

# EcmaScript 6

# Obiektowość w ES6

# Wprowadzenie

# Wprowadzenie

Jak już wiesz w języku JavaScript obiektowość jest zaimplementowana dzięki zasadzie prototypów. W innych językach jest stosowana klasowość np. w PHP czy Java.

Standard ES6 jedynie udostępnia słowa kluczowe takie jak np. **class**, **extends** czy **super**, które ułatwiają implementację. Nie zmieniają jednak zasad działania języka.

Przykład zgodny ze standardem ES6 możemy zobaczyć obok.

```
class Animal {
  constructor(name) {
    this.name = name;
  }
  getName() {
    return this.name;
  }
}
class Cat extends Animal {
  constructor(name, age) {
    super(name);
    this.age = age;
  }
}
let cat = new Cat("Filemon", 4);
console.log(cat);
// Cat {name: "Filemon", age: 4}
```

**class**

# class

Słowo kluczowe **class** pozwala nam tworzyć konstruktor, dzięki któremu będziemy mogli stworzyć instancje obiektu.

```
class Animal {  
}  
let animal = new Animal();  
console.log(animal); // Animal {}
```

# class

W pierwszej kolejności używamy słowa kluczowego **class** a następnie wprowadzamy nazwę konstruktora, którą zwyczajowo piszemy wielką literą. Wewnątrz nawiasów klamrowych tworzymy ciało naszej „klasy”.

Instancje obiektu tworzymy za pomocą słowa kluczowego **new**.

```
class Animal {  
}  
let animal = new Animal();  
console.log(animal); // Animal {}
```

# UWAGA! class

Zapis zgodny ze standardem ES6 zachowuje się identycznie jak kod, który był wykorzystywany przed jego wdrożeniem.

Zasada działania jest ta sama, zmienił się jedynie sposób zapisu.

**JavaScript nie ma klas znanych z innych języków programowania.**

```
class Animal1 {  
    //ciało klasy  
}  
let a1 = new Animal1();  
var Animal2 = function() {  
    //ciało konstruktora  
}  
var a2 = new Animal2();  
console.log(a1); //Animal1 {}  
console.log(a2); //Animal2 {}  
console.log(typeof Animal1); //function  
console.log(typeof Animal2); //function
```



# class i constructor

Przed ES6 stan początkowy dla utworzonej instancji obiektu ustawialiśmy dzięki parametrom konstruktora.

```
var Animal = function(name) {  
  this.name = name;  
}  
var a = new Animal("Filemon");  
console.log(a);  
// Animal {name: "Filemon"}
```

Obeenie możemy wykorzystać metodę **constructor**, która zawsze przyjmuje taką nazwę i to ona ustawia stany początkowe (właściwości) dla obiektu.

```
class Animal {  
  constructor(name) {  
    this.name = name;  
  }  
}  
let a = new Animal("Filemon");  
console.log(a);  
// Animal {name: "Filemon"}
```

# class i constructor

Metody w ciele klasy nie mogą być rozdzielone przecinkami. Każda metoda musi mieć nazwę, nawiasy klamrowe i ewentualnie parametry.

Jeśli chcemy zmieniać właściwości instancji obiektu, to musimy używać słowa kluczowego **this**, tak jak do tej pory.

```
class Animal {  
  constructor(name) {  
    this.name = name;  
  }  
  getName() {  
    return this.name;  
  }  
  setName(name) {  
    this.name = name;  
  }  
}  
let a = new Animal("Filemon");  
console.log(a.getName()); // Filemon  
a.setName("Mruczek");  
console.log(a.getName()); // Mruczek
```

# class i metody

Kod bez wykorzystania rozwiązań z ES6 jest mniej czytelny i może przysparzać więcej problemów. Używamy prototypowania, aby nie zajmować niepotrzebnie pamięci na definicje naszych metod w każdej instancji obiektu.

```
var Animal = function(name) {  
    this.name = name;  
}  
Animal.prototype.getName = function() {  
    return this.name;  
}  
Animal.prototype.setName = function(name) {  
    this.name = name;  
}  
var a = new Animal("Filemon");  
console.log(a.getName()); // Filemon  
a.setName("Mruczek");  
console.log(a.getName()); // Mruczek
```

A large dashed circle in a dark blue-grey color, centered on the page. The word "extends" is written in the center of the circle.

**extends**

# extends

Słowo kluczowe **extends** pozwala nam dziedziczyć właściwości i metody z innej klasy.

Należy to rozumieć jako sposób rozszerzania bazowej klasy o dodatkowe właściwości i metody.

W naszym przykładzie klasa **Bird** rozszerza klasę **Animal** o metodę **fly**.

To znaczy, że obiekty klasy **Bird** będą potrafiły latać. Ale inne zwierzęta tej umiejętności nie będą miały. Natomiast klasa **Bird** będzie miała wszystkie właściwości klasy **Animal**.

```
class Animal {  
    //ciało klasy Animal  
}  
class Bird extends Animal {  
    fly() {  
        //ciało metody fly  
    }  
}
```

## extends i super

W przykładzie obok klasa **Bird** korzysta z metody **getName()**, której nie ma zadeklarowanej.

To dzięki dziedziczeniu może skorzystać z tej metody i własności **name**, która jest zdefiniowana w klasie **Animal**.

```
class Animal {  
    constructor(name) {  
        this.name = name;  
    }  
    getName() {  
        return this.name;  
    }  
}  
  
class Bird extends Animal {  
    fly() {  
        console.log(this.getName()+" lata");  
    }  
}  
  
let birdy = new Bird("Ćwirek");  
birdy.fly();  
// Ćwirek lata
```

## extends i super

W przykładzie obok klasa **Bird** korzysta z metody **getName()**, której nie ma zadeklarowanej.

