want to add an additional parameter you will have to use the `&` character to separate your parameters. Like so:

```
'https://api.datamuse.com/words?
key=value&anotherKey=anotherValue'
```

Tworzenie **żądania GET** przy użyciu zewnętrznego *API Datamuse*, dzięki któremu możemy w konsoli wpisać słowo, a następnie zażądać do niego słów go opisujących:

main.js

```
// Information to reach API
const url = 'https://api.datamuse.com/words?';
const queryParams = 'rel_jjb=';
const additionalParams = '&topics=';

// Selecting page elements
const inputField = document.querySelector('#input');
const topicField = document.querySelector('#topic');
const submit = document.querySelector('#submit');
const responseField =
document.querySelector('#responseField');

// AJAX function
const getSuggestions = () => {
  const wordQuery = inputField.value;
  const topicQuery = topicField.value;
  const endpoint =
`${url}${queryParams}${wordQuery}${additionalParams}${topi
cQuery}`;

  const xhr = new XMLHttpRequest();
  xhr.responseType = 'json';

  xhr.onreadystatechange = () => {
    if (xhr.readyState === XMLHttpRequest.DONE) {
      renderResponse(xhr.response);
    }
  }

  xhr.open('GET', endpoint);
  xhr.send();
}

// Clear previous results and display results to webpage
const displaySuggestions = (event) => {
  event.preventDefault();
  while(responseField.firstChild){
    responseField.removeChild(responseField.firstChild);
  }
  getSuggestions();
}

submit.addEventListener('click', displaySuggestions);
```

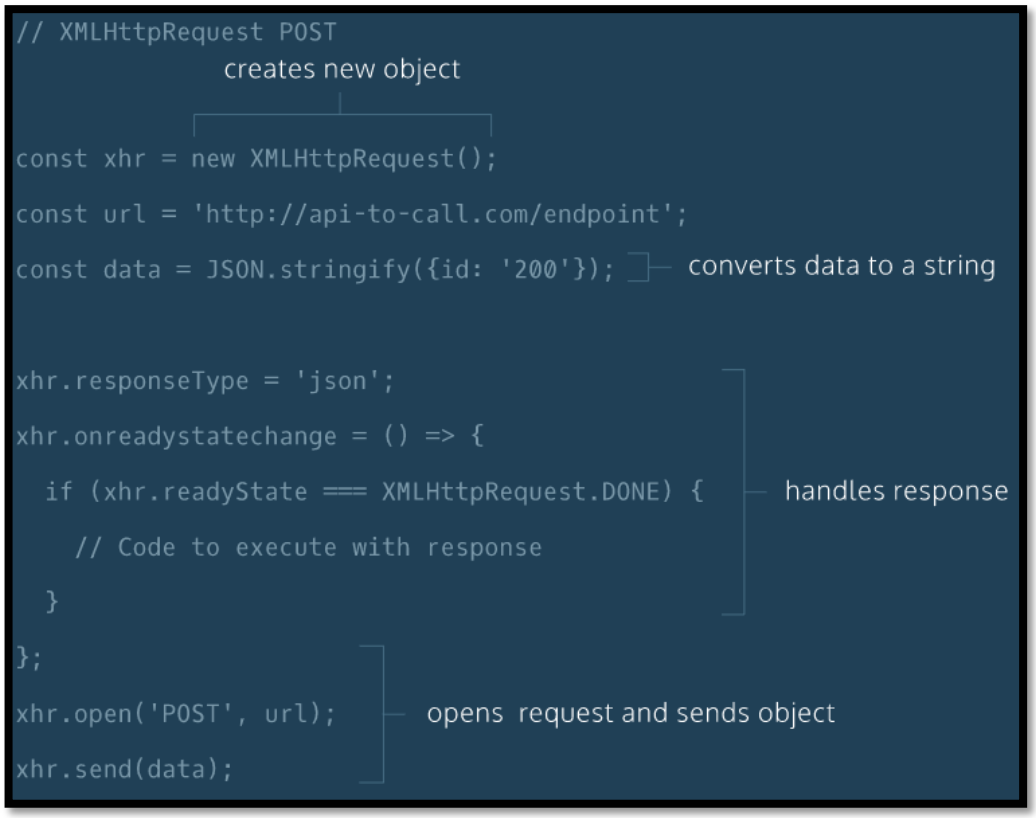
oraz funkcje pomocnicze dla tego żądania:

