EcmaScript 6

Wstęp

Początki JavaScriptu

Brendan Eich jest uznawany za głównego twórcę JavaScriptu. Język ten powstał w połowie lat 90. Początkowo nosił nazwę LiveScript, miał być przyjemniejszą alternatywą apletów Javy w przeglądarce Netscape Navigator.

Aby zachęcić samych programistów Javy, skorzystać z popularności tego języka, zmieniono wspomnianą wcześniej nazwę na JavaScript, która utrzymuje się do dziś.

Popularność JavaScriptu spowodowała, że Microsoft pokusił się o własną wersję tego języka. Nazwał go JScript, który był dostępny od wersji 2. Internet Explorera.

ECMA International

Brendan Eich postanowił chronić swoje dzieło i udał się do **ECMA International**.

ECMA International to stowarzyszenie, które powstało 1961 r., aby standaryzować systemy informatyczne w Europie.

Obecnie członkami tego stowarzyszenia są firmy IT, które wytwarzają, sprzedają lub rozwijają systemy informatyczne np. Mozilla, Google, Yahoo czy Microsoft.

ECMAScript

ECMAScript 1, 2

Standard wypracowany przez stowarzyszanie ECMA International określający obiektowy skryptowy język programowania w wersji pierwszej był dostępny już w 1997 r.

Wersja druga była opublikowana już w kolejnym roku, twórcy przeglądarek nie przejmowali się jednak standardami. Na rynku przeglądarek panowała dowolność w implementowaniu JS-a, co przysparzało wiele problemów webmasterom.

ECMAScript 3, 4

Dopiero opublikowana w 1999 roku wersja trzecia standardu ECMA-262 spowodowała ujednolicenie implementacji JS-a.

W latach 2005–2007 pracowano nad kolejną, czwartą wersją. Brakowało jednak kompatybilności wstecznej. Standard ten nigdy nie ujrzał światła dziennego i powrócono do rozszerzenia wersji trzeciej.

ECMAScript

ECMAScript 5

Standard ES5 został opublikowany w 2009 r. i był jedynie skromnym zestawem interfejsów API dodanych do wersji trzeciej.

Zarówno ten fakt, jak i brak publikacji standardu w wersji czwartej spowodował, że ES5 jest czasami określany mianem wersji 3.1.

ECMAScript

ECMAScript 6 / 2015

Dopiero wersję szóstą można uznać za prawdziwy krok w przód. Opublikowany w 2015 r. standard zawiera wiele nowych form składniowych i organizacji kodu oraz wspomagających interfejsów API.

Standard ten jest kompatybilny z poprzednią wersją, dlatego też należy uznać go za ewolucję a nie rewolucję. Kompatybilna wsteczność jest wymuszona przede wszystkim popularnością języka JavaScript.

Obecnie (2017 r.) wspominany standard nie jest wspierany w 100%, o czym możesz się przekonać dzięki tabeli kompatybilności dla ES6.

Operacje na tablicach

Programowanie funkcyjne

Funkcje wyższego rzędu

Metody, które będziemy opisywać tj. **forEach**, **map** i **reduce** to funkcje wyższego rzędu.

Oznacza to, że funkcje te przyjmują w parametrze inną funkcję lub ją zwracają.

```
var arr = [4, 1, 6, 2];

function fn(a, b) {
  return a - b;
}

arr.sort(fn);

console.log(arr);
```

Funkcje wyższego rzędu

Metody, które będziemy opisywać tj. **forEach**, **map** i **reduce** to funkcje wyższego rzędu.

Oznacza to, że funkcje te przyjmują w parametrze inną funkcję lub ją zwracają.

Jedną z takich funkcji, którą już omawialiśmy, jest metoda **sort** przyjmująca jako parametr inną funkcję.

```
var arr = [4, 1, 6, 2];

function fn(a, b) {
  return a - b;
}

arr.sort(fn);

console.log(arr);
```

Funkcje wyższego rzędu

Metody, które będziemy opisywać tj. **forEach**, **map** i **reduce** to funkcje wyższego rzędu.

Oznacza to, że funkcje te przyjmują w parametrze inną funkcję lub ją zwracają.

Jedną z takich funkcji, którą już omawialiśmy, jest metoda **sort** przyjmująca jako parametr inną funkcję.

```
var arr = [4, 1, 6, 2];

function fn(a, b) {
  return a - b;
}

arr.sort(fn);

console.log(arr);
```

```
[1, 2, 4, 6]
```

sort() – funkcje wyższego rzędu

W zależności od tego, jaką wartość zwróci funkcja **fn**, czyli większą od zera, równą zero lub mniejsza od zera, to zmieni się (lub nie) indeks elementów porównywanych.

Dzięki takiemu rozwiązaniu metoda **sort** ma odpowiedni poziom abstrakcji, dzięki czemu możemy sami definiować algorytm według którego tablica ma być sortowana.

```
var arr = ["aa", "bbbb", "c", "ddd"];

function fn(a, b) {
  return a.length - b.length;
}

arr.sort(fn);

console.log(arr);
```

```
["c", "aa", "ddd", "bbbb"]
```

Funkcja jako parametr – jak to działa?

Spójrz na przykład obok. Do funkcji **calc** przekazujemy anonimową funkcję jako trzeci parametr.

Po wykonaniu parsowania na liczby całkowite wywołujemy tę funkcję, przekazując jej jako parametry już liczby całkowite, a nie np. stringi.

```
function calc(arg1, arg2, callback) {
    var numberA = parseInt(arg1, 10);
    var numberB = parseInt(arg2, 10);
    callback(numberA, numberB);
}

calc("2", 3, function(a, b) {
     console.log(a + b);
});

calc("2", 3, function(a, b) {
     console.log(a * b);
});
```

Funkcja jako parametr – jak to działa?

Spójrz na przykład obok. Do funkcji **calc** przekazujemy anonimową funkcję jako trzeci parametr.

Po wykonaniu parsowania na liczby całkowite wywołujemy tę funkcję, przekazując jej jako parametry już liczby całkowite, a nie np. stringi.

Funkcja o nazwie **callback** to właśnie nasza funkcja anonimowa przekazana jako argument do funkcji **calc**.

```
function calc(arg1, arg2, callback) {
    var numberA = parseInt(arg1, 10);
    var numberB = parseInt(arg2, 10);
    callback(numberA, numberB);
}

calc("2", 3, function(a, b) {
     console.log(a + b);
});

calc("2", 3, function(a, b) {
     console.log(a * b);
});
```

Programowanie funkcyjne i imperatywne

Programowanie funkcyjne

Polega ono na opisywaniu problemu dzięki funkcjom i ich zagnieżdżaniu w sobie a nie jego ścisłym rozwiązaniu.

Programowanie imperatywne (proceduralne)

Podejście do programowania, w którym programista jawnie, krok po kroku, definiuje jak wykonać zadanie.

Programowanie funkcyjne i imperatywne

```
var arr = ["aa", "bbbb", "c", "ddd"];

function bubbleSort(arr) {
   var len = arr.length;
   for(var i=len-1; i>=0; i--) {
     for(var j=1; j<=i; j++) {
       if(arr[j-1].length > arr[j].length) {
           var temp = arr[j-1];
           arr[j-1] = arr[j];
           arr[j] = temp;
        }
     }
   }
   return arr;
}
console.log(bubbleSort(arr));
```

Programowanie funkcyjne i imperatywne

```
var arr = ["aa", "bbbb", "c", "ddd"];

function bubbleSort(arr) {
    var len = arr.length;
    for(var i=len-1; i>=0; i--) {
      for(var j=1; j<=i; j++) {
        if(arr[j-1].length > arr[j].length) {
            var temp = arr[j-1];
            arr[j-1] = arr[j];
            arr[j] = temp;
         }
      }
    }
    return arr;
}
console.log(bubbleSort(arr));
```

["c", "aa", "ddd", "bbbb"]

Programowanie funkcyjne - zalety i wady

Główną zaletą programowania funkcyjnego jest jego czytelność. Zajmujemy się jedynie najistotniejszym elementem (np. regułą sortowania) a nie tworzymy mało czytelnej struktury zmiennych i pętli.

Wadą tego rozwiązania jest jego wydajność.

```
var arr = [4, 1, 6, 2];
function fn(a, b) {
  return a - b;
}

arr.sort(fn);
console.log(arr); //[1, 2, 4, 6]
var arr2 = ["aa", "bbbb", "c", "ddd"];
function fn(a, b) {
  return a.length - b.length;
}

arr2.sort(fn);
//["c", "aa", "ddd", "bbbb"]
```

Operacje na tablicach

W kolejnym rozdziale omówimy szczegółowo trzy funkcje wprowadzające w świat programowania funkcyjnego w JavaScripcie.

forEach

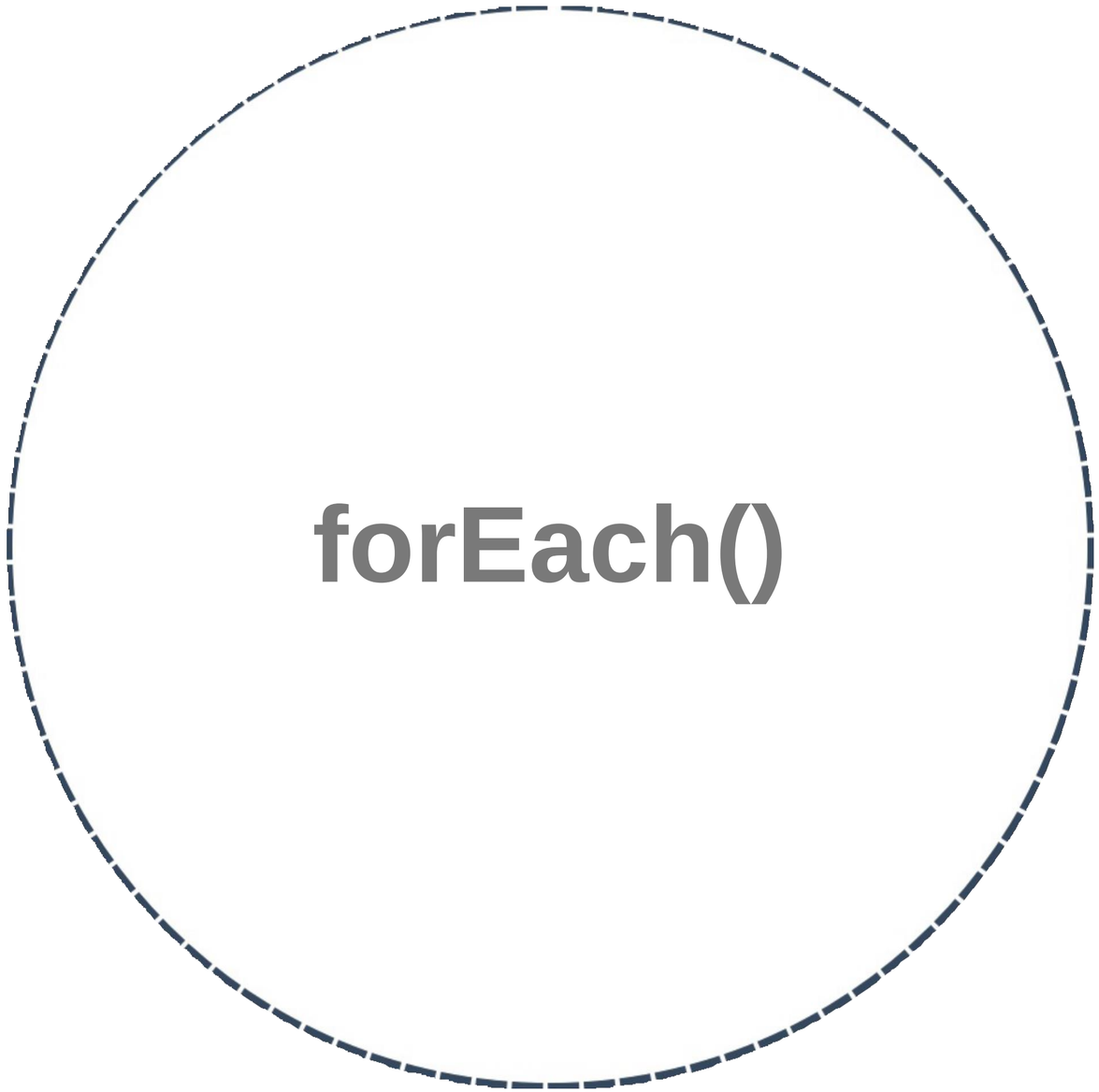
Pozwala wykonywać operacje z wykorzystaniem każdego elementu tablicy.

