Back to Week 1

JS | OOP - class and inheritance

LESSON

Learning Goals

After this lesson you will be able to:

- Create `class` and know how to use it
- Understand what the `constructor` and `new` do
- Understand the concepts of
    - **inheritance**,
    - **abstraction**,
    - **polymorphism** and
    - **encapsulation**

Introduction

In this lesson, we will continue our previous example, the Monopoly game, but we will introduce a new syntax with the **`class`** keyword.

100	-10	0	0	-40
-10				-10
-50				-10
0				5
0	-10	-50	-10	0

	Player	Position	Cash
	Joaquim	0	1,000€
	Maxence	0	1,000€
	Mostafa	0	1,000€

First example of a `class` in JavaScript

Let's see: we learned the way not to repeat the same code as much as we used at the beginning (the first version of our *Monopoly* game) but still there's a quite a lot repetitive code and we see a pattern here: all our objects have the same properties (keys) with different values.

❓ The question now is: *is there a way to create just one object and be able to reuse it **as a blueprint** for all the others (doesn't matter how many of them)?*

And this is where our real OOP journey starts: the answer is **absolutely yes** ❗

This is where the `class` comes to a play.
To create a `class` all we need is a `class` keyword followed by an `identifier` (a name we gave to the class) and a block of code in between the curly `{}` braces.

Let's refactor our previous code by introducing a class `Player`. This class will let us create as many objects as we need and it's going to be so much faster!

```
// Example of a VERY simple Monopoly game simulation

let squares = [100, -10, 0, 0, -40, -10, -10, 5, 0, -10, -50, -10, 0, 0, -50, -10];

// Creation of the class
class Player {
  // The constructor is the method triggered with the `new` keyword
  constructor(name, color) {
    this.name = name;
    this.color = color;
    this.position = 0;
    this.cash = 1000;
  }

  // Method move
  move() {
    let dice = 1 + Math.floor(6 * Math.random());
    this.position = (this.position + dice) % squares.length;
    this.cash += squares[this.position];
    if (this.cash < 0) {
      console.log(`Game over for ${this.name}.`);
    }
  }

  // Method displayInfo
  displayInfo() {
    console.log(`${this.name} is at position ${this.position} and has ${this.cash}€`);
  }
}

// --- Initialization of players ---
let player1 = new Player('Joaquim', 'red');
let player2 = new Player('Maxence', 'blue');
let player3 = new Player('Mostafa', 'black');

// --- Turn 1  ---
player1.move();
player2.move();
player3.move();

// --- Turn 2  ---
player1.move();
player2.move();
player3.move();

player1.displayInfo();
player2.displayInfo();
player3.displayInfo();
```

Explain this code

Whaaaaat? That's it?
The code is now much cleaner and shorter!

`constructor` and `new`

> The `constructor` method is a special method for **creating and initializing an object created with a class**. There can only be one special method with the name “constructor” in a class.

> All objects created using the constructor will have the same structure.

Let’s understand the following line:

```
let player1 = new Player('Joaquim', 'red');
```

Explain this code

The keyword `new` execute the `constructor` of the invoked class, which is the class `Player` in our case. It also adds the methods to the object. **In the constructor, `this` refers to the new object created**. Therefore, the previous line is the same as:

```
// Code of the constructor
let player1 = {};
player1.name = 'Joaquim';
player1.color = 'red';
player1.position = 0;
player1.cash = 1000;

// Link of the methods
player1.move = function () {
  /* ... */
};
player1.displayInfo = function () {
  /* ... */
};
```

Explain this code

As you can see, we called *class* player three times using the `new` keyword and we created three new objects (player Joaquim, player Maxence and player Mostafa) which all have the same structure - the same attributes with different values and the same methods (which are personalized as well).

To conclude:

> The constructor is a method which is used to create the **instance objects**.

Inheritance with `extends`

So, we saw that it is possible to make multiple instances based on the same *class*. But is it possible to create something like a *master class* to *separate* all the repetitive attributes and methods and then just **extend** the class with that (master) class? Again, the answer is yes.

> Using the keyword `extends` we can add one more layer of *abstraction* (we will explain this in a bit) and make even cleaner and shorter code.

In JavaScript, we can create a new class that will have all the attributes and methods of another class (and probably some of their own), and for that we will use the keyword `extends`.
This is known as **inheritance**.