helperFunctions.js

```
// Formats response to look presentable on webpage
const renderResponse = (res) => {
  // handles if res is falsey
  if(!res){
    console.log(res.status)
  }
  // in case res comes back as a blank array
  if(!res.length){
    responseField.innerHTML = "<p>Try again!</p><p>There
were no suggestions found!</p>"
    return
  }

  // creating an array to contain the HTML strings
  let wordList = []
  // looping through the response and maxxing out at 10
  for(let i = 0; i < Math.min(res.length, 10); i++){
    // creating a list of words
    wordList.push(`<li>${res[i].word}</li>`)
  }
  // joins the array of HTML strings into one string
  wordList = wordList.join("")

  // manipulates responseField to render the modified
response
  responseField.innerHTML = `<p>You might be interested in:
</p><ol>${wordList}</ol>`
  return
}

// Renders response before it is modified
const renderRawResponse = (res) => {
  // taking the first 10 words from res
  let trimmedResponse = res.slice(0, 10)
  //manipulates responseField to render the unformatted
response
  responseField.innerHTML =
`<text>${JSON.stringify(trimmedResponse)}</text>`
}

// Renders the JSON that was returned when the Promise from
fetch resolves.
const renderJsonResponse = (res) => {
  // creating an empty object to store the JSON in key-
value pairs
  let rawJson = {}
  for(let key in response){
    rawJson[key] = response[key]
  }
  // converting JSON into a string and adding line breaks
to make it easier to read
  rawJson = JSON.stringify(rawJson).replace(/,/g, ", \n")
  // manipulates responseField to show the returned JSON.
  responseField.innerHTML = `<pre>${rawJson}</pre>`
}
```

Żądanie POST

Boilerplate żądania POST(AJAX):



Różnica miedzy post a *get* jest taka ze *post* wymaga dodatkowej informacji zawartej w żądaniu – ta dodatkowa informacja zawarta jest w *body* żądania *post*.

url podaje ścieżkę(serwer) do której zostanie wysłane żądanie.

Metoda .stringify konwertuje wartość JSON na string, dzięki temu możemy wysłać żądanie na serwer.

.onreadystatechange zawiera funkcje która zostanie wykonana przy zmianie statusu żądania.

W bloku *conditional statement*(pod if) wrzucamy ***return xhr.response*** //wartość response będzie równa danej którą otrzymamy z serwera na który wysłaliśmy żądanie.

.open tworzy nowe żądanie z **argumentami** post(**typ żądania**) oraz **adresem** na który ma zostać wysłane.

Tworzenie żądania POST w celu skrócenia URL.

const urlToShorten = inputField.value;

JSON.stringify({destination: urlToShorten}); *//podawanie tego kodu jest wymagane przez API, które oczekuje obiektu z kluczem destination który ma wartość URL.*

xhr.setRequestHeader('Content-type', 'application/json');