map

Pozwala modyfikować każdy element tablicy i zwraca wynik tych operacji jako nową tablicę.

reduce

Pozwala wykonywać operacje na każdym elemencie i wynik tej operacji wykorzystać przy następnej iteracji.


forEach()

Operacje na tablicach – forEach()

Metoda **forEach()** wykonuje przekazaną w parametrze funkcje dla każdego elementu tablicy.

Funkcja ta może mieć opcjonalnie trzy parametry, które kolejno odzwierciedlają:

- element tablicy (wartość) na którym obecnie jest wywoływana funkcja,
- indeks na którym obecnie jest wywoływana funkcja,
- tablicę na której jest wykonywana metoda **forEach()**.

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

function fn(element, index, array) {
  console.log("["+index+"]= "+element);
}
```

Operacje na tablicach – forEach()

Metoda **forEach()** wykonuje przekazaną w parametrze funkcje dla każdego elementu tablicy.

Funkcja ta może mieć opcjonalnie trzy parametry, które kolejno odzwierciedlają:

- element tablicy (wartość) na którym obecnie jest wywoływana funkcja,
- indeks na którym obecnie jest wywoływana funkcja,
- tablicę na której jest wykonywana metoda **forEach()**.

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

function fn(element, index, array) {
  console.log("["+index+"]= "+element);
}
```

```
arr.forEach(fn);
// [0] = 1
// [1] = 2
// [2] = 3
```

Wyświetlenie w konsoli nastąpiło trzy razy, ponieważ były trzy elementy w tablicy – dla każdego elementu funkcja **fn** została wywołana.

Implementacja forEach()

forEach() – implementacja

W jaki sposób metoda **forEach()** tak naprawdę działa? Spróbujmy napisać uproszczoną implementację tego rozwiązania.

```
Array.prototype.forEach = function(fn){

  if(typeof fn !== "function"){

    //wyrzucamy błąd

  }

  // dalsza cześć naszej implementacji

}
```

forEach() – implementacja

W jaki sposób metoda **forEach()** tak naprawdę działa? Spróbujmy napisać uproszczoną implementację tego rozwiązania.

```
Array.prototype.forEach = function(fn){

  if(typeof fn !== "function"){

    //wyrzucamy błąd

  }

  // dalsza cześć naszej implementacji

}
```

Rozszerzamy obiekt **Array** o nową metodę dzięki prototypowaniu, która przyjmuje jeden parametr.

forEach() – implementacja

W jaki sposób metoda **forEach()** tak naprawdę działa? Spróbujmy napisać uproszczoną implementację tego rozwiązania.

```
Array.prototype.forEach = function(fn){

  if(typeof fn !== "function"){

    //wyrzucamy błąd

  }

  // dalsza cześć naszej implementacji

}
```

Sprawdzamy dzięki **typeof**, czy ten parametr jest funkcją. Jeśli nie, to wyrzucamy błąd.

forEach() – implementacja

Następnie (o ile nie wyrzuciliśmy błędu) pobieramy liczbę elementów tablicy do zmiennej **len**. Pamiętajmy, że działamy w środku obiektu, aby zatem pobrać wartość właściwości (atrybutu) musimy użyć słowa kluczowego **this**.

Teraz zostaje już nam tylko wykorzystać pętlę **for**, aby iterować po wszystkich elementach tablicy i wywołać funkcję, która została przekazana jako parametr.

```
Array.prototype.forEach = function(fn){

  if(typeof fn !== "function"){
    //wyrzucamy błąd
  }

  var len = this.length;
  for(var i=0; i<len; i++){
    fn(this[i], i, this);
  }

}
```

forEach() – implementacja

Następnie (o ile nie wyrzuciliśmy błędu) pobieramy liczbę elementów tablicy do zmiennej **len**. Pamiętajmy, że działamy w środku obiektu, aby zatem pobrać wartość właściwości (atrybutu) musimy użyć słowa kluczowego **this**.

Teraz zostaje już nam tylko wykorzystać pętlę **for**, aby iterować po wszystkich elementach tablicy i wywołać funkcję, która została przekazana jako parametr.

```
Array.prototype.forEach = function(fn){

  if(typeof fn !== "function"){
    //wyrzucamy błąd
  }

  var len = this.length;
  for(var i=0; i<len; i++){
    fn(this[i], i, this);
  }

}
```

Pobranie liczby elementów.

forEach() – implementacja

Następnie (o ile nie wyrzuciliśmy błędu) pobieramy liczbę elementów tablicy do zmiennej **len**. Pamiętajmy, że działamy w środku obiektu, aby zatem pobrać wartość właściwości (atrybutu) musimy użyć słowa kluczowego **this**.

Teraz zostaje już nam tylko wykorzystać pętlę **for**, aby iterować po wszystkich elementach tablicy i wywołać funkcję, która została przekazana jako parametr.

```
Array.prototype.forEach = function(fn){

  if(typeof fn !== "function"){
    //wyrzucamy błąd
  }

  var len = this.length;
  for(var i=0; i<len; i++){
    fn(this[i], i, this);
  }

}
```

Iterowanie po wszystkich elementach tablicy i wywołanie funkcji.

forEach() – implementacja

Na koniec przyjrzyjmy się raz jeszcze całej pętli. W jej środku wywołujemy funkcję **fn** z trzema parametrami:

- element o indeksie **i** dla konkretnej tablicy,
- indeks od **0** do liczby elementów w tablicy,
- cały obiekt (tablica).

Teraz jest już jasne, skąd funkcja, którą przekazujemy do **forEach**, może mieć trzy parametry. Ich kolejność jest istotna.

```
Array.prototype.forEach = function(fn){

    if(typeof fn !== "function"){
        //wyrzucamy błąd
    }

    var len = this.length;
    for(var i=0; i<len; i++){
        fn(this[i], i, this);
    }
}
```

forEach() – implementacja

Na koniec przyjrzyjmy się raz jeszcze całej pętli. W jej środku wywołujemy funkcję **fn** z trzema parametrami:

- element o indeksie **i** dla konkretnej tablicy,
- indeks od **0** do liczby elementów w tablicy,
- cały obiekt (tablica).

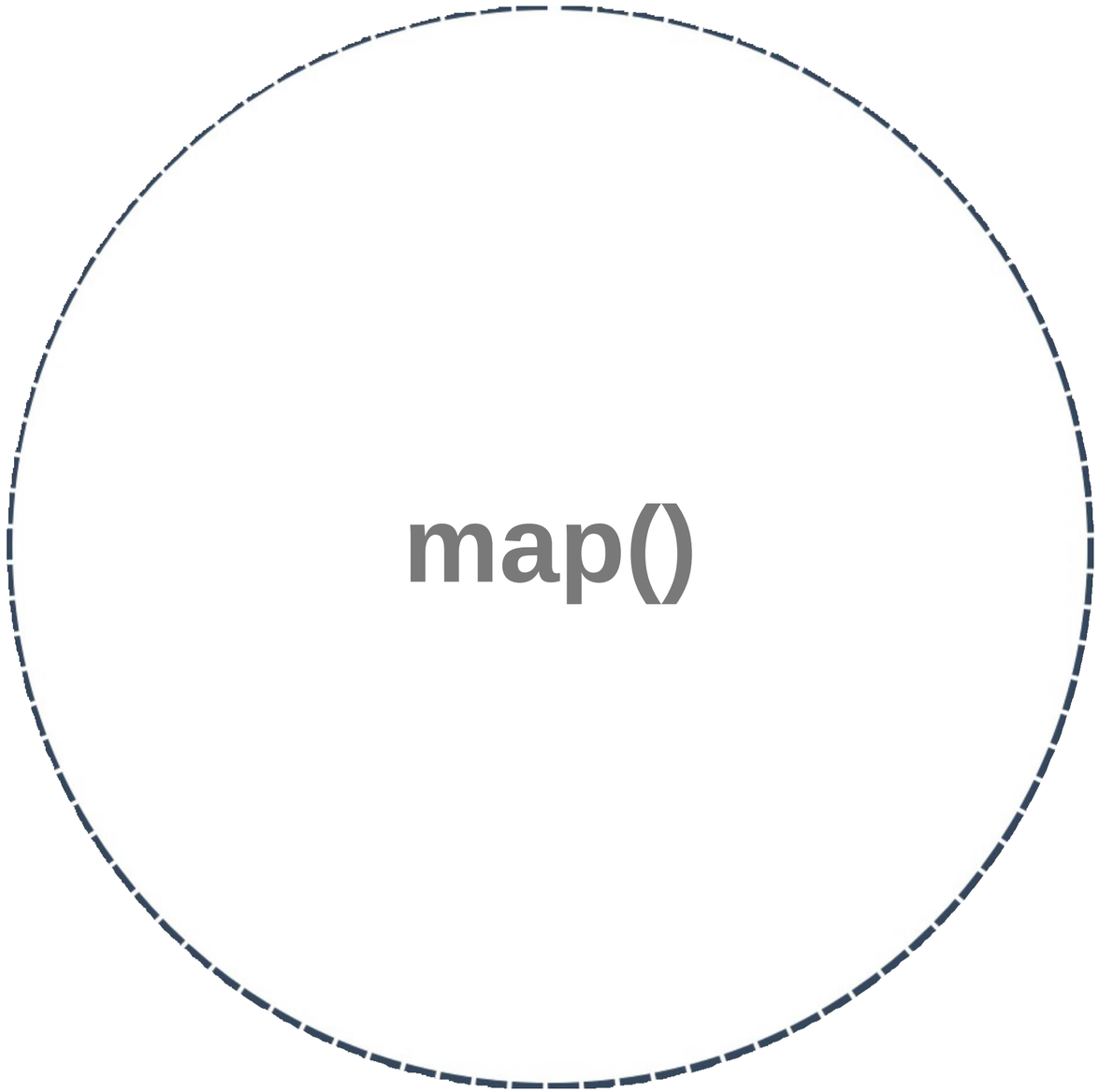
Teraz jest już jasne, skąd funkcja, którą przekazujemy do **forEach**, może mieć trzy parametry. Ich kolejność jest istotna.

```
Array.prototype.forEach = function(fn){

    if(typeof fn !== "function"){
        //wyrzucamy błąd
    }

    var len = this.length;
    for(var i=0; i<len; i++){
        fn(this[i], i, this);
    }
}
```

this[i] - element,
i - indeks,
this - obiekt.