To dzięki dziedziczeniu może skorzystać z tej metody i własności **name**, która jest zdefiniowana w klasie **Animal**.

```
class Animal {  
    constructor(name) {  
        this.name = name;  
    }  
    getName() {  
        return this.name;  
    }  
}  
class Bird extends Animal {  
    fly() {  
        console.log(this.getName()+" lata");  
    }  
}  
let birdy = new Bird("Ćwirek");  
birdy.fly();  
// Ćwirek lata
```

Przykład dziedziczenia.

# extends i super

Czasami potrzebujemy rozszerzyć samą metodę tj. wywołać metodę z klasy bazowej a potem wykonać dodatkowe czynności.

Z pomocą przychodzi nam słowo kluczowe **super**, które pozwala wywołać metodę z klasy bazowej.

W naszym przykładzie definiujemy na nowo **constructor** z dodatkowym parametrem **age**. W ciele tej metody korzystamy z **super**, gdzie przekazujemy odpowiedni parametr i wykonujemy dalsze operacje.

```
class Animal {  
  constructor(name) {  
    this.name = name;  
  }  
}  
  
class Cat extends Animal {  
  constructor(name, age) {  
    super(name);  
    this.age = age;  
  }  
}  
  
let animal = new Animal("Filemon")  
  
console.log(animal);  
//Animal {name: "Filemon"}  
let cat = new Cat("Mruczek", 4);  
console.log(cat);  
//Cat {name: "Mruczek", age: 4}
```



# extends i super

Czasami potrzebujemy rozszerzyć samą metodę tj. wywołać metodę z klasy bazowej a potem wykonać dodatkowe czynności.

Z pomocą przychodzi nam słowo kluczowe **super**, które pozwala wywołać metodę z klasy bazowej.

W naszym przykładzie definiujemy na nowo **constructor** z dodatkowym parametrem **age**. W ciele tej metody korzystamy z **super**, gdzie przekazujemy odpowiedni parametr i wykonujemy dalsze operacje.

```
class Animal {  
  constructor(name) {  
    this.name = name;  
  }  
}  
class Cat extends Animal {  
  constructor(name, age) {  
    super(name);  
    this.age = age;  
  }  
}  
let animal = new Animal("Filemon")  
  
console.log(animal);  
//Animal {name: "Filemon"}  
let cat = new Cat("Mruczek", 4);  
console.log(cat);  
//Cat {name: "Mruczek", age: 4}
```

## extends i super

W przypadku innych metod niż **constructor** słowo kluczowe **super** wykorzystamy w inny sposób.

Samo słowo **super** bez wywołania wskazuje na klasę, z której dziedziczyliśmy. Dzięki temu będziemy mogli odwołać się do dowolnej metody.

```
class Foo {  
    constructor(a, b) {  
        this.a = a;  
        this.b = b;  
    }  
    multiply() {  
        return this.a * this.b;  
    }  
}  
class Bar extends Foo {  
    constructor(a, b, c) {  
        super(a, b);  
        this.c = c;  
    }  
    multiply() {  
        return super.multiply() * this.c;  
    }  
}
```

# extends i super

W przypadku innych metod niż **constructor** słowo kluczowe **super** wykorzystamy w inny sposób.

Samo słowo **super** bez wywołania wskazuje na klasę, z której dziedziczyliśmy. Dzięki temu będziemy mogli odwołać się do dowolnej metody.

W naszym przykładzie w metodzie **multiply** (w klasie **Bar**) wywołujemy metodę o tej samej nazwie, tylko że z klasy **Foo**.

```
class Foo {  
    constructor(a, b) {  
        this.a = a;  
        this.b = b;  
    }  
    multiply() {  
        return this.a * this.b;  
    }  
}  
class Bar extends Foo {  
    constructor(a, b, c) {  
        super(a, b);  
        this.c = c;  
    }  
    multiply() {  
        return super.multiply() * this.c;  
    }  
}
```

# Klasa abstrakcyjna

Możemy powiedzieć, że klasa `Animal` jest klasą abstrakcyjną. Nie mamy żadnego słowa kluczowego na to, co będzie naszą umowną definicją. Ale po co? Wyobraź sobie, że ktoś spróbuje stworzyć obiekt typu `Animal`. Jakie to będzie zwierzę? Żadne.

Nie ma takiego stworzenia, które byłoby tylko zwierzęciem. Są gatunki, typy itd. W niektórych językach projektowania np. w Javie programiści mają dostępne słowo `abstract`, którego używają na początku tworzenia klasy. My nie mamy, dlatego będzie to określenie umowne.

```
class Animal {
    constructor(name) {
        this.name = name;
    }
    getName() {
        return this.name;
    }
}

class Bird extends Animal {
    fly() {
        console.log(this.getName()+" lata");
    }
}

let animal = new Animal("Ćwerek");
animal.getName();
```

# Klasa abstrakcyjna

Możemy powiedzieć, że klasa **Animal** jest klasą abstrakcyjną. Nie mamy żadnego słowa kluczowego na to, co będzie naszą umowną definicją. Ale po co? Wyobraź sobie, że ktoś spróbuje stworzyć obiekt typu **Animal**. Jakie to będzie zwierzę? Żadne.