xhr.setRequestHeader('apikey', apiKey); //aby zdobyc dostepn do reblandly API potrzebne sa te dwie linijki kodu

```
// Information to reach API
const apiKey = '87c96cad4da341bd91112441cdf8cc86';
const url = 'https://api.rebrandly.com/v1/links';

// Some page elements
const inputField = document.querySelector('#input');
const shortenButton = document.querySelector('#shorten');
const responseField =
document.querySelector('#responseField');

// AJAX functions
const shortenUrl = () => {
  const urlToShorten = inputField.value;
  const data = JSON.stringify({destination:
urlToShorten});
  const xhr = new XMLHttpRequest(); //obiekt
  xhr.responseType = 'json';
  xhr.onreadystatechange = () => {
    if (xhr.readyState === XMLHttpRequest.DONE) {
  renderResponse(xhr.response);
}
  }
  xhr.open('POST', url);
  xhr.setRequestHeader('Content-type',
'application/json');
  xhr.setRequestHeader('apikey', apiKey);
  xhr.send(data);
}



// Clear page and call AJAX functions
const displayShortUrl = (event) => {
  event.preventDefault();
  while(responseField.firstChild){
    responseField.removeChild(responseField.firstChild);
  }
  shortenUrl();
}

shortenButton.addEventListener('click', displayShortUrl);
```

Podsumowanie Request 1

- JavaScript is the language of the web because of its asynchronous capabilities. AJAX, which stands for Asynchronous JavaScript and XML, is a set of tools that are used together to take advantage of JavaScript's asynchronous capabilities.
- There are many HTTP request methods, two of which are GET and POST.
- GET requests only request information from other sources.
- POST methods can introduce new information to other sources in addition to requesting it.
- GET requests can be written using an XMLHttpRequest object and vanilla JavaScript.
- POST requests can also be written using an XMLHttpRequest object and vanilla JavaScript.

- Writing GET and POST requests with XHR objects and vanilla JavaScript requires constructing the XHR object using new, setting the responseType, creating a function that will handle the response object, and opening and sending the request.
- To add a query string to a URL endpoint you can use ? and include a parameter.
- To provide additional parameters, use & and then include a key-value pair, joined by =.
- Determining how to correctly write the requests and how to properly implement them requires carefully reading the documentation of the API with which you're working.

Requests w ES6

Aby ulatwic ogarnianie asynchronicznosci w JS wprowadzono w ES6 pętle.

Aby ulatwic działanie na żądaniach wprowadzono funkcję ***.fetch()*** która używa obietnic. Dodatkowo używa się składni ***async await*** przy budowaniu żądań w ES6.

Fetch GET

Wydobywać/sprowadzać:

- Kreuje obiekt żądania, który zawiera podstawowe informacje, których potrzebuje API;
- Wysyła obiekt żądania do API;
- Zwraca obietnicę, która ostatecznie wygląda jak obiekt odpowiedzi(response object), który zawiera status obietnicy wraz z info jaką zwróciła API.
- Wygląda podobnie do *XMLHttpRequest*, lecz jest znacznie potężniejsze wraz z bazą dodatków(*feature set*).

Obok *boilerplate* **fetch'a GET**.

```
// fetch GET


fetch('http://api-to-call.com/endpoint').then(response => {
  if (response.ok) {
    return response.json();
  }
  throw new Error('Request failed!');
}, networkError => console.log(networkError.message)
).then(jsonResponse => {
    // Code to execute with jsonResponse
});
```

sends request

converts response object to JSON

handles errors

handles success

```
fetch('https://api-to-call.com/endpoint').then(response => {
  if(response.ok) { //ta wartosc bedzie boolean, jesli true to zwroci response.json
    return response.json();
  }
  throw new Error('Request failed!'); //kiedy response bedzie false, zwraca Error

}, networkError =>  console.log(networkError.message) //kiedy bedzie cos nie tak z serwerem
).then(jsonResponse => { return jsonResponse; //handler success - gdy sie wszystko uda
});
```

Utworzenie zapytania URL, nazywanego funkcją fetch(), w której zawarto zapytanie URL oraz ustawienia obiektu. Następnie

złączono metodą (.then), w której zawarto dwie funkcje jako argumenty – jedna gdy obietnica się spełni, druga do poradzenia sobie z odrzuceniem obietnicy.

