Object Oriented Programming

Some of examples and definitions are from very good JS docs - https://developer.mozilla.org/ .

1.
Conception of OOP

Conception of OOP
Definition

Object-oriented programming (OOP) is a paradigm in programming where we are using objects that have their own properties such as:

- fields (data, information about the site)
- methods (activities / functions that the object performs)

In object-oriented programming we are defining objects and calling their methods so that they interact with each other.

Conception of OOP
Abstraction

The concept of abstraction in object-oriented programming is most often identified with classes (there are no classes in ES5).

Abstraction (a class) is a model that does not actually represent an existing object. A class is only the basis where instructions how to create objects are defined.

An example is a mammal or an animal - this is no real world object. The real world object based on their properties are eg: Lion or Cat.

Conception of OOP
Inheritance

A programming technique that allows to use of an existing class to create a new one, based on its parent.

In JS, where is no concept of class itself, **objects inherits from others objects, and this is called prototypal inheritance**.

Conception of OOP
Encapsulation / Hermetization

Encapsulation ensures that the object can not change the internal state of other objects in unexpected way.

Only the object's own methods are authorized to change own state.

Each type of object presents its interface to other objects, which it defines acceptable methods of cooperation (setters and getters).

Conception of OOP
Polimorfism

When writing a program, it is convenient to treat even different data in an uniform way way.

Regardless if you should add a number or an array, is usually more readable when the operation is called simply **add**, not **addNumber** nor **addString**.

However, the string must be added differently than the array (it is concatenation rather than adding), so there will be two add implementations, but naming them with a common name creates a convenient one abstract interface independent of the printed value type.

Conception of OOP
Pros and cons

Advantages:
+ transparency, easy to understand
+ easy code modification
+ possibility of working by many programmers on the same code
+ easy expansion, flexibility
+ possibility re-use of once written code

Disadvantages:
- a lot of redundant code
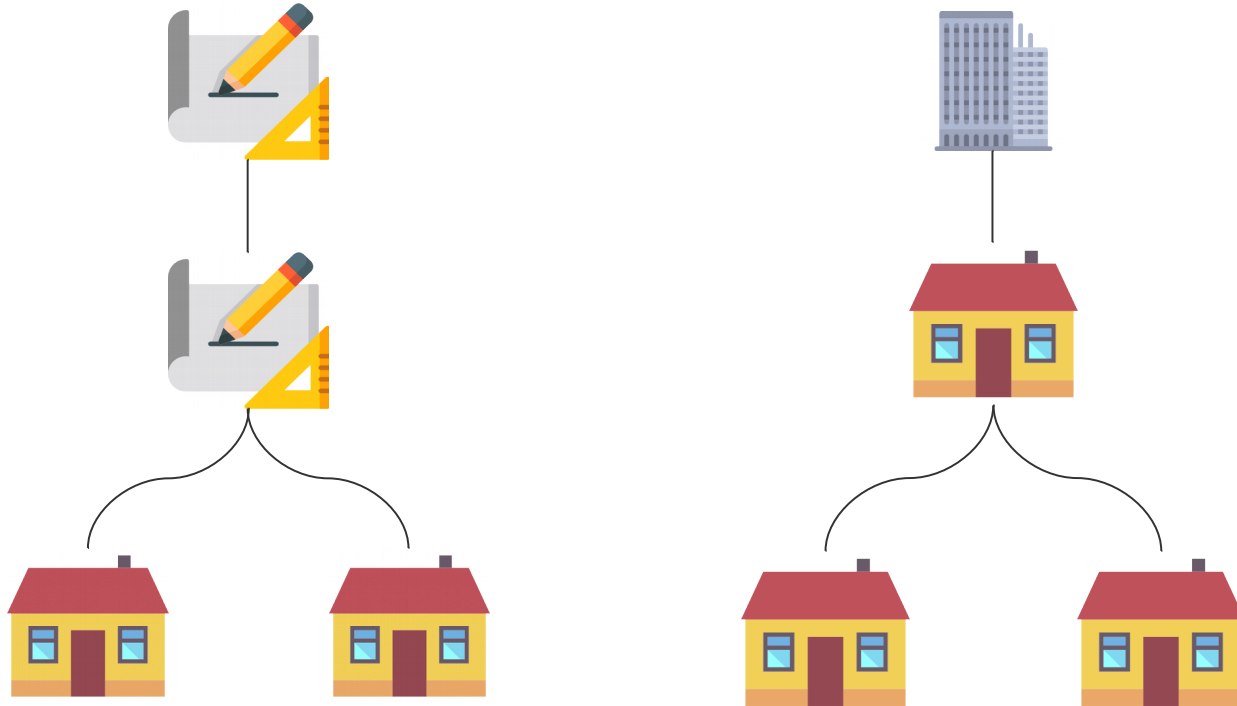- a more abstract approach where more thinking is needed
- in some cases it is unnecessary

Conception of OOP
Class based inheritance vs prototypal inheritance

Conception of OOP
Class based inheritance vs prototypal inheritance

Icons are from https://www.flaticon.com/

2.
Function execution context

Function execution context
this keyword

In most cases, the value of **this** is determined by how a function is called and points on function execution context. If we write a normal function **this** will behave in that way (ES6 arrow functions behave different!).

In the global execution context, **this** refers to the global object - in browsers - **window**.

this can't be set by assignment during execution!

this may be different each time the function is called!

We can set the execution context using **.bind()**, .**call()**, **apply()** methods.

Function execution context
this keyword *

```
this.name = "Ala"

function hello() {
    console.log(this.name + " mówi cześć!");
}

var person = {name: "Zenek", hello: hello}

hello() // Ala mówi cześć!

person.hello() // Zenek mówi cześć!
```

Task 0

 # 1 Try to console.log this in a global scope and in function scope (write and call function for it).

#2 Attach THAT function to object. Call it as a method of that object.