> Inheritance is a feature of object-oriented programming that allows code reusability when a class includes property (attribute or method) of another class.

```
class Animal {
  constructor(name, mainColor, sound) {
    this.name = name;
    this.mainColor = mainColor;
    this.sound = sound;
  }
  scream(intensity) {
    console.log(`${this.sound} ${'!'.repeat(intensity)}`);
  }
}

// The class Cat has, by default, all the same attributes and methods as Animal but it will have some that just belong to the cat
class Cat extends Animal {
  constructor(name, mainColor, sound, nbOfLegs) {
    // `super` refers to the constructor of the parent (Animal)
    // with super Cat gets all the attributes and methods of the Animal class
    super(name, mainColor, sound);
    this.nbOfLegs = nbOfLegs; // <== a new attribute, just for cats
  }
}

const garfield = new Cat('Garfield', 'orange', 'Meow', 4);
console.log(garfield);
// {
//   name: 'Garfield',
//   mainColor: 'orange',
//   sound: 'Meow',
//   nbOfLegs: 4
// }

garfield.scream(2); // <== Meow !!
garfield.scream(5); // <== Meow !!!!!

// 2nd example:
class Chameleon extends Animal {
  // Override of the default constructor
  constructor(name) {
    super(name, 'green', '...');
  }
  // Add a new method 'changeColor()'
  changeColor(newColor) {
    this.mainColor = newColor;
  }
}

const pascal = new Chameleon('Pascal');
pascal.changeColor('red');
console.log(pascal);
// {
//   name: 'Pascal',
//   mainColor: 'red', <== notice the difference
//   sound: '...'
```

```
53  // }
```

Explain this code

This is the output of the 1st `console.log`.

```
▼ Cat {name: "Garfield", mainColor: "orange", sound: "Meow", nbOfLegs: 4}
    mainColor: "orange"
    name: "Garfield"
    nbOfLegs: 4
    sound: "Meow"
  ▼ __proto__: Animal
    ▶ constructor: class Cat
    ▶ __proto__: Object
```

As you can see, the methods are not directly saved inside the object but inside `__proto__`. If you want to understand more this behavior, you can take a look at **prototypal inheritance**.

Exercise

Create a class `Rectangle` with:

- A property `width`
- A property `height`
- A method `constructor(width,height)`
- A method `calculatePerimeter()`
- A method `calculateArea()`

Create a class `Square` that extends `Rectangle` add with:

- A property `side` (equals to the `width` and `height`)
- A method `constructor(side)`

```
1   class Rectangle {
2     // TODO
3   }
4
5   class Square extends Rectangle {
6     // TODO
7   }
8
9   const r1 = new Rectangle(6, 7);
10  console.log('Perimeter of r1 =', r1.calculatePerimeter()); // 26
11  console.log('Area of r1 =', r1.calculateArea()); // 42
12
13  const s1 = new Square(5);
14  console.log('Perimeter of s1 =', s1.calculatePerimeter()); // 20
15  console.log('Area of s1 =', s1.calculateArea()); // 25
16
17  const s2 = new Square(10);
18  console.log('Perimeter of s2 =', s2.calculatePerimeter()); // 40
19  console.log('Area of s2 =', s2.calculateArea()); // 100
```

Explain this code

You can edit this example **here**.

Potential interview questions:
When talking about OOP, never miss to mention and explain:

- **class** (*covered earlier*)
- **new** and **constructor** (*covered earlier*)
- **inheritance** (*covered earlier*)
- **abstraction** - means showing just what's necessary to the outside world and hiding all that is unnecessary-to-be-known. Imagine a car engine - you know it works right, because your car is moving, but how it works, you really don't have to know.
- **polymorphism** - means inheriting the method from the parent class but changing its functionality. ex. If you have a class *Person* and this class has a method *move()*. If you extend class Child with class Person most likely the child will crawl when you invoke *move()* method. However, if you extend class *Grownup* with class *Person*, they will walk when *move()* is called. (overly simplified example but just to help you understand.)
- **encapsulation** means grouping the data and the methods that manipulate this data together. The goal is to keep them safe from interference and misuse. We have to aim to hide internal implementation and to organize our code as if it is a black box: it should do the job but the rest of the application should not know how they do it. (this is very much connected with *abstraction*.)

Summary

- A class is a tool to create objects faster
- To create a class, we have to write `class MyClassName { some code here }`
- To inherit from a class, we have to write `class ChildClass extends ParentClass`

Extra resources

- MDN Class
- Classes (ES6) Sample
- ES6 features
- Better JavaScript with ES6, Pt. II: A Deep Dive into Classes
- OOP Concepts "In Simple English"

Mark as completed

PREVIOUS

← JS | OOP - objects, methods and the 'this' keyword

NEXT

JS | OOP - Function constructor vs. Class →