Manipulowanie stroną po zwrocie informacji z żądania informacji(wraz z obietnicą):

```
  fetch(endpoint, {cache: 'no-cache'}).then(response => {
    if (response.ok) {
      return response.json();
    }
    throw new Error('Request failed!');
  }, networkError => {
    console.log(networkError.message)
  }).then(jsonResponse => { //dodajemy mozliwosc manipulowania stroną
    renderResponse(jsonResponse); //odpowiedz z api przerenderowana w języku JS(json);
  })
}
```

To żądanie wyrzuca nam sugerowane przez API słowa podobne do tego które wrzucimy w wyszukiwarkę na stronie.

Fetch POST

```
// fetch POST

fetch('http://api-to-call.com/endpoint', {
  method: 'POST',
  body: JSON.stringify({id: '200'})
}).then(response => {
    if (response.ok) {
      return response.json();
    }
    throw new Error('Request failed!');
}, networkError => console.log(networkError.message)
).then(jsonResponse => {
    // Code to execute with jsonResponse
});
```

sends request

converts response object to JSON

handles errors

handles success

Boilerplate fetch post;

Inicjowanie *post'a* **wymaga dwóch argumentów**: endpoint'u(**url**) oraz **obiektu**, który zawiera informacje potrzebne do wykonania żądania post. Reszta żądania jest identyczna co get.

Poniżej pełny boilerplate, ale opisany ;)

```
fetch('https://api-to-call.com/endpoint', {
  method: 'POST', //wybieramy metodę - tutaj post
  body: JSON.stringify({id: "200"}) //determinujemy jaką informację ma wysłać js do API;
}).then(response => { //tworzymy success callback function;
  if(response.ok){ //w tym bloku funkcji podajemy co sie stanie jesli spelni sie obietnica;
    return response.json();  //jesli zostanie zwrócona informacja, zostanie podana w następnym .then;
  }
  throw new Error('Request failed!'); //wyrzuca error jesli cos pojdzie nie tak w success callback function;
}, networkError => { //tworzymy funkcje failure, do rozpatrzenia w przypadku niespełnienia obietnicy;
  console.log(networkError.message); //konsola wyrzuca błąd
}).then(jsonResponse => { //tutaj wrzucana jest informacja z rozpatrzonego żądania
  console.log(jsonResponse); //nakazujemy aby wyświetliła się w konsoli;
})
```

Tworzenie post request dla skrocenia URL przy uzyciu rebrandly shortener API:

Tutaj utworzenie fetch'a post z url oraz obiektem.

```
const apiKey = '87c96cad4da341bd91112441cdf8cc86';
const url = 'https://api.rebrandly.com/v1/links'; //wymagany url dla fetch'a

// Some page elements
const inputField = document.querySelector('#input');
const shortenButton = document.querySelector('#shorten');
const responseField = document.querySelector('#responseField');

// AJAX functions
const shortenUrl = () => {
  const urlToShorten = inputField.value; //will keep the value of what is being typed into the input field
  const data = JSON.stringify({destination: urlToShorten}); //prepare the information needed to send in the body.
  fetch(url, {
    method: 'POST', //obiekt którego wymaga żądanie post;
    headers: {
      'Content-type': 'application/json',
  'apikey': apiKey
    },
    body: data
  })
  }).then( response => {
    if(response.ok) {
     return response.json()
    }
    throw new Error ('Request failed!')
  }, networkError => { //By adding this second callback, you're safeguarding yourself in the rare event that the network returns an error!
    console.log(networkError.message)
  }).then(jsonResponse => {
    renderResponse(jsonResponse)
  })
}
```

Ostatni *.then* umożliwia odtworzenie informacji na stronie.

Async await GET Requests

Wprowadzone w ES8, *Async* await upraszcza tworzenie żądań. Do składni wchodzą także *try* oraz *catch*.

```
// async await GET


async function getData() {
  try {
   const response = await fetch('https://api-to-call.com/endpoint');
    if (response.ok) {
      const jsonResponse = await response.json();
      // Code to execute with jsonResponse
    }
    throw new Error('Request Failed!');
  } catch (error) {
    console.log(error);
  }
}
```

sends request

handles response if successful

handles response if unsuccessful

Funkcja async zwraca obietnice. Await może być używane tylko z async. Await pozwala programowi działać kiedy czeka na rozwiązanie obietnicy.

