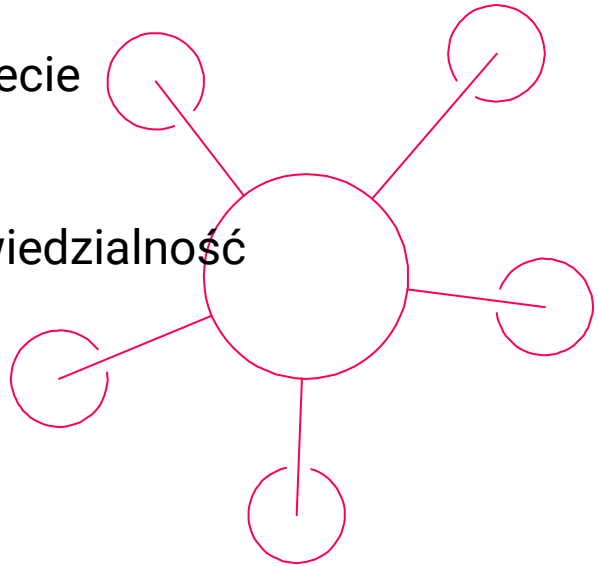
# JavaScript
# OOP & Prototype

# OOP i Prototype
# w JavaScript

- Programowanie obiektowe (ang. object-oriented programming)  - OOP
- Koncepcja
- Terminologia
- Założenia
- OOP w JavaScript

# Programowanie obiektowe

- sposób projektowania oprogramowania poprzez zbiór obiektów i interakcji między nimi
- tworzymy modele oparte na rzeczywistym świecie
- każdy obiekt jest małym światem pełniącym odrębną role i mającym inna odpowiedzialność



# Geneza
# OOP

- 1967 rok, Oslo. Zespół pracował nad symulacjami zachowania się statków w języku Simula 67
- Kłopoty z opanowaniem wszystkich zależności, jakie wywierały na siebie nawzajem wszystkie parametry statków w symulacji
- Wpadli na pomysł, by pogrupować typy statków w różne klasy obiektów, a każda z klas sama odpowiadałaby za określanie własnych danych i zachowań.

# Czym jest **OOP?**

- Paradygmat programowania, w którym programy definiuje się za pomocą obiektów – elementów łączących stan (czyli dane, nazywane najczęściej polami) i zachowanie (czyli metody, funkcje).
- Największym atutem programowania, projektowania oraz analizy obiektowej jest zgodność takiego podejścia z rzeczywistością – mózg ludzki jest w naturalny sposób najlepiej przystosowany do takiego podejścia przy przetwarzaniu informacji

# Czy OOP to jedyny **paradygmat?**

- Programowanie proceduralne
- Programowanie agentowe
- Programowanie funkcyjne

# Podsumowując pojęcie **OOP?**

- Paradygmat (zbiór zasad) programowania
- Każdy obiekt jest jest jak mały świat, który ma swoją rolę i odpowiedzialność
- Podejście jest zgodne z rzeczywistością, a mózg ludzki jest naturalnie przystosowany do przetwarzania takich informacji

# Zalety i wady
# OOP

## Zalety

- przejrzystość i łatwość zrozumienia
- łatwiejsza równoległa praca programistów
- reużywalność kodu
- łatwa rozbudowa i elastyczność

## Wady

- może powstawać nadmiarowy kod
- modyfikacja i rozwój wymaga znajomości OOP

# Terminologia
## OOP

- Klasa (ang. "class") - Definiuje własności obiektu *
- Obiekt (ang. "object") - Instancja (byt, twór) klasy.
- Właściwość (ang. "property") - Własność obiektu, np. kolor.
- Metoda (ang. "method") - Zdolność (czynność) obiektu, np. chodzenie (idź).
- Konstruktor (ang. "constructor") - Metoda wywoływana w momencie inicjalizacji obiektu.