Nie ma takiego stworzenia, które byłoby tylko zwierzęciem. Są gatunki, typy itd. W niektórych językach projektowania np. w Javie programiści mają dostępne słowo **abstract**, którego używają na początku tworzenia klasy. My nie mamy, dlatego będzie to określenie umowne.

```
class Animal {  
  constructor(name) {  
    this.name = name;  
  }  
  getName() {  
    return this.name;  
  }  
}  
  
class Bird extends Animal {  
  fly() {  
    console.log(this.getName()+" lata");  
  }  
}  
  
let animal = new Animal("Ćwerek");  
animal.getName();
```

JavaScript NIE zwróci nam tutaj błędu, ale nie wiemy jakie to konkretnie zwierzę.

# Podsumowanie

# Obiektość w ES6 – podsumowanie

Omówmy sobie teraz nasz kod z pierwszego slajdu z tego działu.

Mamy klasę **Animal** oraz dwie metody.

Pierwsza to **constructor**, która ustawia początkową wartość dla właściwości **name**.

Druga to **getName**, która zwraca nam aktualną wartość dla właściwości **name**.

```
class Animal {  
  constructor(name) {  
    this.name = name;  
  }  
  getName() {  
    return this.name;  
  }  
}  
  
class Cat extends Animal {  
  constructor(name, age) {  
    super(name);  
    this.age = age;  
  }  
}  
  
let cat = new Cat("Filemon", 4);  
console.log(cat);  
// Cat {name: "Filemon", age: 4}
```



# Obiektowość w ES6 – podsumowanie

Następnie mamy drugą klasę – **Cat**, która dziedziczy po klasie **Animal**.

```
class Animal {  
  constructor(name) {  
    this.name = name;  
  }  
  getName() {  
    return this.name;  
  }  
}  
class Cat extends Animal {  
  constructor(name, age) {  
    super(name);  
    this.age = age;  
  }  
}  
let cat = new Cat("Filemon", 4);  
console.log(cat);  
// Cat {name: "Filemon", age: 4}
```



# Obiektość w ES6 – podsumowanie

Następnie mamy drugą klasę – **Cat**, która dziedziczy po klasie **Animal**.

Można powiedzieć, że kopiujemy (w dużym uproszczeniu) właściwości i metody do klasy **Cat** z klasy **Animal**.

```
class Animal {  
  constructor(name) {  
    this.name = name;  
  }  
  getName() {  
    return this.name;  
  }  
}  
class Cat extends Animal {  
  constructor(name, age) {  
    super(name);  
    this.age = age;  
  }  
}  
let cat = new Cat("Filemon", 4);  
console.log(cat);  
// Cat {name: "Filemon", age: 4}
```

# Obiektość w ES6 – podsumowanie

Ponieważ chcemy rozszerzyć **constructor** o dodatkowy parametr, to definiujemy go w ciele klasy **Cat** i używamy słowa kluczowego **super**, aby wywołać funkcję **constructor** z klasy bazowej.

Jest to na tyle istotne, że nie powielamy naszego kodu, tylko korzystamy z tego, co wcześniej napisaliśmy.

```
class Animal {  
  constructor(name) {  
    this.name = name;  
  }  
  getName() {  
    return this.name;  
  }  
}  
class Cat extends Animal {  
  constructor(name, age) {  
    super(name);  
    this.age = age;  
  }  
}  
let cat = new Cat("Filemon", 4);  
console.log(cat);  
// Cat {name: "Filemon", age: 4}
```

# Obiektość w ES6 – podsumowanie

Ponieważ chcemy rozszerzyć **constructor** o dodatkowy parametr, to definiujemy go w ciele klasy **Cat** i używamy słowa kluczowego **super**, aby wywołać funkcję **constructor** z klasy bazowej.

Jest to na tyle istotne, że nie powielamy naszego kodu, tylko korzystamy z tego, co wcześniej napisaliśmy.

```
class Animal {  
  constructor(name) {  
    this.name = name;  
  }  
  getName() {  
    return this.name;  
  }  
}  
class Cat extends Animal {  
  constructor(name, age) {  
    super(name);  
    this.age = age;  
  }  
}  
let cat = new Cat("Filemon", 4);  
console.log(cat);  
// Cat {name: "Filemon", age: 4}
```

# Obiektowość w ES6 – podsumowanie

Następnie wykonujemy operacje związane z drugim parametrem.

```
class Animal {  
  constructor(name) {  
    this.name = name;  
  }  
  getName() {  
    return this.name;  
  }  
}  
class Cat extends Animal {  
  constructor(name, age) {  
    super(name);  
    this.age = age;  
  }  
}  
let cat = new Cat("Filemon", 4);  
console.log(cat);
```

**Cat {name: "Filemon", age: 4}**

# Obiektość w ES6 – podsumowanie

Następnie wykonujemy operacje związane z drugim parametrem.

W tym przypadku przypisujemy go do właściwości **age**.

```
class Animal {  
  constructor(name) {  
    this.name = name;  
  }  
  getName() {  
    return this.name;  
  }  
}  
class Cat extends Animal {  
  constructor(name, age) {  
    super(name);  
    this.age = age;  
  }  
}  
let cat = new Cat("Filemon", 4);  
console.log(cat);
```

**Cat {name: "Filemon", age: 4}**

# Obiektość w ES6 – podsumowanie

Następnie wykonujemy operacje związane z drugim parametrem.

W tym przypadku przypisujemy go do właściwości **age**.

Choć w klasie **Cat** jawnie deklarowaliśmy jedynie **age**, to widać w konsoli, że obiekt ten ma również właściwość **name**.

```
class Animal {  
  constructor(name) {  
    this.name = name;  
  }  
  getName() {  
    return this.name;  
  }  
}  
class Cat extends Animal {  
  constructor(name, age) {  
    super(name);  
    this.age = age;  
  }  
}  
let cat = new Cat("Filemon", 4);  
console.log(cat);
```

**Cat** {name: "Filemon", age: 4}

# Obiektość w ES6 – podsumowanie

Następnie wykonujemy operacje związane z drugim parametrem.