Try…catch powoduje ze ten kod bloku zostanie odpalony i w przypadku error'u, kod z catch zostanie odpalony. Try catch służy do wyłapywania błędów (debugging!).

```
const getData = async () => { //arrow function z async keyword ktory sprawia ze funkcja zwraca obietnicę;
  try {
    const response = await fetch('https://api-to-call.com/endpoint'); //zmienna zapisze odpowiedz z url;
    if (response.ok) {
      const jsonResponse = await response.json();
      return jsonResponse //w normalnym kodzie wyglada to troche inaczej;
    } throw new Error('Request failed!');
  } catch(error) {
     console.log(error) //generalnie używa się przeniesienia na inną stronę w przypadku błędu;
  } //odpalony w przypadku erroru z bloku try;
};
```

Boilerplate z komentarzem.

Tworzenie żądania GET opisującego rzeczownikami wpisane słowo:

Oczywiście potrzebna jest do tego reszta kodu, plik index.html czy funkcje dodatkowe. Ale żądanie generalnie wygląda tak.

Pełny kod *JS* na stronie 35.

```
// AJAX function
// Code goes here
const getSuggestions = async () => {
  const wordQuery = inputField.value;
  const endpoint = (url + queryParams + wordQuery)
  try {
     const response = await fetch(endpoint, {cache: 'no-cache'}); //drugi argument cache po to aby api działo w kursowej przeglądarce;
    if(response.ok) {
      const jsonResponse = await response.json();
      renderResponse(jsonResponse);
    }
  } catch(error) {
    console.log(error)
  }
};
```

Async await POST request

```
// async await POST


async function getData() {
  try {
    const response = await fetch('https://api-to-call.com/endpoint', {
      method: 'POST',
      body: JSON.stringify({id: '200'})
    });
    if (response.ok) {
      const jsonResponse = await response.json();
      // Code to execute with jsonResponse
    }
    throw new Error('Request Failed!');
  } catch (error) {
    console.log(error);
  }
}
```

sends request

handles response if successful

handles response if unsuccessful

try…catch tak samo jak w *get*.

W fetch() jednak potrzebujemy **dodatkowego argumentu**, który zawiera ***metodę*** oraz ***body***.

```
const getData = async () => { //async keyword zapewnia że funkcja zwróci obietnice;
  try {
    const response = await fetch('https://api-to-call.com/endpoint', { //tutaj zapisuje się nasza zwrócona obietnica, jako zmienna response;
      method: 'POST',
      body: JSON.stringify({id: 200})
    })
    if(response.ok){
      const jsonResponse = await response.json(); //metoda json na zmiennej response; zwraca nam obiekt do zmiennej jsonResponse;
      return jsonResponse; //zwykle robie sie troche wiecej rzeczy zamiast tego powrotu;
    }
    throw new Error('Request failed!');
  } catch(error) { //jesli cos pojdzie nie tak, wczytuje ten blok;
    console.log(error); //...i loguje w konsoli error;
  }
}
```

Boilerplate z komentarzem.

Podsumowanie Request 2

- GET and POST requests can be created a variety of ways.
- Use AJAX to asynchronously request data from APIs. fetch() and async/await are new functionalities developed in ES6 (promises) and ES8 respectively.
- Promises are a new type of JavaScript object that represent data that will eventually be returned from a request.
- fetch() is a web API that can be used to create requests. fetch() will return promises.
- We can chain .then() methods to handle promises returned by fetch().
- The .json() method converts a returned promise to a JSON object.
- async is a keyword that is used to create functions that will return promises.
- await is a keyword that is used to tell a program to continue moving through the message queue while a promise resolves.
- await can only be used within functions declared with async.