Funkcje

Przemysław Maćkowiak

Agenda

Motywacja

- ”function” declaration
  - Return
  - Parametry
  - Konwencja nazewnicza
  - Ćwiczenia
- Function scope
- Functions, hoisting
- Function expressions
  - Funkcja anonimowe
  - Callbacks
- Funkcje strzałkowe

Agenda

Motywacja

- DRY – Do not Repeat Yourself

```
// calculate the sum of positive numbers 1, 2, 5
const a = 1, b = 2, c = 5;
sum = a + b + c;
console.log(`Sum is ${sum}`);

// now let's calculate another sum of negative numbers
a = -10, b = -20, c = -5;
sum = a + b + c;
console.log(`Sum is ${sum}`);

// this time we consider fractional numbers 12.45, 10.01, -90.34
a = 12.45, b = 10.01, c = -90.34;
sum = a + b + c;
console.log(`Sum is ${sum}`);
```

funkcja jest Twoim przyjacielem !

Tworzenie funkcji

- Funkcja może być tworzona na trzy sposoby:
  - **function declaration**
  - function expression
  - Function contructor

Motywacja

- function declaration (function definition)

```
function sumThree (a, b, c) {
    const sum = a + b + c;
    console.log(`Sum is ${sum}`);
}

sumThree(1, 2 ,5);
sumThree(-10, -20 ,-5);
sumThree(12.45, 10.01, -90.34);
```

```
Sum is 8
Sum is -35
Sum is -67.88
```

funkcja jest podstawowym blokiem

Return

- Funkcja może zwrócić specyficzną wartość

```
function areTwoRealRoots(a, b, c) {
    const delta = b * b - 4 * a * c;
    if(delta > 0) {
        return true;
    }
    return false;
}

const a = 10, b = 100, c = 1;
const answer = areTwoRealRoots(a, b, c);
console.log(`There are given the following coefficicents ` +
    `of parabola: ${a}, ${b}, ${c}. Has the function two real roots ? ` +
    `${answer}`);
```

Return

- Zwracaną wartością jest *undefined* jeżeli nie ma *return*

```
function sumThree (a, b, c) {
    const sum = a + b + c;
    console.log(`Sum is ${sum}`);
}

const out = sumThree(1,2,3);
console.log(out); // undefined
```

Parametry

- Domyślne parametry zostały wprowadzone w ES2015

```
function presentPerson(name, age, city) {
    console.log(`This is ${name} (${age} years old) ` +
        `and lives in ${city}`);
}

presentPerson("Joseph", 40, "Poznan");
presentPerson("Kimball", 41, "LA");
presentPerson("Sandra", 32);
```

```
This is Joseph (40 years old) and lives in Poznan
This is Kimball (41 years old) and lives in LA
This is Sandra (32 years old) and lives in undefined
```

```
function presentPerson(name, age, city="Warsaw") {
    console.log(`This is ${name} (${age} years old) ` +
        `and lives in ${city}`);
}

presentPerson("Joseph", 40, "Poznan");
presentPerson("Kimball", 41, "LA");
presentPerson("Sandra", 32);
```

```
This is Joseph (40 years old) and lives in Poznan
This is Kimball (41 years old) and lives in LA
This is Sandra (32 years old) and lives in Warsaw
```

Parametry

- Destructuring (ES2015)

```
// destructuring
const person = {
    surname: "Kowalski",
    age: 30,
    city: "Warsaw",
};

const {surname, age, city} = person;
console.log(surname, age, city); // Kowalski 30 Warsaw
```

```
function presentPerson({name, age, city="Warsaw"}) {
    console.log(`This is ${name} (${age} years old) ` +
        `and lives in ${city}`);
}

presentPerson({
    name: "Joseph",
    age: 40,
    city: "Poznan"
}); // This is Joseph (40 years old) and lives in Poznan
presentPerson({
    name:"Kimball",
    age: 41,
    city: "LA"
}); // This is Kimball (41 years old) and lives in LA
presentPerson({
    name: "Sandra",
    age: 32,
}); // This is Sandra (32 years old) and lives in Warsaw
```

Parametry

- Niezdefiniowna liczba parametrów, operator rest (ES2015)

```
function getCapitals(...cities) {
    const capitals = [];
    for(let i = 0; i < cities.length; i++) {
        switch(cities[i]) {
            case "Poland":
                capitals.push("Warsaw");
                break;
            case "France":
                capitals.push("Paris");
                break;
            case "Germany":
                capitals.push("Berlin");
                break;
            //...
        }
    }

    return capitals;
}

getCapitals("Poland", "Germany", "Poland", "France", "Germany");
// ['Warsaw', 'Berlin', 'Warsaw', 'Paris', 'Berlin']
```