map()

map() – implementacja

Metoda **map()** wykonuje przekazaną w parametrze funkcje dla każdego elementu tablicy a zwróconą przez nią wartość przypisuje do nowej tablicy o danym indeksie. Przekazywana funkcja może mieć trzy parametry, które kolejno odzwierciedlają:

- element tablicy (wartość) na którym obecnie jest wywoływana funkcja,
- indeks na którym obecnie jest wywoływana funkcja,
- tablicę na której jest wykonywana metoda **map()**.

```
var arr = [1, 2, 3];
function fn(element, index, array) {
  return element * index;
}
var result = arr.map(fn);
console.log(result); // [0, 2, 6]
```

map() – implementacja

Metoda **map()** wykonuje przekazaną w parametrze funkcje dla każdego elementu tablicy a zwróconą przez nią wartość przypisuje do nowej tablicy o danym indeksie. Przekazywana funkcja może mieć trzy parametry, które kolejno odzwierciedlają:

- element tablicy (wartość) na którym obecnie jest wywoływana funkcja,
- indeks na którym obecnie jest wywoływana funkcja,
- tablicę na której jest wykonywana metoda **map()**.

```
var arr = [1, 2, 3];
function fn(element, index, array) {
  return element * index;
}
var result = arr.map(fn);
console.log(result); // [0, 2, 6]
```

```
arr.map(fn);
// result[0] = 1 * 0
// result[1] = 2 * 1
// result[2] = 3 * 2
```

Implementacja map()

map() – implementacja

Tym razem i dla metody **map()** postaramy się napisać uproszczoną implementację.

```
Array.prototype.map = function(fn) {

  if(typeof fn !== "function") {
    //wyrzucamy błąd
  }

  // dalsza cześć naszej implementacji
}
```

map() – implementacja

Tym razem i dla metody **map()** postaramy się napisać uproszczoną implementację.

Ponownie rozszerzamy obiekt **Array** o nową metodę dzięki prototypowaniu.

```
Array.prototype.map = function(fn) {

  if(typeof fn !== "function") {
    //wyrzucamy błąd
  }

  // dalsza cześć naszej implementacji
}
```

map() – implementacja

Tym razem i dla metody **map()** postaramy się napisać uproszczoną implementację.

Ponownie rozszerzamy obiekt **Array** o nową metodę dzięki prototypowaniu.

Znów dzięki **typeof** sprawdzamy, czy ten parametr jest funkcją. Jeśli nie, to wyrzucamy błąd.

```
Array.prototype.map = function(fn) {

  if(typeof fn !== "function") {
    //wyrzucamy błąd
  }

  // dalsza cześć naszej implementacji
}
```

map() – implementacja

Teraz deklarujemy pustą tablicę, do której będziemy dodawać nowe elementy zwrócone przez funkcję **fn**.

Następnie pobieramy liczbę elementów tablicy do zmiennej **len** (podobnie jak poprzednio).

Teraz zostaje już nam tylko wykorzystać pętlę **for**, aby iterować po wszystkich elementach tablicy i wywołać funkcję, która została przekazana jako parametr.

```
Array.prototype.map = function(fn) {

  if(typeof fn !== "function") {
    //wyrzucamy błąd
  }

  var arr = [];
  var len = this.length;
  for(var i=0; i<len; i++) {
    arr[i] = fn(this[i], i, this);
  }

  return arr;
}
```

map() – implementacja

Teraz deklarujemy pustą tablicę, do której będziemy dodawać nowe elementy zwrócone przez funkcję **fn**.

Następnie pobieramy liczbę elementów tablicy do zmiennej **len** (podobnie jak poprzednio).

Teraz zostaje już nam tylko wykorzystać pętlę **for**, aby iterować po wszystkich elementach tablicy i wywołać funkcję, która została przekazana jako parametr.

```
Array.prototype.map = function(fn) {

  if(typeof fn !== "function") {
    //wyrzucamy błąd
  }

  var arr = [];
  var len = this.length;
  for(var i=0; i<len; i++) {
    arr[i] = fn(this[i], i, this);
  }

  return arr;
}
```

Deklaracja tablicy.

map() – implementacja

Teraz deklarujemy pustą tablicę, do której będziemy dodawać nowe elementy zwrócone przez funkcję **fn**.

Następnie pobieramy liczbę elementów tablicy do zmiennej **len** (podobnie jak poprzednio).

Teraz zostaje już nam tylko wykorzystać pętlę **for**, aby iterować po wszystkich elementach tablicy i wywołać funkcję, która została przekazana jako parametr.

```
Array.prototype.map = function(fn) {

  if(typeof fn !== "function") {
    //wyrzucamy błąd
  }

  var arr = [];
  var len = this.length;
  for(var i=0; i<len; i++) {
    arr[i] = fn(this[i], i, this);
  }

  return arr;
}
```

Pobranie liczby elementów.

map() – implementacja

Teraz deklarujemy pustą tablicę, do której będziemy dodawać nowe elementy zwrócone przez funkcję **fn**.

Następnie pobieramy liczbę elementów tablicy do zmiennej **len** (podobnie jak poprzednio).

Teraz zostaje już nam tylko wykorzystać pętlę **for**, aby iterować po wszystkich elementach tablicy i wywołać funkcję, która została przekazana jako parametr.

```
Array.prototype.map = function(fn) {

  if(typeof fn !== "function") {
    //wyrzucamy błąd
  }

  var arr = [];
  var len = this.length;
  for(var i=0; i<len; i++) {
    arr[i] = fn(this[i], i, this);
  }

  return arr;
}
```

Iteracja po wszystkich elementach tablicy.

map() – implementacja

Tym razem to, co zwróci nam funkcja, przypisujemy do tablicy o tym samym indeksie.

Po wykonaniu wszystkich iteracji zwracamy nowo utworzoną tablicę.

Jak widać **map()** nie modyfikuje tablicy, na której wykonuje operacje. Tworzy nową i zwraca ją po zakończeniu iteracji w pętli **for**.

```
Array.prototype.map = function(fn) {

  if(typeof fn !== "function") {
    //wyrzucamy błąd
  }

  var arr = [];
  var len = this.length;
  for(var i=0; i<len; i++) {
    arr[i] = fn(this[i], i, this);
  }

  return arr;
```

map() – implementacja

Tym razem to, co zwróci nam funkcja, przypisujemy do tablicy o tym samym indeksie.

Po wykonaniu wszystkich iteracji zwracamy nowo utworzoną tablicę.

Jak widać **map()** nie modyfikuje tablicy, na której wykonuje operacje. Tworzy nową i zwraca ją po zakończeniu iteracji w pętli **for**.

```
Array.prototype.map = function(fn) {

  if(typeof fn !== "function") {
    //wyrzucamy błąd
  }

  var arr = [];
  var len = this.length;
  for(var i=0; i<len; i++) {
    arr[i] = fn(this[i], i, this);
  }

  return arr;
```

Przypisanie do tablicy.

map() – implementacja

Tym razem to, co zwróci nam funkcja, przypisujemy do tablicy o tym samym indeksie.

Po wykonaniu wszystkich iteracji zwracamy nowo utworzoną tablicę.

Jak widać **map()** nie modyfikuje tablicy, na której wykonuje operacje. Tworzy nową i zwraca ją po zakończeniu iteracji w pętli **for**.

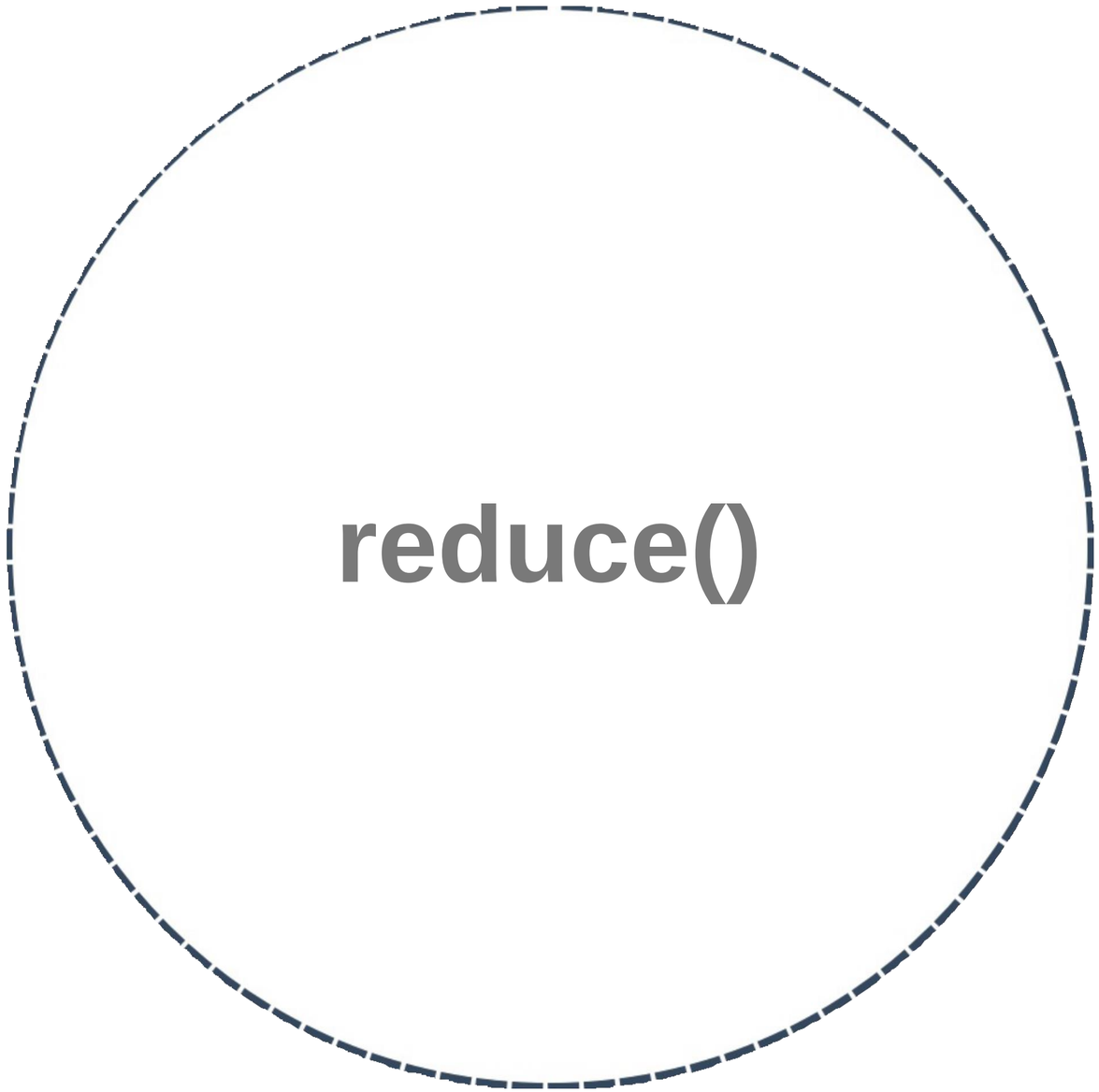
```
Array.prototype.map = function(fn) {

  if(typeof fn !== "function") {
    //wyrzucamy błąd
  }

  var arr = [];
  var len = this.length;
  for(var i=0; i<len; i++) {
    arr[i] = fn(this[i], i, this);
  }

  return arr;
```