# Paradygmaty
# OOP

- Abstrakcja - Obiekt służy jako model abstrakcyjnego „wykonawcy”, który może wykonywać pracę, opisywać i zmieniać swój stan oraz komunikować się z innymi obiektami w systemie bez ujawniania, w jaki sposób zaimplementowano dane cechy

# Paradygmaty
## OOP

- Hermetyzacja - ukrywanie implementacji, enkapsulacja. Zapewnia, że obiekt nie może zmieniać stanu wewnętrznego innych obiektów w nieoczekiwany sposób. Tylko własne metody obiektu są uprawnione do zmiany jego stanu. Każdy typ obiektu prezentuje innym obiektom swój interfejs, który określa dopuszczalne metody współpracy.

# Paradygmaty
## OOP

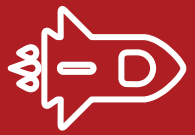
- Dziedziczenie - Porządkuje i wspomaga polimorfizm i enkapsulację dzięki umożliwieniu definiowania i tworzenia specjalizowanych obiektów na podstawie bardziej ogólnych. Dla obiektów specjalizowanych nie trzeba redefiniować całej funkcjonalności, lecz tylko tę, której nie ma obiekt ogólniejszy.

# Paradygmaty
## OOP

- Polimorfizm - Referencje i kolekcje obiektów mogą dotyczyć obiektów różnego typu, a wywołanie metody dla referencji spowoduje zachowanie odpowiednie dla pełnego typu obiektu wywoływanego. Często jest to pojęcie powiązane z dziedziczeniem. Oznacza to po prostu "wiele kształtów".

# Sposoby tworzenia obiektów w JavaScript

# Object Literal



# Object Literal
## przykład

```
const infoShareAcademy = {
    year: 2017,
    name: 'infoShare Academy',
    city: 'Gdańsk',
    sayHello: function (name, surname) {
        alert(`Hello ${name} ${surname}`);
    },
    courses: [
        {
            active: true,
            name: 'JUNIOR FRONT-END DEVELOPER',
            beginAt: new Date(2017, 2, 18, 15, 0, 0, 0)
        },
        {
            active: false,
            name: 'JUNIOR FRONT-END DEVELOPER',
            beginAt: new Date(2017, 5, 5, 9, 0, 0, 0)
        }
    ]
};
```

# Object Literal

## przykład mapowania realnych obiektów

```
const chair = {
    color: 'brown',
    x: 100,
    y: 200,
    angle: 30,
    move: function(x, y) {}
};
```

```
const car = {
    x: 100,
    y: 200,
    color: 'silver',
    doors: 4,
    tires: 4,
    maxSpeed: 200,
    drive: function(x,y) {},
    stop: function() {},
};
```

```
const hen = {
    x: 2,
    y: 1,
    color: 'white',
    eags: 5,
    eat: function() {},
    sleap: function() {},
    layEgg: function() {}
};
```

# Context funkcji jest nadawany podczas jej wywoływania, a nie tworzenia

**car**.calculatePrice.call({ price: 200 }, arg1, arg2, arg3, …);

**car**.calculatePrice.apply({ price: 200 }, [arg1, arg2, arg3, …]);

# Object Literal
## zalety i wady

### zalety
- prostota
- wykorzystywane do przesyłania danych miedzy obiektami

### wady
- brak możliwości dodania/usunięcia/podmiany metod czy własności wszystkich obiektom w łatwy sposób
- publiczne własności samych obiektów

# Constructor Function

# Constructor Function
## przykład

```
const Book = function Book(name, price) {}

function Book() {
    this.name = name;
    this.available = true;
    this.price = price;

    this.borrow = function(person) {}
}

const book1 = new Book('Hobbit', 120);
// book1.name = 'Hobbit';
// book1.price = 10;

const book2 = new Book('Treasure Island', 20);
// book2.price = 20;
// book2.name = 'Treasure Island';
```

# konwencja nazewnicza - PascalCase

Operator **new** tworzy obiekt i nadaje mu context

# Constructor Function
## zalety i wady

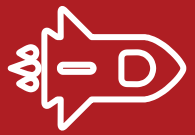
## zalety

- enkapsulacja (metody, własności)
- funkcje pomocnicze
- tworzenie wielu instancji z różnymi parametrami
- możliwość sprawdzenia Typu obiektu (instanceOf)

## wady

- brak możliwości dodania/usunięcia/podmiany metod czy własności wszystkich obiektom w łatwy sposób

# Object Literal



Object.create()