”function” declaration - podsumowanie

- „zwyczajna” funkcja zawiera:
  - słowo kluczowe *function*
  - następnie nazwa funkcji
  - dalej, lista parametrów
  - nawiasy klamrowe,
  - return

Jakkolwiek funkcja nie musi przyjmować parametrów czy zwracać wartości

```
function functionName(param1, param2, paramN) {
    // function body
    //...
    return someValue; // return is not necessary
}

const returnedValue = functionName("World", "Poland", "Europe");
```

```
function someOtherFunction() {
    // function body
    //...
}

someOtherFunction();
```

Konwencja nazewnicza

Użyj czasownika na nazwę funkcji

Funkcja, której nazwa rozpoczyna się od „*is”* powinno zwracać wartość boolean
isValidUserName, isPositiveNumber, isLoginAccepted

Funkcja powinna robić jedną rzecz

Ćwiczenie

1) Napisz funkcje, która loguje do konsoli informacje czy wszystkie trzy przesłane parametry są parzyste

2) Zaimplementuj funkcje, która symuluje rzut monetą, zwraca orzeł / reszka

3) Funkcja pobiera trzy współczynniki i zwraca pierwiastki paraboli

Ćwiczenie

4) Zmodyfikuj kod (patrz oparator *rest*) aby funkcja była wołana z jednym tablicą a nie listą parametrów (usuń operator *rest*)

5) Zaimplementuj funkcje, która zwraca losową liczbę z określonego zakresu. Jeżeli żaden parametr nie zostanie przesłany, załóż min = 0, max = 100

getRandom(5,10) → zwraca losową liczbę z przedziału <5; 10)

getRandom() → zwraca losową liczbę z przedziału <0; 100)

Rozwiązanie

```
getCapitals("Poland", "Germany", "Poland", "France", "Germany");
// ['Warsaw', 'Berlin', 'Warsaw', 'Paris', 'Berlin']

function getCapitals(cities) {
    const capitals = [];
    for(let i = 0; i < cities.length; i++) {
        switch(cities[i]) {
            case "Poland":
                capitals.push("Warsaw");
                break;
            case "France":
                capitals.push("Paris");
                break;
            case "Germany":
                capitals.push("Berlin");
                break;
            //...
        }
    }

    return capitals;
}

getCapitals(["Poland", "Germany", "Poland", "France", "Germany"]);
// ['Warsaw', 'Berlin', 'Warsaw', 'Paris', 'Berlin']
```

Function scope

- Zmienne zadeklarowane w obrębie funkcji, mają zakres tejże funkcji

```
//console.log(PI); -> ReferenceError: PI is not defined

function circleArea (radious) {
    //console.log(PI); -> ReferenceError: Cannot access 'PI' before initialization
    const PI = 3.14;
    const area = PI * radious * radious;

    // console.log(PI); -> 3.14
    return area;
}

// console.log(PI); -> ReferenceError: PI is not defined
circleArea(10.5);
```

Function scope

- Zmienne zadeklarowane w obrębie funkcji, mają zakres tejże funkcji

```
// console.log(temp); -> ReferenceError: temp is not defined
function displayThreeBiggerNoThan(ref) {
    //console.log(temp); -> ReferenceError: Cannot access 'temp' before initialization
    let temp = ref + 1;
    console.log(temp);
    temp = ref + 2;
    console.log(temp);
    temp = ref + 3;
    console.log(temp);
    //console.log(temp); -> 5
}

// console.log(temp); -> ReferenceError: temp is not defined
```

Hoisting

- Do zmiennych zadeklarowanych przez *var* możemy odnosić się przed miejscem ich deklaracji

```
// console.log(area); -> ReferenceError: area is not defined
function getTriangleDescriptor(a, b, c, h) {
    // console.log(perimeter); -> undefined (no error)
    var area = 0.5 * a * h;
    var perimeter = a + b + c;

    // console.log(perimeter); -> result of a + b + c
    return {
        area,       // area: area,
        perimeter,  // perimeter: perimeter,
    }
}
// console.log(area); -> ReferenceError: area is not defined
```

var rządzi się własnymi prawami !