Zwrócenie nowo utworzonej tablicy.


reduce()

Operacje na tablicach – reduce()

Metoda **reduce()** przyjmuje dwa parametry – pierwszy to funkcja, drugi (opcjonalny) to wartość początkowa, którą omówimy w następnych slajdach.

Metoda ta wykonuje operacje na poszczególnych elementach tablicy, zwracana wartość to wynik ostatniej operacji.

W przykładzie obok została obliczona suma wszystkich elementów w tablicy za pomocą opisywanej metody.

```
var arr = [1, 2, 3, 4];
function fn(prev, curr, index, array) {
  return prev + curr;
}
var result = arr.reduce(fn);
console.log(result);
```

Operacje na tablicach – reduce()

Metoda **reduce()** przyjmuje dwa parametry – pierwszy to funkcja, drugi (opcjonalny) to wartość początkowa, którą omówimy w następnych slajdach.

Metoda ta wykonuje operacje na poszczególnych elementach tablicy, zwracana wartość to wynik ostatniej operacji.

W przykładzie obok została obliczona suma wszystkich elementów w tablicy za pomocą opisywanej metody.

```
var arr = [1, 2, 3, 4];
function fn(prev, curr, index, array) {
  return prev + curr;
}
var result = arr.reduce(fn);
console.log(result);
```

```
10
```

Operacje na tablicach – reduce()

Zastanówmy się, jak obliczyć sumę wszystkich elementów w tablicy, nie używając **reduce()**?

Czy przypadkiem nie musimy dodać do wyniku poprzedniego działania kolejny element tablicy?

Zapiszmy to teraz za pomocą pętli **for** w taki sposób, aby liczba elementów w tablicy nie wpływała na nasz program.

```
var arr = [1, 2, 3, 4];
var sum = arr[0];
for(var i=1; i<arr.length; i++) {
  sum += arr[i];
}
console.log(sum); // 10;
```

Operacje na tablicach – reduce()

Zastanówmy się, jak obliczyć sumę wszystkich elementów w tablicy, nie używając **reduce()**?

Czy przypadkiem nie musimy dodać do wyniku poprzedniego działania kolejny element tablicy?

Zapiszmy to teraz za pomocą pętli **for** w taki sposób, aby liczba elementów w tablicy nie wpływała na nasz program.

```
var arr = [1, 2, 3, 4];
var sum = arr[0];
for(var i=1; i<arr.length; i++) {
  sum += arr[i];
}
console.log(sum); // 10;
```

```
var sum = arr[0]; // 1
```

Operacje na tablicach – reduce()

Zastanówmy się, jak obliczyć sumę wszystkich elementów w tablicy, nie używając **reduce()**?

Czy przypadkiem nie musimy dodać do wyniku poprzedniego działania kolejny element tablicy?

Zapiszmy to teraz za pomocą pętli **for** w taki sposób, aby liczba elementów w tablicy nie wpływała na nasz program.

```
var arr = [1, 2, 3, 4];
var sum = arr[0];
for(var i=1; i<arr.length; i++) {
  sum += arr[i];
}
console.log(sum); // 10;
```

```
sum = sum + arr[1]; // 3 <=1 + 2
sum = sum + arr[2]; // 6 <=3 + 3
sum = sum + arr[3]; // 10 <=6 + 4
```

Operacje na tablicach – reduce()

Podobnie jest w **reduce()**. Te dwie wartości tj. wartość poprzednią i obecną otrzymujemy w funkcji, którą przekazujemy jako parametr.

Funkcja ta przyjmuje cztery parametry wylistowane obok.

- wartość zwrócona w poprzednim wywołaniu funkcji (przy pierwszej iteracji wartość drugiego parametru metody **reduce()**, jeśli został przekazany lub pierwszy element tablicy),
- element tablicy (wartość) na którym obecnie jest wywoływana funkcja,
- indeks na którym obecnie jest wywoływana funkcja,
- tablicę na której jest wykonywana metoda **reduce()**.

Operacje na tablicach – reduce()

Tylko trzy iteracje, ponieważ pierwszy element tablicy jest przekazywany jako parametr **prev**.

```
var arr = [1, 2, 3, 4];
function fn(prev, curr, index, array) {
  return prev + curr;
}
var result = arr.reduce(fn);
console.log(result); // 10
```

Operacje na tablicach – reduce()

Tylko trzy iteracje, ponieważ pierwszy element tablicy jest przekazywany jako parametr **prev**.

```
var arr = [1, 2, 3, 4];
function fn(prev, curr, index, array) {
  return prev + curr;
}
var result = arr.reduce(fn);
console.log(result); // 10
```

1 + 2 = 3
3 + 3 = 6
6 + 4 = 10

Operacje na tablicach – reduce()

Przyjrzymy się raz jeszcze tej metodzie. Tym razem podamy drugi parametr, aby zobaczyć co się stanie.

Jak widać liczba iteracji zgadza się teraz z liczbą elementów w tablicy, ponieważ w pierwszej iteracji wartość parametru **prev** to 10, które przekazaliśmy jako parametr drugi do metody **reduce()**.

```
var arr = [1, 2, 3, 4];
function fn(prev, curr, index, array) {
  return prev + curr;
}
var result = arr.reduce(fn, 10);
console.log(result); // 20
```

Operacje na tablicach – reduce()

Przyjrzymy się raz jeszcze tej metodzie. Tym razem podamy drugi parametr, aby zobaczyć co się stanie.

Jak widać liczba iteracji zgadza się teraz z liczbą elementów w tablicy, ponieważ w pierwszej iteracji wartość parametru **prev** to 10, które przekazaliśmy jako parametr drugi do metody **reduce()**.

```
var arr = [1, 2, 3, 4];
function fn(prev, curr, index, array) {
  return prev + curr;
}
var result = arr.reduce(fn, 10);
console.log(result); // 20
```

```
10 + 1 = 11
11 + 2 = 13
13 + 3 = 16
16 + 4 = 20
```

Implementacja reduce()

reduce() – implementacja

Zobaczmy, jak mogłaby wyglądać implementacja takiej metody.

Tak jak w poprzednich przypadkach rozszerzamy obiekt **Array** i sprawdzamy, czy pierwszy parametr jest funkcją.

Następnie deklarujemy zmienne **prev** i **start**.

Prev będziemy przekazywać jako pierwszy parametr funkcji **fn**, dlatego musimy sprawdzić, czy zmienna **init** jest zdefiniowana.

```
Array.prototype.reduce =
function(fn, init){
  if(typeof fn !== "function") {
    //wyrzucamy informację o błędzie
  }
  var prev, start;
  if(typeof init === "undefined") {
    // drugi parametr nie został podany

  } else {
    // drugi parametr został podany
  }
  //dalsza część implementacji
}
```

reduce() – implementacja

Zobaczmy, jak mogłaby wyglądać implementacja takiej metody.

Tak jak w poprzednich przypadkach rozszerzamy obiekt **Array** i sprawdzamy, czy pierwszy parametr jest funkcją.

Następnie deklarujemy zmienne **prev** i **start**.

Prev będziemy przekazywać jako pierwszy parametr funkcji **fn**, dlatego musimy sprawdzić, czy zmienna **init** jest zdefiniowana.

```
Array.prototype.reduce =
function(fn, init){
  if(typeof fn !== "function") {
    //wyrzucamy informację o błędzie
  }
  var prev, start;
  if(typeof init === "undefined") {
    // drugi parametr nie został podany

  } else {
    // drugi parametr został podany
  }
  //dalsza część implementacji
}
```

Rozszerzamy prototyp.

reduce() – implementacja

Zobaczmy, jak mogłaby wyglądać implementacja takiej metody.

Tak jak w poprzednich przypadkach rozszerzamy obiekt **Array** i sprawdzamy, czy pierwszy parametr jest funkcją.

Następnie deklarujemy zmienne **prev** i **start**.

Prev będziemy przekazywać jako pierwszy parametr funkcji **fn**, dlatego musimy sprawdzić, czy zmienna **init** jest zdefiniowana.

```
Array.prototype.reduce =
function(fn, init){
  if(typeof fn !== "function") {
    //wyrzucamy informację o błędzie
  }
  var prev, start;
  if(typeof init === "undefined") {
    // drugi parametr nie został podany

  } else {
    // drugi parametr został podany
  }
  //dalsza część implementacji
}
```

Sprawdzamy, czy pierwszy parametr jest funkcją.

reduce() – implementacja

Zobaczmy, jak mogłaby wyglądać implementacja takiej metody.

Tak jak w poprzednich przypadkach rozszerzamy obiekt **Array** i sprawdzamy, czy pierwszy parametr jest funkcją.

Następnie deklarujemy zmienne **prev** i **start**.

Prev będziemy przekazywać jako pierwszy parametr funkcji **fn**, dlatego musimy sprawdzić, czy zmienna **init** jest zdefiniowana.

```
Array.prototype.reduce =
function(fn, init){
  if(typeof fn !== "function") {
    //wyrzucamy informację o błędzie
  }
  var prev, start;
  if(typeof init === "undefined") {
    // drugi parametr nie został podany

  } else {
    // drugi parametr został podany
  }
  //dalsza część implementacji
}
```

Deklarujemy zmienne **prev** i **start**.

reduce() – implementacja

Zobaczmy, jak mogłaby wyglądać implementacja takiej metody.