# Object Literal

## Object.create()

```
const baseBook = {
    price: 12.99,
    available: true,
    borrow: function() {
        this.available = false;
    }
};

const book1 = Object.create(baseBook);
book1.name = 'Hobbit';
book1.author = 'Tolkien';
book1.price = 120;

const book2 = Object.create(baseBook);
book2.name = 'Treasure Island';
book2.returnBack = function() {
    this.available = true;
};

book1.borrow();
book2.returnBack();
```

```
> book1.available
< false
> book1.price
< 120
> book2.available
< true
> book2.price
< 12.99
> book1.returnBack();
✖ ▶ Uncaught TypeError: book1.returnBack is not a function
      at <anonymous>:1:7
> book2.returnBack();
< undefined
```

# Object Literal
## Object.create()

```
const baseBook = {
    price: 12.99,
    available: true,
    borrow: function() {
        this.available = false;
    }
};

const book1 = Object.create(baseBook);
book1.name = 'Hobbit';
book1.author = 'Tolkien';
book1.price = 120;

const book2 = Object.create(baseBook);
book2.name = 'Treasure Island';
book2.returnBack = function() {
    this.available = true;
};

book1.borrow();
book2.returnBack();
```

```
> book1
< ▼ Object i
    author: "Tolkien"
    available: false
    name: "Hobbit"
    price: 120
  ▼ __proto__: Object
    available: true
  ▶ borrow: ()
    price: 12.99
  ▼ __proto__: Object
    ▶ __defineGetter__: __defineGetter__()
    ▶ __defineSetter__: __defineSetter__()
    ▶ __lookupGetter__: __lookupGetter__()
    ▶ __lookupSetter__: __lookupSetter__()
    ▶ constructor: Object()
    ▶ hasOwnProperty: hasOwnProperty()
    ▶ isPrototypeOf: isPrototypeOf()
    ▶ propertyIsEnumerable: propertyIsEnumerable()
    ▶ toLocaleString: toLocaleString()
    ▶ toString: toString()
    ▶ valueOf: valueOf()
    ▶ get __proto__: __proto__()
    ▶ set __proto__: __proto__()
```

# Object Literal
## Object.create()

```
const baseBook = {
    price: 12.99,
    available: true,
    borrow: function() {
        this.available = false;
    }
};

const book1 = Object.create(baseBook);
book1.name = 'Hobbit';
book1.author = 'Tolkien';
book1.price = 120;

const book2 = Object.create(baseBook);
book2.name = 'Treasure Island';
book2.returnBack = function() {
    this.available = true;
};

book1.borrow();
book2.returnBack();
```

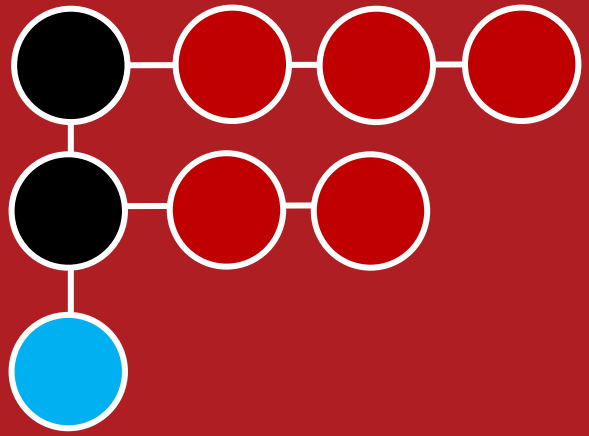
```
> book2
< ▼ Object i
    available: true
    name: "Treasure Island"
  ▶ returnBack: ()
  ▼ __proto__: Object
    available: true
  ▶ borrow: ()
    price: 12.99
  ▼ __proto__: Object
    ▶ __defineGetter__: __defineGetter__()
    ▶ __defineSetter__: __defineSetter__()
    ▶ __lookupGetter__: __lookupGetter__()
    ▶ __lookupSetter__: __lookupSetter__()
    ▶ constructor: Object()
    ▶ hasOwnProperty: hasOwnProperty()
    ▶ isPrototypeOf: isPrototypeOf()
    ▶ propertyIsEnumerable: propertyIsEnumerable()
    ▶ toLocaleString: toLocaleString()
    ▶ toString: toString()
    ▶ valueOf: valueOf()
    ▶ get __proto__: __proto__()
    ▶ set __proto__: __proto__()
```