Hoisting

- Funkcje są dostępne podobnie jak zmienne zadeklarowane przez *var*; t.j. przed deklaracją

```
console.log(getRectangleArea(10, 100)); // 1000

function getRectangleArea(a, b) {
    return a * b;
}

console.log(getRectangleArea(10, 20)); // 200
```

Świat JavaScript!
Brak błędu !!!

```
console.log(getRectangleArea(10, 100)); // 1000

function getRectangleArea(a, b) {
    return a * b;
}
```

jest interpreto-wane jako

```
function getRectangleArea(a, b) {
    return a * b;
}

console.log(getRectangleArea(10, 100)); // 1000
```

Hoisting

- Funkcje są dostępne podobnie jak zmienne zadeklarowane przez *var*; t.j. przed deklaracją

```
// console.log(getSquareArea(10)) -> ReferenceError: getSquareArea is not defined
// console.log(getArea("square", 10)); -> 100

function getArea(figureName, a, b) {
    if (figureName === "rectangle") {
        return getRectangleArea(a, b);
    } else if (figureName === "square") {
        return getSquareArea(a);
    }

    function getSquareArea(side) {
        return side * side;
    }

    function getRectangleArea(side1, side2) {
        return side1 * side2;
    }
}

// console.log(getSquareArea(10)) -> ReferenceError: getSquareArea is not defined
// console.log(getArea("rectangle", 1, 2)); -> 2
```

Ciekawy przypadek – wyciek pamięci

- Tworzona jest zmienna globalna *area*. *Garbage collector* może mieć problemy aby ją usunąć ponieważ nadal tj. (po wyjściu z funkcji) jest do niej dostęp

```
console.log(area); // Uncaught ReferenceError: area is not defined

function getSimpleTriangleArea() {

    var a = 100;
    var h = 10;
    area = 0.5 * a * h;

    return area;
}

getSimpleTriangleArea();
console.log(area); // 500
```

Tworzenie funkcji

- Funkcja może być tworzona na trzy sposoby
  - function declaration
  - **function expression**
  - Function contructor

Wyrażenia funkcyjne

```
function someFunction(a, b) {
    //...
}
```

function declaration

```
const someFunction = function (a, b) {
    //...
}
```

anonimowe wyrażenie funkcyjne

```
const funToBeCalled = function someFunction(a, b) {
    //...
}
```

nazwane wyrażenie funkcyjne

- Wyrażenia funkcyjne nie potrzebują mieć nazwy
- W tym przypadku wyrażenie funkcyjne jest przypisywane do zmiennej, która później może być przesłana jako parametr funkcji

Wyrażenie funkcyjne

- „Zwykła funkcja” oraz wyrażenie funkcyjne po prawej stronie. Są wywoływane w ten sam sposób. Jedną z różnic jest to, że w tym drugim przypadku nie możemy zawołać wyrażenia przed deklarację (patrz hoisting)
- Wyrażenie funkcyjne jest tworzone gdy *execution flow* trafia w miejsce deklaracji

```
getSquareRoot(10, 20, 30);
// {num1: 3.1622776601683795, num2: 4.47213595499958, num
function getSquareRoot (a, b, c) {
    return {
        num1: Math.sqrt(a),
        num2: Math.sqrt(b),
        num3: Math.sqrt(c),
    };
}
```

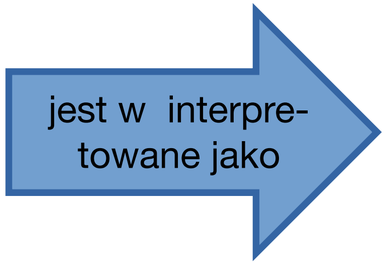
Funkcja anonimowa

```
var getSquareRoot = function (a, b, c) {
    return {
        num1: Math.sqrt(a),
        num2: Math.sqrt(b),
        num3: Math.sqrt(c),
    };
};

getSquareRoot(10, 20, 30);
```

Wyrażenie funkcyjne

- Hoisting po raz wtóry

```
getSquareRoot(10, 20, 30);

var getSquareRoot = function (a, b, c) {
    return {
        num1: Math.sqrt(a),
        num2: Math.sqrt(b),
        num3: Math.sqrt(c),
    };
};
```




```
var getSquareRoot;
getSquareRoot(10, 20, 30);

getSquareRoot = function (a, b, c) {
    return {
        num1: Math.sqrt(a),
        num2: Math.sqrt(b),
        num3: Math.sqrt(c),
    };
};
```

```
getSquareRoot(10, 20, 30);

function getSquareRoot (a, b, c) {
    return {
        num1: Math.sqrt(a),
        num2: Math.sqrt(b),
        num3: Math.sqrt(c),
    };
};
```


```
function getSquareRoot (a, b, c) {
    return {
        num1: Math.sqrt(a),
        num2: Math.sqrt(b),
        num3: Math.sqrt(c),
    };
};

getSquareRoot(10, 20, 30);
```