Tak jak w poprzednich przypadkach rozszerzamy obiekt **Array** i sprawdzamy, czy pierwszy parametr jest funkcją.

Następnie deklarujemy zmienne **prev** i **start**.

Prev będziemy przekazywać jako pierwszy parametr funkcji **fn**, dlatego musimy sprawdzić, czy zmienna **init** jest zdefiniowana.

```
Array.prototype.reduce =
function(fn, init){
  if(typeof fn !== "function") {
    //wyrzucamy informację o błędzie
  }
  var prev, start;
  if(typeof init === "undefined") {
    // drugi parametr nie został podany

  } else {
    // drugi parametr został podany
  }
  //dalsza część implementacji
}
```

Sprawdzamy, czy zmienna **init** jest zdefiniowana.

reduce() – implementacja

```
Array.prototype.reduce =
function(fn, init) {

  var prev, start;
  if(typeof init === "undefined") {
    // drugi parametr nie został podany
    prev = this[0];
    start = 1;
  } else {
    // drugi parametr został podany
    prev = init;
    start = 0;
  }
  //dalsza część implementacji
}
```

reduce() – implementacja

Jeśli zmienna **`init`** jest **`undefined`**, to **`prev`** będzie pierwszym elementem tablicy (**`this[0]`**);

```
Array.prototype.reduce =
function(fn, init) {

  var prev, start;
  if(typeof init === "undefined") {
    // drugi parametr nie został podany
    prev = this[0];
    start = 1;
  } else {
    // drugi parametr został podany
    prev = init;
    start = 0;
  }
  //dalsza część implementacji
}
```

reduce() – implementacja

Jeśli zmienna **`init`** jest **`undefined`**, to **`prev`** będzie pierwszym elementem tablicy (**`this[0]`**);

W przeciwnym razie **`prev`** będzie mieć wartość **`init`** dla pierwszej iteracji.

```
Array.prototype.reduce =
function(fn, init) {

  var prev, start;
  if(typeof init === "undefined") {
    // drugi parametr nie został podany
    prev = this[0];
    start = 1;
  } else {
    // drugi parametr został podany
    prev = init;
    start = 0;
  }
  //dalsza część implementacji
}
```

reduce() – implementacja

```
Array.prototype.reduce =
function(fn, init) {
  var prev, start;
  if(typeof init === "undefined") {
    prev = this[0];
    start = 1;
  } else {
    prev = init;
    start = 0;
  }
  var len =  this.length;
  for(var i=start; i<len; i++) {
    var result =
    fn(prev, this[i], i, this);
    prev = result;
  }
  return prev;
}
```

reduce() – implementacja

Zmienna **start** przechowuje informacje o indeksie, od którego mamy rozpocząć iteracje w pętli **for**.

```
Array.prototype.reduce =
function(fn, init) {
  var prev, start;
  if(typeof init === "undefined") {
    prev = this[0];
    start = 1;
  } else {
    prev = init;
    start = 0;
  }
  var len =  this.length;
  for(var i=start; i<len; i++) {
    var result =
    fn(prev, this[i], i, this);
    prev = result;
  }
  return prev;
}
```

reduce() – implementacja

Zmienna **start** przechowuje informacje o indeksie, od którego mamy rozpocząć iteracje w pętli **for**.

To co funkcja zwraca jest przypisywane do zmiennej **result** a następnie podmieniana jest wartość zmiennej **prev**.
Powyższe powoduje, że przy kolejnej iteracji funkcja **fn** ma inną wartość (tj. wynik z ostatniej operacji) dla pierwszego parametru.

```
Array.prototype.reduce =
function(fn, init) {
  var prev, start;
  if(typeof init === "undefined") {
    prev = this[0];
    start = 1;
  } else {
    prev = init;
    start = 0;
  }
  var len =  this.length;
  for(var i=start; i<len; i++) {
    var result =
    fn(prev, this[i], i, this);
    prev = result;
  }
  return prev;
}
```

reduce() – przykład z tabelą

```
var arr = [2, 3, 5];
function fn(prev, curr) {
  return prev + curr;
}
var result = arr.reduce(fn);
```

Tabela z wartościami zmiennych podczas każdej iteracji w pętli

Init	prev	start	I	len	i < len	this[i]	result
undefined	2	1	1	3	true	3	5
undefined	5	1	2	3	true	5	10
undefined	10	1	3	3	false		

Podsumowanie

Operacje na tablicach – podsumowanie

Funkcje wyższego rzędu, które omawialiśmy, pozwalają definiować nam dowolne rozwiązania dzięki możliwości przekazywania funkcji jako parametr.

Musimy jednak uważać, aby nie wywoływać tych funkcji w momencie ich przekazywania.

Zły przykład

```
var arr = [1, 2, 3];
function fn(value) {
  console.log(value);
}
arr.forEach(fn());  // BŁĄD!
```

Operacje na tablicach – podsumowanie

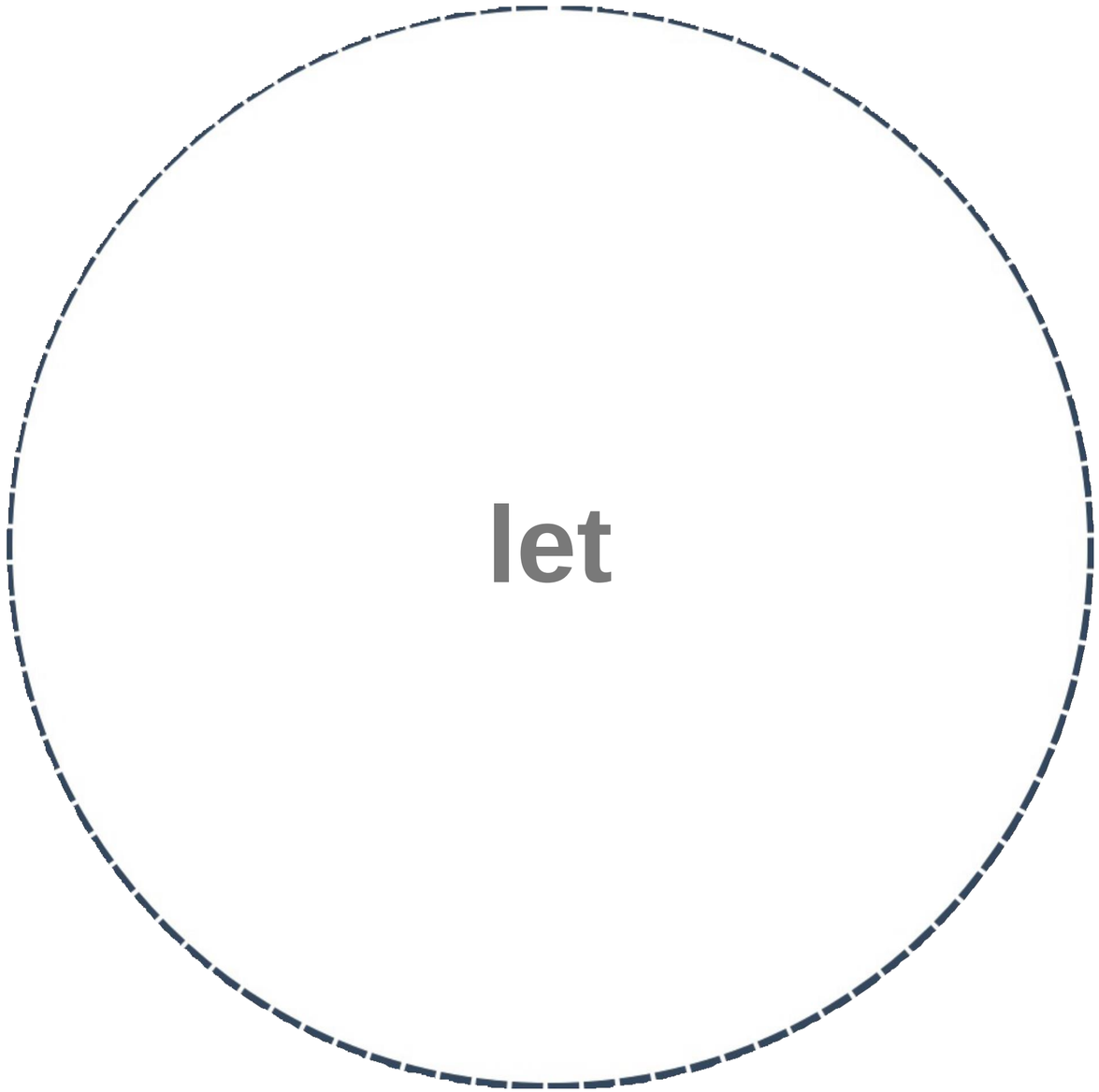
Pamiętajmy również, że możemy używać funkcji anonimowych, których przykłady znajdziemy obok.

Nazwy parametrów nie są istotne, a jedynie ich kolejność. Nie musimy także używać wszystkich dostępnych parametrów.

```
var arr = [1, 2, 3];
arr.forEach(function(value) {
  console.log(value);
});
var result = arr.map(function(item, i){
  return item * i;
});
```

Czas na zadania

Słowa kluczowe let i const


let

let – definicja

Słowo kluczowe **let**, które pojawiło się w standardzie ES6, ma to samo zastosowanie co **var**, czyli deklaruje zmienne.

Istnieje jednak między nimi różnica, chodzi o **let**, który pozwala deklarować zmienne o zakresie bloku.

Blok jest tworzony przez nawiasy klamrowe, wewnątrz których ta zmienna jest dostępna.

```
{
  let x = 5;

  console.log(x); // 5
}

// błąd: x jest niezdefiniowany
console.log(x);
```

let – zakres bez funkcji

W **let** wystarczy deklaracja zmiennej i wykorzystanie nawiasów klamrowych jako bloku, aby stworzyć zakres dla zmiennej.

W przykładzie obok zmienna lokalna **y** (wewnątrz nawiasów klamrowych) przesłania zmienną globalną **y**. Po wykonaniu operacji w bloku zmienna globalna **y** nie jest już przesłaniana, dlatego w konsoli widzimy wartość **4**.

Zakresem dla zmiennej lokalnej **y** jest blok **{...}**.

```
let x = 3;
let y = 4;
{
  let y = 5;

  console.log(x, y); // 3, 5
}
console.log(x, y); // 3 4
```

let – zakres bez funkcji

W **let** wystarczy deklaracja zmiennej i wykorzystanie nawiasów klamrowych jako bloku, aby stworzyć zakres dla zmiennej.

W przykładzie obok zmienna lokalna **y** (wewnątrz nawiasów klamrowych) przesłania zmienną globalną **y**. Po wykonaniu operacji w bloku zmienna globalna **y** nie jest już przesłaniana, dlatego w konsoli widzimy wartość **4**.

Zakresem dla zmiennej lokalnej **y** jest blok **{...}**.

```
let x = 3;
let y = 4;
{
  let y = 5;

  console.log(x, y); // 3, 5
}
console.log(x, y); // 3 4
```

Zmienna globalna.

let – zakres bez funkcji

W **let** wystarczy deklaracja zmiennej i wykorzystanie nawiasów klamrowych jako bloku, aby stworzyć zakres dla zmiennej.