# Object Oriented Programming

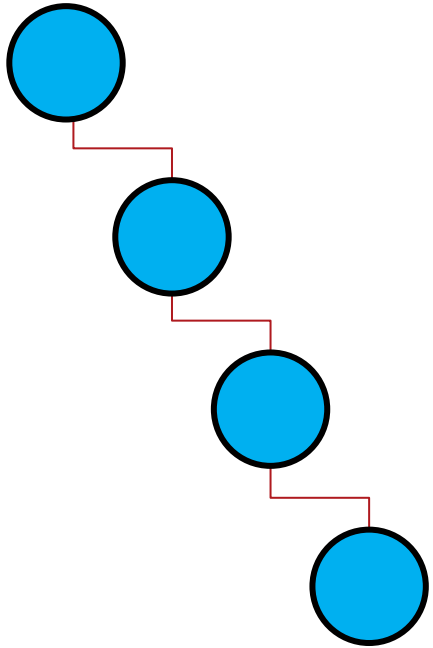
**class** Definicja obiektu (własności, metody)

**Interface** Sposób komunikacji przez metody

**Instance** obiekt stanowy



# Prototype-based programming



# Object Literal

## Object.getPrototypeOf(obj)

```
const baseBook = {
    price: 12.99,
    available: true,
    borrow: function() {
        this.available = false;
    }
};

const book2 = Object.create(baseBook);
book2.name = 'Treasure Island';

baseBook.price = 33.33;

Object.getPrototypeOf(book2).price = 44.44;
Object.getPrototypeOf(book2).borrow = function() {
    throw new Error("Method not supported");
};
delete Object.getPrototypeOf(book2).borrow;
book2.borrow();
```

# Object Literal

```
const baseBook = {
    price: 12.99,
    available: true,
    borrow: function() {
        this.available = false;
    }
};

const book2 = Object.create(baseBook);
book2.name = 'Treasure Island';
book2.cover = 'red';
```

```
> Object.keys(book2);
< ▶ ["name", "cover"]
> Object.keys(baseBook);
< ▶ ["price", "available", "borrow"]
> for(let key in book2) {
    console.log(key, book2.hasOwnProperty(key));
  }
  name true
  cover true
  price false
  available false
  borrow false
```

# Constructor Function

prototype

# Constructor Function
## prototype

```
function Person(name, surname) {
    this.health = 50;
    this.name = name;
    this.surname = surname;
}

Person.prototype.sayHello = function() {
    console.info('Greetings from ' + this.name);
};

function Doctor(name, surname) {
    Person.call(this, name, surname);
    this.hospital = 'Main';
}

Doctor.prototype = Object.create(Person.prototype);
Doctor.prototype.constructor = Doctor;

Doctor.prototype.cure = function(person) {
    if (person instanceof Person === false) {
        throw new Error("Can't cure!");
    }
    person.health += 10;
};
```

# Constructor Function

## prototype

## zasady

- W Construtor Function przypisujemy tylko własności (nigdy metody, ponieważ złamiemy `prototype chain` i nie będziemy mogli podmienić metody dla wszystkich obiektów)
- Metody definiujemy zawsze w prototypie CF
- W Construtor Function wywołujemy inną CF z własnym contextem w celu skopiowania własności

# Zła praktyka!

## Dodawanie metod do prototypów obiektów natywnych (Array, Object, String etc.)

```
if (!Array.prototype.forEach) {
    Array.prototype.forEach = function(callback) { /* ... */ }
}
```

# Single Responsibility Principle

Do one thing and do it well!

# Open-Closed Principle

Your JavaScript objects should be
open to extension, but closed to modification

# Open-Closed Principle

```
function Product() {
    this.colors = ["red", "black"];
    this.printColors = function() {
        console.log(this.colors);
    }
}
```

```
function Product() {
    const colors = ["red", "black"];

    this.printColors = function() {
        console.log(this.colors);
    };

    this.addColor = function(color) {
        colors.push(color);
    }
}
```