Wyrażenia funkcyjne i callbacks

- Wyrażenie funkcyjne jest pomocne (może poprawić czytelność) kiedy ma być użyte w roli *callback'a* – przesłane jako argument innej funkcji

```
const addRandomValue = function (ref) {
    return ref + Math.random();
}

const multiplyByTwo = function (ref) {
    return ref * 2;
}

function process(arr, f) {
    for (let i = 0; i < arr.length; i++) {
        arr[i] = f(arr[i]);
    }
}

const integerArr = [10, 20, -10, -20];
process(integerArr, addRandomValue);
process(integerArr, multiplyByTwo);
```

```
integerArr
(4) [10.830926380580038, 20.321396083870216, -9.64960195158443, -19.31948
5825598317]
  0: 21.661852761160077
  1: 40.64279216774043
  2: -19.29920390316886
  3: -38.638971651196634
  length: 4
```

addRandomValue

```
integerArr
(4) [21.661852761160077, 40.64279216774043, -19.29920390316886, -38.63897
1651196634]
  0: 21.661852761160077
  1: 40.64279216774043
  2: -19.29920390316886
  3: -38.638971651196634
  length: 4
▶ [[Prototype]]: Array(0)
```

multiplyByTwo

Ćwiczenie

- Zaimplementuj funkcje, która pobiera jako argumenty inną funkcję wykonującą albo mnożenie albo odejmowanie. Kolejne parametry stanowią liczby, który są przedmiotem obliczeń

  performOperation(subtract, 10, 20); // woła wewnętrznie subtract z parametrami 10 i 20
  output: ”The result is -10”
  performOperation(subtract, 10, 20, 30, 50); // woła wewnętrznie subtract z parametrami 10, 20, 30, 50
  output: ”The result is -90”
  performOperation(multiply, 10, 20, 30, 50, 100); // woła wewnętrznie multiply z parametrami 10, … 100
  output: ”The result is 30000000”

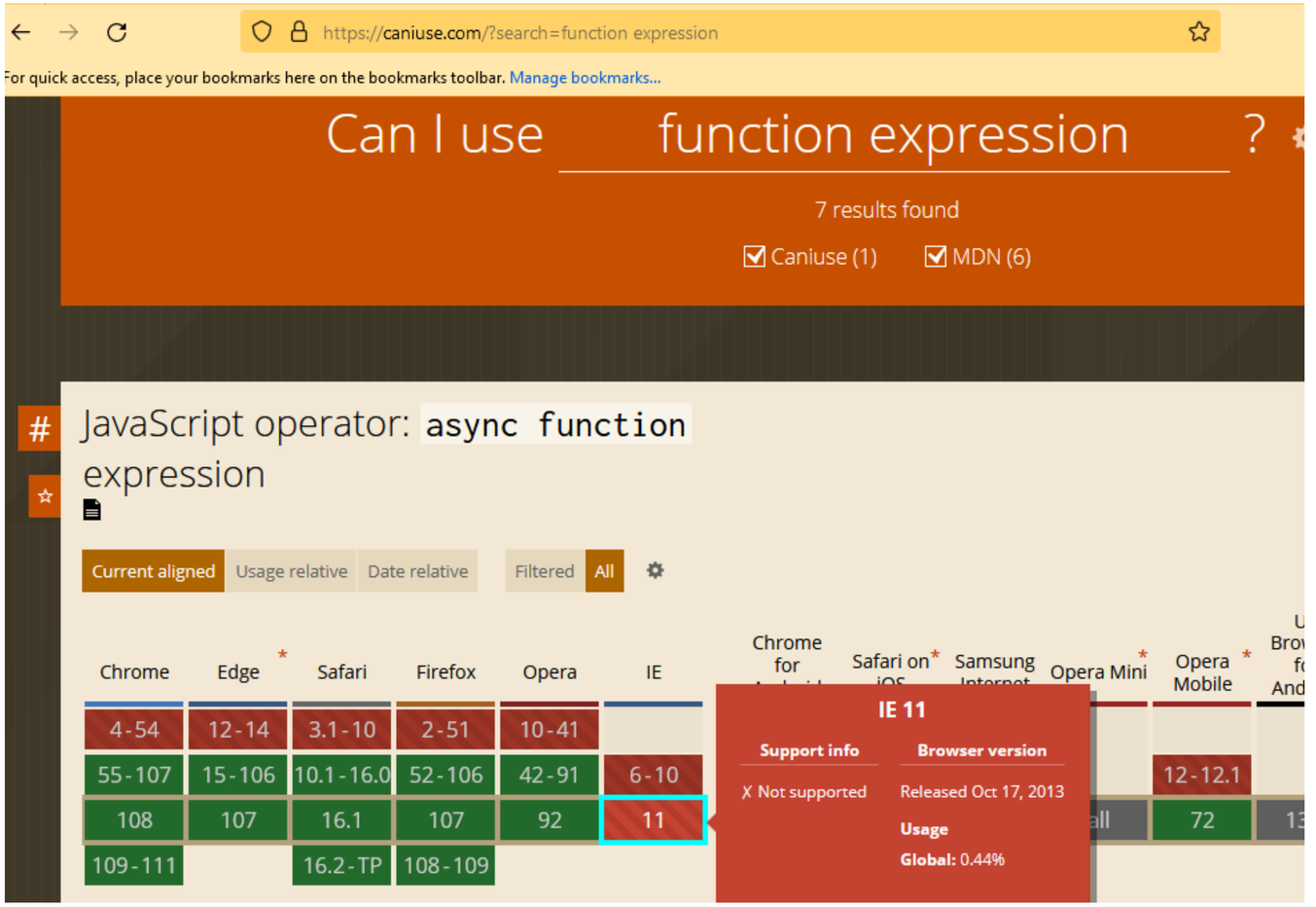
Rozwiązanie

```
function performOperation(operFun, ...operands) {
    // operands is an array like [10, 20, 5, 16]
    const result = operFun(operands);
    console.log(`The result is ${result}`);
}

const subtract = function(arr) {
    let diff = arr[0];
    for (let i = 1; i < arr.length; ++i) {
        diff -= arr[i];
    }
    return diff;
}
```

```
const multiply = function(arr) {
    let product = 1;
    for (let i = 0; i < arr.length; ++i) {
        product *= arr[i];
    }
    return product;
}

performOperation(multiply, 10, 20, 30, 50, 100);
performOperation(subtract, 10, 20, 30, 50);
```