W przykładzie obok zmienna lokalna **y** (wewnątrz nawiasów klamrowych) przesłania zmienną globalną **y**. Po wykonaniu operacji w bloku zmienna globalna **y** nie jest już przesłaniana, dlatego w konsoli widzimy wartość **4**.

Zakresem dla zmiennej lokalnej **y** jest blok **{...}**.

```
let x = 3;
let y = 4;
{
  let y = 5;

  console.log(x, y); // 3, 5
}
console.log(x, y); // 3 4
```

Zmienna lokalna.

let i zarządzanie pamięcią

Let pozwala nam także lepiej zarządzać pamięcią, ponieważ zmienna zadeklarowana za jego pomocą jest dostępna jedynie wewnątrz nawiasów klamrowych a potem jest niszczona.

```
{
  let x = 5;
  var y = 3;
}

//3
console.log(y);

//błąd: x jest niezdefiniowany
console.log(x);
```

Zmienne let w pętlach

Problem deklaracji za pomocą **var** obrazuje jeszcze lepiej pętla **for**, w której deklarujemy zmienną **i** (sterującą). Pętla wykona się trzy razy, ponieważ wartość **3** nie spełnia warunku **i <3**.

Następnie wyświetlamy wartość zmiennej **i**. Nie ma tutaj deklaracji zakresu, więc możemy wyświetlić wartość zmiennej **i**.

Tej zmiennej już nie będziemy potrzebowali, więc niepotrzebnie ona zalega w pamięci komputera. W przeciwieństwie do zmiennej **j**, która została usunięta z pamięci po wykonaniu pętli **for**.

```
for(var i=0; i<3; i++) {
   //do something
}

console.log(i); // 3

for(let j=0; j<3; j++) {
   //do something
}

//błąd: j jest niezdefiniowany
console.log(j);
```

Pętle i deklaracja funkcji

Przedstawiony problem się nawarstwia w momencie użycia wewnątrz pętli deklaracji funkcji.

Ponieważ zakres dla **i** jest globalny, to funkcja zapisana do tablicy odczyta jej wartość równą **3**.

Dzieje się tak, ponieważ funkcja odczytuje wartość zmiennej w momencie wywołania, a nie deklarowania.

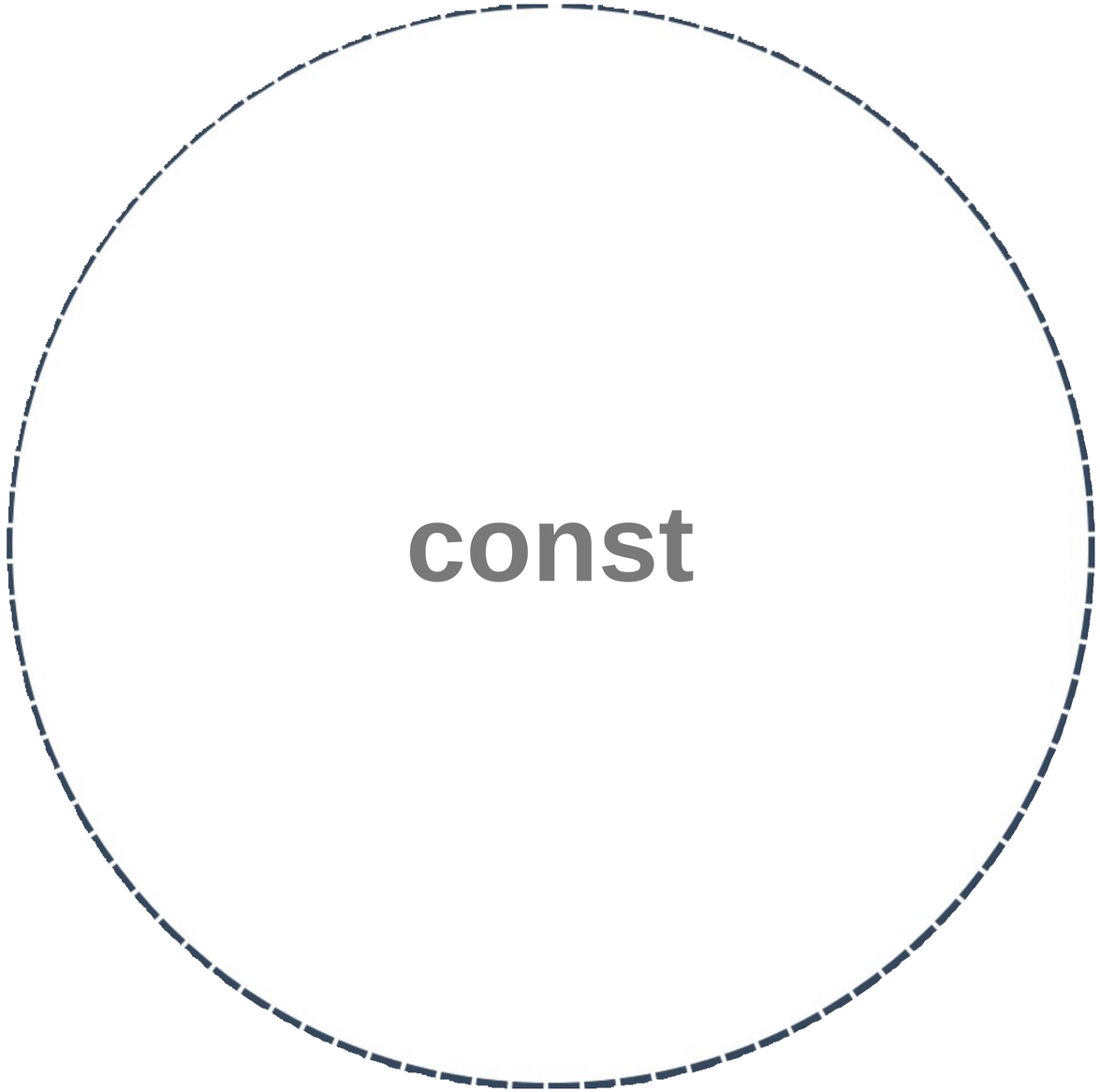
```
var arr = [];
for(var i=0; i<3; i++) {
  arr[i] = function() {
      console.log(i);
  }
}
arr[1]();        // 3!!
console.log(i); // 3
```

Pętle z let i deklaracja funkcji

W przypadku instrukcji **let**, zmienna jest deklarowana w nagłówku pętli **for**.

Dodatkowo będzie ponownie deklarowana podczas każdej iteracji pętli, co powoduje wyświetlenie wartości, jaka jest w momencie deklaracji funkcji.

```
var arr = [];
for(let i=0; i<3; ++i) {
  arr[i] = function() {
      console.log(i);
  }
}
arr[1]();        // 1!!
console.log(i); // błąd
```


const

const

Słowo kluczowe **const** to kolejna forma deklaracji tworzącej zmienne o zakresie bloku.

const w przeciwieństwie do **let** tworzy stałe. To oznacza, że raz przypisana wartość nie może się zmienić.

Jest to przydatne, aby zabezpieczyć się przed przypadkowym nadpisaniem zmiennej w dalszej części programu.

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

  //błąd: ponowne przypisanie wartości
  num = 2;
}
```

Modyfikacja zmiennej a zmiana wartości

Zmiana wartości zmiennej to zmiana adresu, na który ma wskazywać (w dużym uproszczeniu), co jest niedozwolone dla zmiennych deklarowanych przez słowo kluczowe **const**.

```
{
  const num = 1;
  const arr = [1, 2];
  num = 3; // błąd
  num = arr; // błąd
  arr = [3, 5]; // błąd
}
```

Natomiast modyfikacja obiektu (np. dodanie elementu do tablicy) nie powoduje zmiany adresu a jedynie zwiększa zajmowaną przestrzeń (w dużym uproszczeniu). Co przy zmiennych deklarowanych przez **const** jest dozwolone.

```
{
  const obj = {a: 1};
  obj.b = 2;
  console.log(obj); // Object {a:1, b:2}

  const arr = [1, 2, 3];
  arr.push(4);
  console.log(arr); // [1, 2, 3, 4]
}
```

Hoisting dla zmiennych - var

Odwołanie się do zmiennej, która jest deklarowana przez instrukcję **var** przed miejscem, w którym została zapisana, powoduje zwrócenie wartości **undefined**.

Dzieje się tak dzięki mechanizmowi hoistingu.

```
{
  console.log(x);  // undefined
  if(typeof x === "undefined") { //  ok!
    //do something
  }
  var x;
}
```

UWAGA! Hoisting – przypomnienie

Hoisting to mechanizm, który przenosi instrukcje tj. deklaracje funkcji oraz zmiennych na samą górę kodu jeszcze przed wystartowaniem programu.

```
function zrobHerbate() {
  // ciało funkcji
}
// tylko deklaracja bez inicjalizacji
var x;

console.log(x); //undefined
zrobHerbate(); // ok

function zrobHerbate() {
  console.log("Nalej wodę");
  console.log("Wsyp do kubka herbatę");
  console.log("Zagotuj wodę");
  console.log("Zalej herbatę");
}
var x = 10;
```

Hoisting dla zmiennych let i const

W przypadku instrukcji **let** oraz **const**, odwołanie się do zmiennej przed jej deklaracją spowoduje wystąpienie błędu.

Wynika on ze zbyt wczesnego odwołania się do zadeklarowanej zmiennej
(ang. **TDZ - Temporal Dead Zone** - tymczasowa martwa strefa).

Dobrą praktyką jest deklarowanie zmiennych **let** i **const** na samym początku bloku. Nie tylko spowoduje to zmniejszenie liczby błędów, ale i zwiększy czytelność kodu.

```
{
  console.log(x); // błąd
  if(typeof x === "undefined") { // błąd
    //do something
  }
  let x;
}
```

Deklaracja zakresu zmiennej bez let

Deklaracja zakresu zmiennej bez let

Aby móc osiągnąć ten sam efekt za pomocą **var**, co w przypadku słowa kluczowego **let** (czyli przed wejściem w życie standardu ES6), musieliśmy wykorzystać funkcję, która stworzy nam zakres zmiennej.

Minusem tego rozwiązania jest to, że sama deklaracja nie wystarczy. Musimy również wywołać funkcję **foo**.

```
function foo() {
  var x = 5;

  console.log(x);
}
foo(); // 5
console.log(x); // undefined
```

Po co tworzyć zakres zmiennej?

Tworzenie zakresu zmiennej pozwalała nam na swobodne używanie zmiennych wewnątrz zakresu, ponieważ powielenie nazwy zmiennej nie powoduje jej nadpisania a jedynie przesłonięcie.