W tym przypadku przypisujemy go do właściwości **age**.

Choć w klasie **Cat** jawnie deklarowaliśmy jedynie **age**, to widać w konsoli, że obiekt ten ma również właściwość **name**.

Dzieje się tak, ponieważ dziedziczyliśmy po klasie **Animal** i dodatkowo parametr **name** przekazaliśmy do metody **constructor** z klasy **Animal** dzięki słowu kluczowemu **super**.

```
class Animal {  
  constructor(name) {  
    this.name = name;  
  }  
  getName() {  
    return this.name;  
  }  
}  
class Cat extends Animal {  
  constructor(name, age) {  
    super(name);  
    this.age = age;  
  }  
}  
let cat = new Cat("Filemon", 4);  
console.log(cat);
```

```
Cat {name: "Filemon", age: 4}
```



# Obiektowość w ES5



# Obiektowość w ES5

Przypomnijmy sobie jeszcze jak taką strukturę zapisalibyśmy bez dobrodziejstw standardu ES6.

```
var Animal = function(name) {  
    this.name = name;  
}  
Animal.prototype.getName = function() {  
    return this.name;  
}  
var Cat = function(name, age) {  
    Animal.call(this, name, age);  
    this.age = age;  
}  
Cat.prototype =  
Object.create(Animal.prototype);  
Cat.prototype.constructor = Cat;  
var cat = new Cat("Filemon", 4);  
console.log(cat);
```

**Cat {name: "Filemon", age: 4}**

# Obiektowość w ES5

Przypomnijmy sobie jeszcze jak taką strukturę zapisalibyśmy bez dobrodziejstw standardu ES6.

W pierwszej kolejności tworzymy konstruktor Animal – w ES6 robiliśmy to za pomocą słowa kluczowego **class**.

```
var Animal = function(name) {  
    this.name = name;  
}  
Animal.prototype.getName = function() {  
    return this.name;  
}  
var Cat = function(name, age) {  
    Animal.call(this, name, age);  
    this.age = age;  
}  
Cat.prototype =  
    Object.create(Animal.prototype);  
Cat.prototype.constructor = Cat;  
var cat = new Cat("Filemon", 4);  
console.log(cat);
```

Cat {name: "Filemon", age: 4}

# Obiektowość w ES5

Przypomnijmy sobie jeszcze jak taką strukturę zapisalibyśmy bez dobrodziejstw standardu ES6.

W pierwszej kolejności tworzymy konstruktor `Animal` – w ES6 robiliśmy to za pomocą słowa kluczowego **class**.

Jednocześnie konstruktor ustawia właściwość `name`. W ES6 robiła to metoda **constructor** z odpowiednim parametrem.

```
var Animal = function(name) {  
  this.name = name;  
}  
Animal.prototype.getName = function() {  
  return this.name;  
}  
var Cat = function(name, age) {  
  Animal.call(this, name, age);  
  this.age = age;  
}  
Cat.prototype =  
  Object.create(Animal.prototype);  
Cat.prototype.constructor = Cat;  
var cat = new Cat("Filemon", 4);  
console.log(cat);
```

**Cat {name: "Filemon", age: 4}**

# Obiektowość w ES5

Przypomnijmy sobie jeszcze jak taką strukturę zapisalibyśmy bez dobrodziejstw standardu ES6.

W pierwszej kolejności tworzymy konstruktor `Animal` – w ES6 robiliśmy to za pomocą słowa kluczowego **class**.

Jednocześnie konstruktor ustawia właściwość `name`. W ES6 robiła to metoda **constructor** z odpowiednim parametrem.

Już teraz wiadomo skąd taka nazwa a nie inna.

```
var Animal = function(name) {  
    this.name = name;  
}  
Animal.prototype.getName = function() {  
    return this.name;  
}  
var Cat = function(name, age) {  
    Animal.call(this, name, age);  
    this.age = age;  
}  
Cat.prototype =  
    Object.create(Animal.prototype);  
Cat.prototype.constructor = Cat;  
var cat = new Cat("Filemon", 4);  
console.log(cat);
```

`Cat {name: "Filemon", age: 4}`

# Obiektowość w ES6 – podsumowanie

Następnie rozszerzamy nasz obiekt o metodę **getName**.

```
var Animal = function(name) {  
  this.name = name;  
}  
Animal.prototype.getName = function() {  
  return this.name;  
}  
var Cat = function(name, age) {  
  Animal.call(this, name, age);  
  this.age = age;  
}  
Cat.prototype =  
Object.create(Animal.prototype);  
Cat.prototype.constructor = Cat;  
var cat = new Cat("Filemon", 4);  
console.log(cat);
```

**Cat {name: "Filemon", age: 4}**

# Obiektość w ES6 – podsumowanie

Następnie rozszerzamy nasz obiekt o metodę **getName**.

Korzystamy z prototypów.

```
var Animal = function(name) {  
  this.name = name;  
}  
Animal.prototype.getName = function() {  
  return this.name;  
}  
var Cat = function(name, age) {  
  Animal.call(this, name, age);  
  this.age = age;  
}  
Cat.prototype =  
Object.create(Animal.prototype);  
Cat.prototype.constructor = Cat;  
var cat = new Cat("Filemon", 4);  
console.log(cat);
```

**Cat {name: "Filemon", age: 4}**

# Obiektość w ES6 – podsumowanie

Następnie rozszerzamy nasz obiekt o metodę **getName**.

