Programowanie w JavaScript

WSEI

#szkoła programowania

Magiczne słówko this

Obiekty w JS tworzymy w konkretnym celu. Aby zgrupować jakieś właściwości. Z tego wynika że zazwyczaj jest jeden obiekt wywoływany w różnych kontekstach. Aby móc swobodnie operować takim obiektem niezależnie od wywoływanego kontekstu używamy słówka **this** które wskazuje na obiekt, zamiast używać nazwy obiektu ponieważ nie zawsze ją znamy.

This wskazuje zawsze na obiekt w kontekście którego zostało wywołane. Jest to bardzo ważne.

```
let person = {
    name: 'Mateusz',
    age: 32,
    color: white,
    myName: function() {
        console.log(person.name);
        console.log(this.name);
    }
}
```

Magiczne słówko this

W przykładzie obok funkcja myName() nie ma wiedzy o nazwie obiektu w kontekście którego zostanie wywołana. I nie powinna mieć. Funkcje powinny być niezależne od kontektu

```
let person = {
    name: 'Mateusz',
    age: 32
}
let person2 = {
    name: 'Piotr',
    age: 32
}
function myName() {
    console.log(this.name);
}
person.myName = myName;
person2.myName = myName;

person.myName(); // Mateusz
person2.myName(); // Piotr
```

Ręczne tworzenie wielu podobnych lub identycznych obiektów nie jest efektywne. Aby móc to robić automatycznie w JS mamy tak zwany **Constructor funtion**. Jest to specjalna funkcja której wywołanie poprzedzamy słówkiem **new**, nazwę piszemy **z dużej litery** i służy do tworzenia podobnych obiektów na podstawie „szablonu”.

```
function MyFirstConstructorFunction() {
    console.log('To jest konstruktor');
}
```

Wywoływanie konstruktora jest bardzo proste:

```
function Person() {
    this.name = 'Mateusz';
    console.log(this.name);
}

let user1 = new Person(); // Mateusz
let user2 = new Person(); // Mateusz
let user3 = new Person(); // Mateusz
```

Inny przykład

```
function Person(personName) {
    this.name = personName;
    this.country = "Poland";
    this.sayMyName = function(){
        console.log(`My name is ${this.name}`)
    }
}

let user1 = new Person('Mateusz');
let user2 = new Person('Ala');
let user3 = new Person('Kot');

user1.sayMyName(); // My name is Matesz
user2.sayMyName(); // My name is Ala
user3.sayMyName(); // My name is Kot
```

Constructor function

Funkcję tworzącą obiekt na podstawie szablonu, którą piszemy z dużej litery i wywołujemy z użyciem słówna new to kontruktor

Konstruktor służy do ustawiania stanu początkowego obiektu po jego utworzeniu

Konstruktor jest funkcją (nie arrow function)

Używany słówka this wewnątrz konstruktora ponieważ nie znamy nazwy kontekstu w obrębie którego będziemy wywoływać konstruktor

Nieodłącznym elementem konstruktorów i obiektów jest Prototype. Każdy obiekt zajmuje w pamięci miejsce. Jeśli przy pomocy konstruktora zaczniemy tworzyć duże ilości obiektów może nam się skończyć pamięć. Z pomocą przychodzi nam **Prototype**

Powtarzające się rzeczy:

1. **Country**
2. **sayMyName()**

```
function Person(personName) {
    this.name = personName;
    this.country = "Poland";
    this.sayMyName = function(){
        console.log(`My name is ${this.name}`)
    }
}

let users = [];
for(let i = 0; i < 1000000; i++) {
    users.push(new Person(`person ${i}`));
}
```

PAMIĘTAJ!

W JS wszystko poza typami prostymi (string, numer, boolean, null, undefined) jest obiektem!

A zatem funkcje również

Prototype

Każdy obiekt (funkcja) posiada pole o nazwie **prototype**

prototype jest również obiektem

Obiekt prototype jest widoczny dla każdego obiektu który zostanie stworzony na podstawie konstruktora

```
> function prototypeTest() {}
< undefined
> console.dir(prototypeTest);
  ▼ ƒ prototypeTest() i
      arguments: null
      caller: null
      length: 0
      name: "prototypeTest"
    ▶ prototype: {constructor: ƒ}
    ▶ __proto__: ƒ ()
      [[FunctionLocation]]: VM386:1
    ▶ [[Scopes]]: Scopes[2]
```

A zatem możemy zaoszczędzić sporo miejsca poprzez wyciągnięcie tych samych rzeczy do pola prototype. Obiekt ten jest dostępny dla każdego obiektu stworzonego na podstawie konstruktora a skoro jest to obiekt a jak wiemy pojedynczy obiekt zajmuje w pamięci jedno miejsce a zmienne (właściwości) przechowują jedynie referencję to dodając cokolwiek do pola prototype oszczędzamy miejsce w pamięci i tworzymy reużywalny kod.

```
function Person(personName) {
    this.name = personName;
}

Person.prototype.country = 'Poland';
Person.prototype.sayMyName = function () {
    console.log(`My name is ${this.name}`)
}
```

Hoisting (nie hosting) jest to automatyczne przenoszenie pewnych instrukcji na górę kodu przed wystartowaniem aplikacji a po interpretacji kodu przez silnik JavaScriptu.

Przenoszone są tylko i wyłącznie deklaracje zmiennych i deklaracje funkcji. Wyrażenia funkcyjne nie są przenoszone.

```
myName(); // To nie zadziała
let myName = function myNameIs(){
    console.log('Mateusz');
}
```

```
// Hoiting
function muNameIs() {};
// Hoiting

muNameIs();
function myNameIs(){
    console.log('Mateusz');
}
```