Function execution context
.bind() *

Creates a copy of a function that has the same body and parameters as the original function, but the execution context is bind to that function.

```
function returnX () {
    return this.x;
}

var obj = { x: 42 }

var newReturnX = returnX;

newReturnX() // execution in global scope, undefined is returned

var boundReturnX = newReturnX.bind(obj)

boundReturnX() // execution in obj object context , 42
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/bind

Function execution context
.call()

Calls a function with context provided in call argument, and argument listed next, separated by colon.

```
function returnX () {
    return this.x;
}

var obj = { x: 42 }

returnX.call(this, 1, 2, 3) // undefined
returnX.call(obj, 1, 2, 3) // 42

returnX.call({}, 1, 2, 3) // undefined
returnX.call({ x: 142 }, 1, 2, 3) // 142
```

Function execution context
.apply()

Similar to **.call().** Calls a function with context provided in apply argument, and argument listed next in an array.

```
function returnX () {
    return this.x;
}

var obj = { x: 42 }

returnX.call(this, [1, 2, 3]) // undefined
returnX.call(obj, [1, 2, 3]) // 42

returnX.call({}, [1, 2, 3]) // undefined
returnX.call({ x: 142 }, [1, 2, 3]) // 142
```

Task 2

"

Write function that logs all of function arguments in first console.log and this in second.

Call it using apply and call, passing some object as context and some arguments.

3.
Object creation

Object creation
Object literal *

The simplest way to create object in JS is object literal.

```
var cat = {
    name: 'Fluffy',
    age: 1,
    sound: 'Meeeeow!',
    makeSound: function() {
        console.log(this.sound);
    },
    speak: function() {
        console.log('Sorry cats can't speak');
    }
};
```

Object creation
Factory functions *

Factory function is a function that creates something using arguments that are passed to function. Usually it creates an object:

```
function catFactory(name) {
   return {
      name: name,
      sound: 'Meeeeow!',
      speak: function(){
         console.log('Sorry cats can't speak');
      }
   }
}
```

Task 3

"

Make a factory function that produces a cat object with name property that will passed through function parameter.

Make several cats and assign them to some variables.

Task 4

Extend function from task 3.

Now we want to pass also sound property, of cat object, through the parameter and we want to have "makeSound" method that logs this.sound

Object creation
Object.assign()

The **Object.assign()** method is used to copy the values of all own properties from one or more source objects to a target object. It will return the target object.

```
const obj1 = { a: 1, b: 2, c: 3 }

const obj2 = Object.assign({c: 4, d: 5}, obj1)

console.log(obj2) // { a: 1, b: 2, c: 4, d: 5}
```

Task 5

"

Make 3 objects with some random properties.

Combine them together by Object.assign, creating a NEW object.

4.
Inheritance

Inheritance
Object.create()

The **Object.create()** method creates a new object, using an existing object to provide the newly created object's prototype.

```
var base = {
  baseProperty: function(){
    console.log('I'm from prototype!');
  }
}

var obj = Object.create(base);

console.log(obj) // WTF? it's empty object? -> {}

obj.baseProperty() // I'm from prototype!
```

Inheritance
What is prototype in JS?

When it comes to inheritance, JavaScript only has one construct: objects. Each object has a private property which holds a link to another object called its prototype. That prototype object has a prototype of its own, and so on until an object is reached with null as its prototype. By definition, null has no prototype, and acts as the final link in this prototype chain.

Nearly all objects in JavaScript are instances of Object which sits on the top of a prototype chain.

Task 6

#1 Make a "cat" object, that has a function that logs this.sound.

#2 Make a new object by Object.create with "cat". Call function that logs this.sound.

Inheritance
Constructor functions *

```
function Cat(name, age){
    this.name = name
    this.age = age
    this.sound = "Meeeeow!",
    this.makeSound = function(){
        console.log(this.sound)
    }
    this.speak = function(){
        console.log('Sorry cats can't speak')
    }
}
```

Inheritance
new keyword *

Constructor function is a function that is written to be invoked with **new** keyword.

New keyword makes 4 things with function:
1. Makes new empty object
2. Sets a prototype (**__proto__**) of this object to object in constructor function **prototype property**
3. Cals that function in context of new object created in step 1.
4. Return object created in step 1.

Inheritance

Constructor functions *

```
function Cat(name, age){
    this.name = name
    this.age = age
    this.sound = 'Meeeeow!'
}

Cat.prototype.makeSound = function(){
    console.log(this.sound);
}
Cat.prototype.speak = function(){
    console.log('Sorry cats can't speak');
}
```

" Task 7

Make a constructor function that makes cats object, with name, sound properties, and speak, makeSound methods, from scratch.

infoShare

Inheritance
Constructor functions with IIFE *

```javascript
var Cat = (function() {
  function Cat(name) {
    this.sound = "Meow";
  }

  Cat.prototype.makeSound = function makeSound() {
    console.log(this.sound);
  };

  return Cat;
})();
```

Inheritance
Other conception - functional programming

Functional programming is a programming paradigm where programmers tend to write small and simple functions that avoids changing-state and mutable data.

In functional style we use function composition to achieve benefits similar to inheritance.

Main profit of functional programming is simplicity of code.

In JS we can easily write both object and functional code.

https://www.youtube.com/watch?v=wfMtDGfHWpA

Inheritance vs composition?

Inheritance
Inheritance in practice

Using inheritance -> https://codesandbox.io/s/4rnxxz9zyw

Doing the same thing without inheritance ;) ->
https://codesandbox.io/s/0pv1xyv72p

Functional approach -> https://codesandbox.io/s/o5z6p7y8k5

5.
Extras