Korzystamy z prototypów.

W ES6 po prostu definiowaliśmy nową metodę w ciele klasy.

```
var Animal = function(name) {  
  this.name = name;  
}  
Animal.prototype.getName = function() {  
  return this.name;  
}  
var Cat = function(name, age) {  
  Animal.call(this, name, age);  
  this.age = age;  
}  
Cat.prototype =  
Object.create(Animal.prototype);  
Cat.prototype.constructor = Cat;  
var cat = new Cat("Filemon", 4);  
console.log(cat);
```

**Cat {name: "Filemon", age: 4}**

# Obiektowość w ES5

Pozostał nam jeszcze konstruktor Cat.

```
var Animal = function(name) {  
    this.name = name;  
}  
Animal.prototype.getName = function() {  
    return this.name;  
}  
var Cat = function(name, age) {  
    Animal.call(this, name);  
    this.age = age;  
}  
Cat.prototype =  
    Object.create(Animal.prototype);  
Cat.prototype.constructor = Cat;  
var cat = new Cat("Filemon", 4);  
console.log(cat);
```

**Cat {name: "Filemon", age: 4}**



# Obiektowość w ES5

Pozostał nam jeszcze konstruktor **Cat**.

Wewnątrz funkcji odwołujemy się (za pomocą metody **call**) do **Animal** z odpowiednim kontekstem tj. **this**, który wskazuje na instancje obiektu **Cat** oraz parametrem **name** i **age**.

```
var Animal = function(name) {  
    this.name = name;  
}  
Animal.prototype.getName = function() {  
    return this.name;  
}  
var Cat = function(name, age) {  
    Animal.call(this, name);  
    this.age = age;  
}  
Cat.prototype =  
Object.create(Animal.prototype);  
Cat.prototype.constructor = Cat;  
var cat = new Cat("Filemon", 4);  
console.log(cat);
```

**Cat {name: "Filemon", age: 4}**

# Obiektowość w ES5

Pozostał nam jeszcze konstruktor **Cat**.

Wewnątrz funkcji odwołujemy się (za pomocą metody **call**) do **Animal** z odpowiednim kontekstem tj. **this**, który wskazuje na instancje obiektu **Cat** oraz parametrem **name** i **age**.

Wygląda to dość skomplikowanie. Przypomnijmy, że w ES6 wystarczyło nam słowo kluczowe **super**!

```
var Animal = function(name) {  
    this.name = name;  
}  
Animal.prototype.getName = function() {  
    return this.name;  
}  
var Cat = function(name, age) {  
    Animal.call(this, name);  
    this.age = age;  
}  
Cat.prototype =  
    Object.create(Animal.prototype);  
Cat.prototype.constructor = Cat;  
var cat = new Cat("Filemon", 4);  
console.log(cat);
```

**Cat** {name: "Filemon", age: 4}

# Obiektowość w ES5

Nakoniec musimy zdefiniować zależność między **Cat** a **Animal**, którą możemy nazwać dziedziczeniem.

```
var Animal = function(name) {  
    this.name = name;  
}  
Animal.prototype.getName = function() {  
    return this.name;  
}  
var Cat = function(name, age) {  
    Animal.call(this, name, age);  
    this.age = age;  
}  
Cat.prototype =  
    Object.create(Animal.prototype);  
Cat.prototype.constructor = Cat;  
var cat = new Cat("Filemon", 4);  
console.log(cat);
```

**Cat {name: "Filemon", age: 4}**

# Obiektowość w ES5

Nakoniec musimy zdefiniować zależność między **Cat** a **Animal**, którą możemy nazwać dziedziczeniem.

Ustawiamy prototyp dla **Cat** jako nowy obiekt **Animal**. Dzięki temu rozwiązaniu właściwości i metody z **Animal** będą dostępne dla **Cat**.

```
var Animal = function(name) {  
    this.name = name;  
}  
Animal.prototype.getName = function() {  
    return this.name;  
}  
var Cat = function(name, age) {  
    Animal.call(this, name, age);  
    this.age = age;  
}  
Cat.prototype =  
Object.create(Animal.prototype);  
Cat.prototype.constructor = Cat;  
var cat = new Cat("Filemon", 4);  
console.log(cat);
```

**Cat** {name: "Filemon", age: 4}

# Obiektowość w ES5

Ustawiamy jeszcze właściwość **constructor** na **Cat**, aby odpowiedni konstruktor został uruchomiony przy tworzeniu nowej instancji obiektu za pomocą słowa kluczowego **new**.

W ES6 to wszystko jest robione dzięki **extends**!

```
var Animal = function(name) {  
    this.name = name;  
}  
Animal.prototype.getName = function() {  
    return this.name;  
}  
var Cat = function(name, age) {  
    Animal.call(this, name, age);  
    this.age = age;  
}  
Cat.prototype =  
Object.create(Animal.prototype);  
Cat.prototype.constructor = Cat;  
var cat = new Cat("Filemon", 4);  
console.log(cat);
```

**Cat {name: "Filemon", age: 4}**

# Obiektowość w ES5

Ustawiamy jeszcze właściwość **constructor** na **Cat**, aby odpowiedni konstruktor został uruchomiony przy tworzeniu nowej instancji obiektu za pomocą słowa kluczowego **new**.

W ES6 to wszystko jest robione dzięki **extends**!