```
var x = 1;
function foo() {
  var x = 5;

  console.log(x);
}
foo(); // 5
console.log(x); // 1
```

Deklaracja zakresu – IIFE

Istnieje taka konstrukcja, która może nam pomóc i nosi ona nazwę: natychmiast wywoływanego wyrażenia funkcji
(ang. **IIFE - Immediately Invoked Function Expression**).

Deklarację funkcji umieszczamy między nawiasami okrągłymi a następnie ją wywołujemy za pomocą kolejnych nawiasów okrągłych.

```
var i = 2;
( // zaczynamy od nawiasu
  function fn(index) {
    // zmienna index ma swój zakres

    // tmp dostępna tylko w fn
    var tmp = 0;
  }
)
(i); //wywołanie z parametrem
```

UWAGA! Deklaracja zakresu – IIFE

Najczęściej spotykaną konstrukcją jest ta obok. Czyli nawiasy okrągłe okalające definicje funkcji są w tym samym wierszu.

Użycie nazwy nie jest konieczne, ale pozwala rozpoznać funkcje w stosie wywołań, co czasem bywa bardzo pomocne przy debugowaniu.

Należy jeszcze zwrócić uwagę na parametr, który przekazujemy w funkcji. Jest to **x**, który wewnątrz funkcji ma swój zakres jako zmienna **param**, dodatkowo zmienna lokalna **x** przesłania nam zmienna globalną **x**.

```
var x =  3;

(function (param){
   var x = 5;

  console.log(param, x); // 3  5
})(x);

console.log(x); // 3
```

UWAGA! Deklaracja zakresu – IIFE

Najczęściej spotykaną konstrukcją jest ta obok. Czyli nawiasy okrągłe okalające definicje funkcji są w tym samym wierszu.

Użycie nazwy nie jest konieczne, ale pozwala rozpoznać funkcje w stosie wywołań, co czasem bywa bardzo pomocne przy debugowaniu.

Należy jeszcze zwrócić uwagę na parametr, który przekazujemy w funkcji. Jest to **x**, który wewnątrz funkcji ma swój zakres jako zmienna **param**, dodatkowo zmienna lokalna **x** przesłania nam zmienna globalną **x**.

```
var x =  3;

(function (param){
   var x = 5;

  console.log(param, x); // 3 5
})(x);

console.log(x); // 3
```

IIFE

UWAGA! Deklaracja zakresu – IIFE

Najczęściej spotykaną konstrukcją jest ta obok. Czyli nawiasy okrągłe okalające definicje funkcji są w tym samym wierszu.

Użycie nazwy nie jest konieczne, ale pozwala rozpoznać funkcje w stosie wywołań, co czasem bywa bardzo pomocne przy debugowaniu.

Należy jeszcze zwrócić uwagę na parametr, który przekazujemy w funkcji. Jest to **x**, który wewnątrz funkcji ma swój zakres jako zmienna **param**, dodatkowo zmienna lokalna **x** przesłania nam zmienna globalną **x**.

```
var x =  3;

(function (param){
   var x = 5;

  console.log(param, x); // 3 5
})(x);

console.log(x); // 3
```

(x) – wywołanie z parametrem.

UWAGA! Deklaracja zakresu – IIFE

Najczęściej spotykaną konstrukcją jest ta obok. Czyli nawiasy okrągłe okalające definicje funkcji są w tym samym wierszu.

Użycie nazwy nie jest konieczne, ale pozwala rozpoznać funkcje w stosie wywołań, co czasem bywa bardzo pomocne przy debugowaniu.

Należy jeszcze zwrócić uwagę na parametr, który przekazujemy w funkcji. Jest to **x**, który wewnątrz funkcji ma swój zakres jako zmienna **param**, dodatkowo zmienna lokalna **x** przesłania nam zmienna globalną **x**.

```
var x =  3;

(function (param){
   var x = 5;

  console.log(param, x); // 3 5
})(x);

console.log(x); // 3
```

(...) – deklaracja funkcji w nawiasach okrągłych.

Czas na zadania

Arrow functions

Arrow functions – co to jest?

Funkcje z użyciem strzałek (**Arrow functions** lub też czasem **Fat arrow**) to rozwiązanie, za pomocą którego możemy łatwo i szybko deklarować funkcje.

Ale to nie jedyna zaleta **Arrow functions**.

Dzięki temu likwidujemy również problem z wyrażeniem **this** przy tworzeniu funkcji w funkcjach.

parametry => ciało funkcji

```
let func = (x, y) => x + y;
```

Arrow functions – porównanie

Na początek przypomnijmy sobie, jak wygląda wyrażenie funkcyjne.

```
let foo = function(a, b) {
  if(a > b) {
    return a;
  }
  return b;
}
```

Teraz przyjrzyjmy się funkcji ze strzałkami.

```
let foo = (a, b) => {
  if(a > b) {
    return a;
  }
  return b;
}
```

Arrow functions
różny zapis

UWAGA! Arrow functions – różny zapis

Bez parametrów i jedna instrukcja

```
let foo = () => 1;
```

Jeden parametr i jedna instrukcja w ciele

```
let foo = a =>  a * 2;
```

```
let foo = a =>  { return a * 2; }
```

UWAGA! Arrow functions – różny zapis

Bez parametrów i jedna instrukcja

```
let foo = () => 1;
```

() – jeśli nie mamy parametrów nawiasy okrągłe muszą być.

Jeden parametr i jedna instrukcja w ciele

```
let foo = a =>  a * 2;
```

```
let foo = a =>  { return a * 2; }
```

UWAGA! Arrow functions – różny zapis

Bez parametrów i jedna instrukcja

```
let foo = () => 1;
```

=> – zamiast słowa **function** mamy strzałkę.

Jeden parametr i jedna instrukcja w ciele

```
let foo = a =>  a * 2;
```

```
let foo = a =>  { return a * 2; }
```

UWAGA! Arrow functions – różny zapis

Bez parametrów i jedna instrukcja

```
let foo = () => 1;
```

1 – jedna instrukcja nie wymaga ani nawiasów klamrowych, ani **return**.

Jeden parametr i jedna instrukcja w ciele

```
let foo = a =>  a * 2;
```

```
let foo = a =>  { return a * 2; }
```

UWAGA! Arrow functions – różny zapis

Bez parametrów i jedna instrukcja

```
let foo = () => 1;
```

Jeden parametr i jedna instrukcja w ciele

```
let foo = a =>  a * 2;
```

```
let foo = a =>  { return a * 2; }
```

a – jeden parametr nie wymaga nawiasów okrągłych.

UWAGA! Arrow functions – różny zapis

Bez parametrów i jedna instrukcja

```
let foo = () => 1;
```

Jeden parametr i jedna instrukcja w ciele

```
let foo = a =>  a * 2;
```

```
let foo = a =>  { return a * 2; }
```

a* 2 / { return a * 2; } – wciąż jedna instrukcja, nawiasy klamrowe są opcjonalne ale przy ich użyciu należy pamiętać o słowie kluczowym **return**.

UWAGA! Arrow functions – różny zapis

Wiele parametrów i wiele instrukcji w ciele

```
let foo = (a, b) => {
  if(a > b) {
    return a;
  }
  return b;
}
```

Immediately Invoked Function Expression

```
( (a, b) =>  a * b ) (1, 2)
```

UWAGA! Arrow functions – różny zapis

Wiele parametrów i wiele instrukcji w ciele

```
let foo = (a, b) => {
  if(a > b) {
    return a;
  }
  return b;
}
```

(a, b) – wiele parametrów, nawiasy okrągłe muszą być.

Immediately Invoked Function Expression

```
( (a, b) =>  a * b ) (1, 2)
```

UWAGA! Arrow functions – różny zapis

Wiele parametrów i wiele instrukcji w ciele

```
let foo = (a, b) => {
  if(a > b) {
    return a;
  }
  return b;
}
```

{...} – wiele instrukcji, nawias klamrowy musi być.

Immediately Invoked Function Expression

```
( (a, b) =>  a * b ) (1, 2)
```

Arrow functions – wcięcia

Wiele parametrów i wiele instrukcji w ciele

Syntax error

```
let foo = a
=>  a * 2;
```

Syntax error

```
let foo = a
=>  { a * 2; }
```

OK!

```
let foo = a =>  {
  return a * 2;
}
```

OK!

```
let foo = a =>
{
  return a * 2;
}
```

Arrow functions – przykład

Spróbujmy napisać funkcję z jednym parametrem oraz jedną instrukcją.

Wykorzystajmy to rozwiązanie w metodzie **`map()`**, którą omawialiśmy dokładnie na początku tego materiału.

Nie używamy nawiasów okrągłych, ponieważ jest tylko jeden parametr. Nie ma też wyrażenia **`return`**, ponieważ mamy tylko jedną instrukcję.

```
let arr = [1, 2, 3];
let newArr = arr.map(a => a * 2);
console.log(newArr); // [2, 4, 6]
```

Arrow functions – przykład

Tak samo możemy wykorzystać to rozwiązanie w metodzie **forEach()**, ponieważ mieliśmy jedną instrukcję.

```
let arr = [1, 2, 3];
let newArr = [];
arr.forEach(
  (item, idx) => newArr.push(item * idx)
);
console.log(newArr); // [0, 2, 6];
```

Arrow functions & this

Problem z this – przykład 1/4

Spójrzmy na przygotowany przykład. Mamy tam obiekt **Basket** stworzony za pomocą konstruktora mającego dwie właściwości:

- **`prices`** – tablica przechowująca ceny,
- **sum** – zmienna liczbowa przechowująca sumę wszystkich cen.

Następnie rozszerzyliśmy ten obiekt o metodę **`calculatePrices`**. Docelowo będzie ona obliczać sumę wszystkich cen i przypisywać do właściwości **sum**, ale na razie zobaczmy, na co wskazuje **`this`**.

Dlaczego funkcja Basket jest zwykłą funkcją? Ponieważ funkcje strzałkowe nie mogą być konstruktorami!

Problem z this – przykład 1/4

```
var Basket = function(prices) {
  this.prices = prices; // tablica przechowująca ceny
  //zmienna liczbowa przechowująca sumę wszystkich cen
  this.sum = 0;
};
//dodajemy do prototypu metodę calculatePrices
Basket.prototype.calculatePrices = function() {
  console.log(this);
}

var basket = new Basket([9.99, 7.77]);

basket.calculatePrices();
```

Problem z this – przykład 1/4

```
var Basket = function(prices) {
  this.prices = prices; // tablica przechowująca ceny
  //zmienna liczbowa przechowująca sumę wszystkich cen
  this.sum = 0;
};
//dodajemy do prototypu metodę calculatePrices
Basket.prototype.calculatePrices = function() {
  console.log(this);
}

var basket = new Basket([9.99, 7.77]);

basket.calculatePrices();
```

this wskazuje na obiekt, który utworzyliśmy:

Basket {prices: [9.99, 7.77], sum: 0}

Problem z this – przykład 2/4

```
Basket.prototype.calculatePrices = function() {

    this.prices.forEach(function(element) {
      this.sum += element;
    });

  console.log(this.sum);
}
```

Zmodyfikujmy teraz metodę **calculatePrices** w taki sposób, aby zliczała sumę wszystkich cen.

Użyliśmy do tego dobrze nam znaną metodę **forEach()**. Po ponownym uruchomieniu skryptu w konsoli pojawi się napis **0**? Dlaczego?

Problem z this – przykład 2/4

```
Basket.prototype.calculatePrices = function() {

  this.prices.forEach(function(element) {
    console.log(this);
  });

console.log(this.sum);
}
```

W momencie uruchomienia metody **forEach()**, której przekazaliśmy przez parametr funkcje, zmienił się nam kontekst.

Zobaczmy, co pojawi się w konsoli po drobnej modyfikacji w naszym kodzie.

Problem z this – przykład 2/4

```
Basket.prototype.calculatePrices = function() {

  this.prices.forEach(function(element) {
    console.log(this);
  });

console.log(this.sum);
}
```

W momencie uruchomienia metody **forEach()**, której przekazaliśmy przez parametr funkcje, zmienił się nam kontekst.

Zobaczmy, co pojawi się w konsoli po drobnej modyfikacji w naszym kodzie.

Window { ... }

UWAGA! Problem z this – przykład 3/4

Wewnątrz funkcji – **this** wskazuje na całkiem inny obiekt – jest to obiekt globalny **Window**.