Wyrażenie funkcyjne

- Dostępność



Funkcja strzałkowa

- Zwięzła alternatywa

```
let someFunc = function (param1, param2, /*...,*/ paramN) {
    return expression;
}

let someArrowFunc = (param1, param2, /*...,*/ paramN) => expression;

let computeSum = function (a, b, c) {
    return a + b + c;
}

let computeSumArrow = (a, b, c) => a + b + c;
```

Funkcja strzałkowa

- Pojedynczy parametr

```
let someFuncSingleParam = function (param1) {
    return expression;
}

let someFuncSingleParamArrow = param1 => expression;

let getRandomValue = function (max) {
    return Math.round(Math.random() * max);
}

let getRandomValueArrow = max => Math.round(Math.random() * max);
```

Funkcja strzałkowa

- Multiline

```
let someFuncMultiLineNoArrow = function (param1, param2, /*...,*/ paramN) {
    // some multiline function body ...
    return expression;
}

let someArrowFuncMultiLine = (param1, param2, /*...,*/ paramN) => {
    // some multiline function body ...
    return expression;
}

let getMathValuesNoArrow = function (param1 = 10, param2 = 20) {
    return {
        product: param1 * param2,
        sum: param1 + param2,
    };
}

let getMathValuesArrowFunc = (param1 = 10, param2 = 20) => {
    return {
        product: param1 * param2,
        sum: param1 + param2,
    };
}
```

Funkcja strzałkowa

- vs wyrażenie funkcyjne
  - domyślny *return*
  - brak potrzeby użycia nawiasu w przypadku pojedynczego wyrażenia
  - nie może być zawołana z operatorem *new*
  - wiąże *this*
  - nie ma parametru *arguments* oraz meta operatrora *new.target*

Funkcja strzałkowa

- Zalety

```
let studentMIT = {
    people: ["Kalinka", "Marta", "Faustine"],
    showDelay1() {
      const ONE_SEC = 1000;
      setTimeout(function () {
        for (let i = 0; i < this.people.length; ++i) {
          // Uncaught TypeError: Cannot read properties of undefined (reading 'length')
          console.log(this.people[i]); // error this is bound with the global object
        }
      }, ONE_SEC);
    },
    showDelay2: function () {
      const ONE_SEC = 1000;
      setTimeout(() => {
        for (let i = 0; i < this.people.length; ++i) {
          console.log(this.people[i]); // this is bound with studentMIT
        }
      }, ONE_SEC);
    }
  }
  studentMIT.showDelay2();
  studentMIT.showDelay1();
```

Dziękuję !