Ustawienie konstruktora.

```
var Animal = function(name) {  
    this.name = name;  
}  
Animal.prototype.getName = function() {  
    return this.name;  
}  
var Cat = function(name, age) {  
    Animal.call(this, name, age);  
    this.age = age;  
}  
Cat.prototype =  
Object.create(Animal.prototype);  
Cat.prototype.constructor = Cat;  
var cat = new Cat("Filemon", 4);  
console.log(cat);
```

**Cat {name: "Filemon", age: 4}**

# Obiektość – ES5 vs ES6

```
class Animal {
  constructor(name) {
    this.name = name;
  }
  getName() {
    return this.name;
  }
}
class Cat extends Animal {
  constructor(name, age) {
    super(name);
    this.age = age;
  }
}
let cat = new Cat("Filemon", 4);
console.log(cat);
```

Cat {name: "Filemon", age: 4}

```
var Animal = function(name) {
  this.name = name;
}
Animal.prototype.getName = function() {
  return this.name;
}
var Cat = function(name, age) {
  Animal.call(this, name, age);
  this.age = age;
}
Cat.prototype =
Object.create(Animal.prototype);
Cat.prototype.constructor = Cat;
var cat = new Cat("Filemon", 4);
console.log(cat);
```

Cat {name: "Filemon", age: 4}



Czas na zadania



# Operator reszty / rozproszenia

# Operator rozproszenia (spread)

Standard ES6 sprawił, że pojawił się nowy operator `...` (trzy kropki), który nosi nazwę reszty lub rozproszenia – w zależności od przeznaczenia.

## Operator rozproszenia

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

Obiekt iterowany implementuje interfejs iterator. Ten temat wykracza poza zakres tej prezentacji.

# Operator rozproszenia (spread)

Standard ES6 sprawił, że pojawił się nowy operator `...` (trzy kropki), który nosi nazwę reszty lub rozproszenia – w zależności od przeznaczenia.

W przypadku gdy znajduje się on przed dowolnym obiektem iterowanym (np. tablicą), to powoduje rozdzielenie każdego elementu tablicy na osobny byt.

## Operator rozproszenia

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

Obiekt iterowany implementuje interfejs iterator. Ten temat wykracza poza zakres tej prezentacji.

# Operator rozproszenia (spread)

Standard ES6 sprawił, że pojawił się nowy operator `...` (trzy kropki), który nosi nazwę reszty lub rozproszenia – w zależności od przeznaczenia.

W przypadku gdy znajduje się on przed dowolnym obiektem iterowanym (np. tablicą), to powoduje rozdzielenie każdego elementu tablicy na osobny byt.

Można to interpretować jak przekazanie każdego elementu z osobna.

## Operator rozproszenia

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

Obiekt iterowany implementuje interfejs iterator. Ten temat wykracza poza zakres tej prezentacji.

# Operator rozproszenia (spread)

To rozwiązanie bardzo ułatwia nam przekazanie wszystkich elementów tablicy jako kolejne parametry funkcji.

Takie wywołanie funkcji jest tożsame z przekazaniem każdego elementu tablicy jako kolejnego parametru.

## Operator rozproszenia

```
function sum(a, b, c) {  
    return a + b + c;  
}  
var res1 = sum(...[1, 2, 3]);  
console.log(res1); // 6  
var arr = [1, 2, 3];  
var res2 = sum(arr[0], arr[1], arr[2]);  
console.log(res2); // 6
```

# apply() zamiast rozproszenia

Aby wykonać to samo bez operatora rozproszenia musielibyśmy wykorzystać metodę **apply()**, która przyjmuje dwa parametry.

- Obiekt, na którym ma być wywołana funkcja, lub **null**, jeśli ma być to obiekt globalny.
- Lista argumentów jako tablica, z którymi ma być ta funkcja wywołana.

## Operator rozproszenia

```
function sum(a, b, c) {  
    return a + b + c;  
}  
var res1 = sum(...[1, 2, 3]);  
console.log(res1); // 6  
var arr = [1, 2, 3];  
var res2 = sum.apply(null, arr);  
console.log(res2); // 6
```

# Operator rozproszenia (spread)

Możemy również użyć tego operatora w innym kontekście np. do podzielenia stringa na poszczególne znaki i zapisanie wyniku tej operacji do tablicy.

## Operator rozproszenia

```
var str = "text";  
var chars = [ ...str ];  
console.log(chars);  
// ["t", "e", "x", "t"]  
console.log(typeof str);  
// string  
console.log(typeof chars);  
// object
```

# Operator rozproszenia (spread)

Możemy również użyć tego operatora w innym kontekście np. do podzielenia stringa na poszczególne znaki i zapisanie wyniku tej operacji do tablicy.

Czyli najpierw rozpraszamy na poszczególne znaki tekst w zmiennej **str** a potem osadzamy je w tablicy.

## Operator rozproszenia

```
var str = "text";  
var chars = [ ...str ];  
console.log(chars);  
// ["t", "e", "x", "t"]  
console.log(typeof str);  
// string  
console.log(typeof chars);  
// object
```



# Operator rozproszenia (spread)

Możemy również użyć tego operatora w innym kontekście np. do podzielenia stringa na poszczególne znaki i zapisanie wyniku tej operacji do tablicy.

Czyli najpierw rozpraszamy na poszczególne znaki tekst w zmiennej **str** a potem osadzamy je w tablicy.

Pamiętajmy, że tablica to obiekt!

## Operator rozproszenia

```
var str = "text";  
var chars = [ ...str ];  
console.log(chars);  
// ["t", "e", "x", "t"]  
console.log(typeof str);  
// string  
console.log(typeof chars);  
// object
```

# Operator rozproszenia (spread)

W podobny sposób moglibyśmy osadzić jedną tablicę w drugiej.