```
Basket.prototype.calculatePrices = function() {
  this.prices.forEach(function(element) {
    console.log(this);
  }, this);
  console.log(this.sum);
}
```

UWAGA! Problem z this – przykład 3/4

Wewnątrz funkcji – **this** wskazuje na całkiem inny obiekt – jest to obiekt globalny **Window**.

```
Basket.prototype.calculatePrices = function() {
  this.prices.forEach(function(element) {
    console.log(this);
  }, this);
  console.log(this.sum);
}
```

Aby temu zapobiec, musielibyśmy przekazać w parametrze obiekt, na który ma wskazywać **this** wewnątrz funkcji.

Problem z this – przykład 4/4

Teraz wszystko działa jak należy, ale czy nie moglibyśmy zrobić tego w prostszy sposób?

```
var Basket = function(prices) {
  this.prices = prices;
  this.sum = 0;
};
Basket.prototype.calculatePrices = function() {
  let self = this;
  this.prices.forEach(function(element) {
     self.sum += element;
  });
  console.log(this.sum.toFixed(2));
}
var basket = new Basket([9.99, 7.77]);
basket.calculatePrices(); // 17.76
```

Z pomocą przychodzą nam **Arrow Functions** , które nie zmieniają kontekstu dla funkcji.

UWAGA! Problem z this => Arrow functions

Poniżej zmieniony kod dotyczący metody **forEach()**, do której przekazujemy funkcję strzałkową - **arrow function**.

```
var Basket = function(prices) {
  this.prices = prices;
  this.sum = 0;
};
Basket.prototype.calculatePrices = function() {
  this.prices.forEach(
    element => this.sum += element
  );
  console.log(this.sum.toFixed(2));
}
var basket = new Basket([9.99, 7.77]);
basket.calculatePrices(); // 17.76
```

UWAGA! Problem z this => Arrow functions

Poniżej zmieniony kod dotyczący metody **forEach()**, do której przekazujemy funkcję strzałkową - **arrow function**.

```
var Basket = function(prices) {
  this.prices = prices;
  this.sum = 0;
};
Basket.prototype.calculatePrices = function() {
  this.prices.forEach(
    element => this.sum += element
  );
  console.log(this.sum.toFixed(2));
}
var basket = new Basket([9.99, 7.77]);
basket.calculatePrices(); // 17.76
```

Arrow function nie zmienia wartości this.

This wewnątrz Arrow functions

This wewnątrz **Arrow Functions** wskazuje na ten sam obiekt **`this`**, który istnieje poza ciałem tej funkcji.

To znaczy, że **Arrow Functions** są zawsze „związane” z najbliższym kontekstem, w którym zostały wywołane a to znaczy, że obiekt **`this`** nie jest tworzony bezpośrednio w **Arrow functions**.

Kontekst, w którym istnieje this

```
[2,3,4].forEach(
  element => this.sum += element
);
```

This wewnątrz Arrow functions

This wewnątrz **Arrow Functions** wskazuje na ten sam obiekt **`this`**, który istnieje poza ciałem tej funkcji.

To znaczy, że **Arrow Functions** są zawsze „związane” z najbliższym kontekstem, w którym zostały wywołane a to znaczy, że obiekt **`this`** nie jest tworzony bezpośrednio w **Arrow functions**.

Kontekst, w którym istnieje this

```
[2,3,4].forEach(
  element => this.sum += element
);
```

Wewnątrz **Arrow Functions** **`this`** nie zmienił kontekstu.

Arrow functions - kiedy nie używać?

This w Node.js

Jeszcze raz o this

Jak już wiesz **this** wskazuje na kontekst, na którym została wywołana funkcja lub obiekt **Window**. Ale czy zawsze?

Jeśli uruchamiasz swój skrypt bezpośrednio w przeglądarce, to jej silnik zajmuje się interpretacją każdej linii twojego kodu.

Jeśli natomiast kompilujesz twój kod za pomocą Webpacka to Babel zajmuje się kompilacją. Wszystko dzieje się poza przeglądarką.

Tak naprawdę możemy nasz kod uruchomić nawet w konsoli dzięki środowisku Node.js. W związku z tym w konsoli nie będziemy mieli dostępnego obiektu **Window**.

Jeszcze raz o this

Jeśli skompilujesz poniższy kod:

```
console.log("this: " + this);
```

... to po spojrzeniu na kod wynikowy zobaczysz, że Webpack stworzył coś takiego:

```
console.log("this: " + undefinded);
```

Dzieje się tak, ponieważ Webpack nie wie nic o globalnym kontekście.

Definiowanie metod w obiektach

Pamiętaj, że **Arrow functions** to nie zawsze dobre rozwiązanie. Spójrz na przykład obok. Próbujemy wypisać w metodzie **getName** imię psa. Niestety w konsoli dostajemy błąd z informacją, że własność **name** jest niezdefiniowana.

```
var dog = {
  name : "Puszek",
  getName: () => {
    return this.name;
  }
};
console.log(dog.getName());
```

```
Uncaught TypeError: Cannot read
property 'name' of undefined
```

Definiowanie metod w obiektach

Spróbujmy wypisać, czym jest **this**.

```
var dog = {
  name : "Puszek",
  getName: () => {
    console.log(this)
  }
};
// undefined / Window
console.log(dog.getName());
```

Jak pamiętasz **Arrow Functions** korzystają z obiektu **this** z zewnętrznego **scope**. Przykład ten uruchomiony w przeglądarce zwróci **Window**, ale skompilowany przez Babel wypisze **undefined**.

W takim przypadku lepiej użyć normalnej funkcji.

Arrow functions Podsumowanie

Arrow functions – podsumowanie

Arrow functions zachowują się zupełnie inaczej niż tradycyjne funkcje:

- są zawsze „związane” z najbliższym kontekstem, w którym zostały wywołane, to znaczy, że nie jest dla nich tworzone wyrażenie **this**,
- **Arrow functions** są zawsze anonimowe,
- Nie mogą być konstruktorami tzw. nie mogą być wywoływane z użyciem operatora **new**,
- Nie mają własności **prototype**.

Domyślne wartości

Domyślne wartości parametrów

Standard ES6 przyniósł ze sobą możliwość definiowania domyślnych wartości parametrów funkcji. Wystarczy, że do parametru przypiszemy wartość domyślną za pomocą operatora równa się.

W momencie gdy podczas wywoływania funkcji nie podamy parametru (czyli parametr będzie miał wartość **undefined**), to przyjmie przypisaną wartość domyślną.

Domyślnymi wartościami mogą być również tablice, obiekty a nawet funkcje.

```
let fn = (a = 10, b = 5) =>  {
  console.log(a, b);
}
fn(12) // 12 5
fn() // 10 5
fn(undefined, 10) // 10 10
```

Oczywiście standardowa funkcja również może przyjąć wartości domyślne.

```
function fn(a = 10, b = 5) {
  console.log( a, b );
}
```

Domyślne wartości w ES5

Domyślne wartości w ES5

Zobaczmy jeszcze, jak do tej pory (w ES5) developerzy radzili sobie z tym problemem.

Pierwsze rozwiązanie mamy zaprezentowane w funkcji **fn1** z operatorem alternatywy (**||**).

Działa ono w taki sposób, że jeśli wyrażenie stojące po jego lewej stronie (np. **a**) nie jest fałszywe, to ono jest przypisywane do zmiennej.

W przeciwnym wypadku wartość po prawej (np. **10**) jest przypisywana.

```
function fn1(a, b) {
  a = a || 10;
  b = b || 5;
  console.log(a, b);
}
fn1(4, 12); // 4 12
fn1(); // 10 5
```

Domyślne wartości w ES5

Pamiętamy, że wartościami fałszywymi są:

- 0
- null
- false
- undefined
- ""
- NaN

```
fn1(); // 10 5
fn1(undefined, undefined); // 10  5
fn1(0, 4);         // 10 4
fn1(null, 7);      // 10 7
fn1(false, 17); // 10 17
fn1(2, false);   // 2 5
fn1(false);       // 10 5
fn1(4, "");       // 4 5
```

Domyślne wartości w ES5

Zaprezentowane wcześniej rozwiązanie jest o tyle niebezpieczne, że nie zawsze wartość fałszywa (np. `0` lub `, ""`) ma być interpretowana jako wartość niepożądana, którą chcemy nadpisać wartością domyślną.

Dlatego społeczność wypracowała inne, bardziej restrykcyjne rozwiązanie.

Sprawdzamy, czy parametr ma wartość **`undefined`**, jeśli tak, to tylko wtedy korzystamy z wartości domyślnej.

```
function fn2(a, b) {
  a = typeof a !== "undefined" ? a : 10;
  b = typeof b !== "undefined" ? b : 5;
  console.log(a + b);
}
fn2(4, 12); // 4 12
fn2();      // 10 5
```

Domyślne wartości w ES5

zastosowanie operatora alternatywy ||

```
fn1(4, 12); // 4 12
fn1(); // 10 5
fn1(undefined, undefined); // 10 5
fn1(0, 4);       // 10 4
fn1(null, 7);    // 10 7
fn1(false, 17); // 10 17
fn1(2, false);  // 2 5
fn1(false);      // 10 5
fn1(4, "");      // 4 5
```

zastosowanie typeof a !== "undefined"

```
fn2(4, 12); // 4 12
fn2(); // 10 5
fn2(undefined, undefined); // 10 5
fn2(0, 4);       // 0 4
fn2(null, 7);    // null 7
fn2(false, 17); // false 17
fn2(2, false);  // 2 false
fn2(false);      // false 5
fn2(4, "");      // 4 ""
```

Czas na zadania