## Operator rozproszenia

```
var x = [3, 4];  
var y = [1, 2, ...x, 7];  
console.log(y); // [1, 2, 3, 4, 7]
```

# Operator rozproszenia (spread)

W podobny sposób moglibyśmy osadzić jedną tablicę w drugiej.

Wystarczy, że w miejsce, w które ma być wstawiona tablica, wpiszemy nazwę zmiennej, poprzedzając ją operatorem rozproszenia.

## Operator rozproszenia

```
var x = [3, 4];  
var y = [1, 2, ...x, 7];  
console.log(y); // [1, 2, 3, 4, 7]
```

# Operator reszty (rest)

Do tej pory wykorzystywaliśmy operator trzech kropek do rozpraszania, teraz spróbujmy „zebrać” resztę.

W poniższym przykładzie mamy funkcję **fn**, która przyjmuje dowolną liczbę parametrów, gdzie dwa pierwsze to **a** i **b** a reszta parametrów jest przechowywana w **args**.

Zmienna **args** to tablica, której zawartość składa się z kolejnych argumentów (rozpoczynając od trzeciego) przekazanych do funkcji **fn**.

## Operator reszty

```
function fn(a, b, ...args) {  
    return (a + b) * args.length;  
}  
console.log(fn(2, 4, "IT", true, 9))  
// 18
```

# Operator reszty (rest)

Nazwa parametru, który wykorzystuje operator reszty jest dowolna.

W obu wywołaniach funkcji **foo** oraz **bar**, pierwszy parametr to tablica zawierająca wartości, jakie zostały przekazane jako argumenty funkcji.

## Operator reszty

```
function foo(...params) {  
    console.log(params);  
}  
function bar(...par) {  
    console.log(par);  
}  
foo(1, 2, 3, true);  
// [1, 2, 3, true];  
bar([1], {x: 1, y: 2});  
// [ [1], {x: 1, y: 2} ];
```

# Operator reszty (rest)

Jedynym ograniczeniem podczas korzystania z operatora reszty jest pozycja parametru – musi to być ostatni parametr w funkcji. Inaczej pojawi się nam błąd.

## Operator reszty

```
function foo(...arr, par) { // błąd
  console.log(arr, par);
}
function bar(...x, y, ...z) { // błąd
  console.log(x, y, z);
}
```

# Domyślne wartości i operator reszty

Niestety operator reszty nie przyjmuje domyślnych wartości, które omawialiśmy wcześniej.


Źle

```
function fn(a = 10, ...args=[1, 2, 3]) {  
  console.log(a * args.length );  
  // Błąd!  
}
```



Czas na zadania





# **Łańcuchy szablonów i interpolacja stringów**

# Łańcuchy szablonowe – template strings

W ES6 mamy do dyspozycji tzw. łańcuchy szablonowe (ang. Template strings). Wyglądają jak zwykłe stringi, z tym że nie umieszczamy ich wewnątrz cudzysłowu ani apostrofów. Stosujemy apostrof odwrotny (ang. backtick): `.

## Przykład

```
let text = `Przez te oczy zielone...`;
```

W przykładzie obok `${user}` oraz `${a + 2}` to tzw. zamienniki szablonowe.

## Zalety łańcuchów

Mamy możliwość interpolacji stringów, czyli w przyjemny sposób możemy za ich pośrednictwem wstawiać kod JavaScript do stringa.

## Przykład

```
let user = "Agata";  
console.log(`Witaj ${user}`);
```

Witaj Agata

```
let a = 23;  
console.log(`Suma ${a + 2}`);
```

Suma 25

# Łańcuchy szablone – ważne

Kilka ważnych kwestii związanych z łańcuchami szablonowymi:

- Kod wewnątrz zamiennika `${ten tutaj}` może być dowolnym wyrażeniem JavaScriptu - dodawaniem, mnożeniem, funkcją, pętlą itp.
- Łańcuchy mogą zawierać więcej niż jedną linię kodu. Zobacz obok przykład.
- Nie rozwiązują problemów ze znakami specjalnymi – jest to związane z bezpieczeństwem.

```
console.log(`pierwsza linia tekstu  
druga linia tekstu,  
trzecia linia i zaraz czwarta,  
wynik dodawania to ${2+2}`);
```

**pierwsza linia tekstu**  
**druga linia tekstu**  
**trzecia linia i zaraz czwarta,**  
**wynik dodawania to 4**



Czas na zadania

# Destrukturyzacja



# **Destrukturyzacja tablic**

# Przykład

Destrukturyzacja w ES6 polega na przypisaniu własności tablicy lub obiektu do zmiennych.

## Przykład

```
let fruits = ["Banana", "Apple"];  
let [value1, value2] = fruits;  
console.log(value1)  
console.log(value2)
```

"Banana"

"Apple"

Bez destrukturyzacji ten sam kod napisalibyśmy w następujący sposób:

## Przykład

```
let fruits = ["Banana", "Apple"];  
let value1 = fruits[0];  
let value2 = fruits[1];  
console.log(value1)  
console.log(value2)
```

"Banana"

"Apple"

# Destrukturyzacja – składnia

Przypisujemy do **zmiennej1** wartość pierwszego elementu tablicy, potem do kolejnej zmiennej kolejną wartość, aż do końca.

```
[ zmienna1, zmienna2, zmiennaN ] = nazwaTablicy;
```



# Destrukturyzacja – więcej przykładów

## Funkcja

Funkcja może zwracać tablicę.

```
var foo = () => [1, 2, 3];  
var [a, b] = foo();  
console.log(a, b); //1 2
```

Zauważ, że podaliśmy dwie zmienne a mamy trzy wartości w tablicy, co oznacza, że trzecia wartość nie zostanie uwzględniona.

## Omijanie niektórych wartości

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

Zostawienie wolnego miejsca sugeruje, że nie będziemy przypisywać tej z kolei wartości.

# Destrukturyzacja – więcej przykładów

## Spread / rest operator

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

Pierwszy element tablicy przypisujemy do zmiennej **a**, natomiast resztę (w postaci tablicy) do zmiennej **b**. W związku z czym:

**a** = 1

oraz

**b** = [2,3]

## Zamiana wartości (bez dodatkowej zmiennej)

```
var a = 1, b = 2;  
[b, a] = [a, b];  
console.log(a, b); // 2 1
```

Jeśli chcielibyśmy zamienić wartościami dwie zmienne, potrzebowalibyśmy trzeciej – dodatkowej. Spójrz na przykład:

```
var a = 1, b = 2, c = a;  
a = b;  
b = c;  
console.log(a, b); // 2 1
```



# **Destrukturyzacja obiektów**

# Destrukturyzacja – przykład

Destrukturyzacja obiektów pozwala nam powiązać zmienne z własnościami danego obiektu. Spójrz na przykład:

```
let point = {sum: 42, isOk: true};  
let {sum, isOk} = point;  
console.log(sum); // 42  
console.log(isOk); // true
```

**Nazwa zmiennej, pod którą podstawiamy, oraz nazwa atrybutu obiektu muszą być takie same.**

Możemy zrobić destrukturyzację obiektu bez jego deklaracji.

```
let {sum, isOk} = {sum: 42, isOk: true};  
console.log(sum); // 42  
console.log(isOk); // true
```

# UWAGA! Destrukturyzacja – więcej przykładów

A gdybyśmy chcieli zmienić nazwę zmiennej, pod którą będzie widoczna dana własność obiektu?

```
let userA = { name: "Alicja" };  
let { name: nameAlice } = userA;  
console.log(nameAlice); // Alicja
```



Czas na zadania

# Export & import

# Notacja obiektów w ES6

Zanim przejdziemy do eksportowania i importowania, poznamy jeszcze bardzo przyjemną notację obiektów, która na pewno Ci się spodoba.

Wyobraź sobie, że masz obiekt **balloon**. Chcesz przypisać do jego pól zmienne **color** i **size**. Standardowo moglibyśmy to zrobić tak:

```
const color = "red";
const size = 200;
const balloon = {
  color: color,
  size: size
};
console.log( balloon.color ); // red
```

## Przypisanie w ES6

Mamy do dyspozycji notację ES6, możemy zatem przypisać zmienne prościej.

```
const color = "red";
const size = 200;
const balloon = { color, size } ;
console.log( balloon.color ); // red
```



# CommonJS

## CommonJS – przypomnienie

Do tej pory wiemy, że istnieje kilka sposobów na ładowanie modułów w JavaScript. Jednym ze standardów, który omówiliśmy, był **CommonJS**.

Standard, który wspiera synchroniczne ładowanie modułów. Jego składnia skupia się na słowach kluczowych **export** oraz **require**.

## Plik calc.js

```
function add(a, b) {  
    return a + b;  
}  
module.exports = add;
```

## Plik app.js

```
var add = require("./calc.js");  
console.log( add(10, 2) );
```

## Wynik w konsoli po uruchomieniu

```
$ node ./app.js  
12
```

# Moduły ES6 – przykład

W ES6 modułem określamy plik zawierający kod JavaScript. Moduły w ES6 działają domyślnie w trybie ścisłym (**"use strict"**), dodatkowo możemy w nich korzystać ze słów kluczowych **import** oraz **export**.

Kiedy korzystaliśmy z dobrodziejstw **CommonJS**, nie wspomagaliśmy się żadną biblioteką, która pozwalała, by nam otwierać nasze pliki w przeglądarce. Tym razem korzystamy z ES6, po kompilacji Webpackiem spokojnie możemy otworzyć nasz plik w przeglądarce.

## Plik calc.js

```
function add(a, b) {  
    return a + b;  
}  
export { add };
```

## Plik app.js

```
import { add } from "./calc";  
console.log( add(10, 2) );
```

## Otwieramy plik index.html

12

# UWAGA! Import i export wielu rzeczy

Przy eksportowaniu i importowaniu warto pamiętać o nawiasach klamrowych. Zwróć uwagę, jak eksportujemy i importujemy wiele rzeczy. Należy rozdzielać je przecinkiem.

Przy importowaniu w nawiasach klamrowych podajemy, co chcemy zaimportować a w kolejnej części wyrażenia skąd (z jakiego modułu). Zauważ, że możemy pominąć rozszerzenie `.js`.

## Moduły w ES6 – przykład

```
function add(a, b) {  
    return a + b;  
}  
const cat = {  
    name: "Puszek"  
}  
export { add, cat } ;
```

## Plik app.js

```
import { add, cat } from "./calc";  
console.log( add(10, 2) );  
console.log( cat.name );
```

## Wynik w konsoli

12

Puszek

# UWAGA! Import i alias

Jeżeli importujesz moduł i chcesz zmienić jego nazwę w pliku, w którym będziesz się nim posługiwać, możesz ustawić mu tzw. alias.

## Moduły w ES6 – przykład

```
function add(a, b) {  
    return a + b;  
}  
const cat = {  
    name: "Puszek"  
}  
export { add, cat } ;
```

## Plik app.js

```
import {add, cat as kitty} from"./calc";  
console.log( add(10, 2) );  
console.log( kitty.name );
```

Czas na zadania