Back to Week 1

JS | OOP - Function constructor vs. Class

SELF GUIDED

Learning goals

After this lesson you will:

- Understand the differences between `function constructor`, and the new `arrow functions`.

Introduction

In this learning unit, we will see how the “old” OOP way using constructor functions (ES5 way) transitioned into a new fancy way with “class”, which we just covered.

Before introducing its ES2015 version (ES6), JavaScript used `prototype–based` programming. What does this mean? It actually means that the object encapsulates all the properties (whether these are just regular data or methods). So there were no classes. You could add new properties to objects at any time. This meant at the same time that objects are individual and independent since they are not instances of the class. To conclude, we can create an object without having to create a class first.

A huge pro was that this way was more flexible and straight forward. However, it had, at the same time, much more inconsistencies and a higher risk of incorrectness. And there was no much of abstraction (read, everything was pretty transparent).

On the opposite end, the class approach brought a much more rigid way of doing the same what constructor functions have done before, and now, class (the blueprint) was required. At the same time, there were much fewer chances to make bugs now, and definitely, there was much less transparency (having classes meant the existence of abstraction layer).

Function constructors into the `class`

Function constructors

As you know already, functions are also objects in JavaScript. The reason (in case you forgot or missed this up to this point) is because they have their own properties and methods, the same as any other object. When functions are used to construct other objects, we are talking about *constructor functions*.

The reasoning behind *constructor functions* is to use them when we know we will have to create a lot of pretty similar objects. In that case, to avoid repeating ourselves, we can use *constructor functions* to create as many objects as needed in an effective way.

Classes introduced in ECMAScript 2015, are primarily syntactical sugar over JavaScript’s existing prototype-based inheritance. The class syntax does not present a new object-oriented inheritance model to JavaScript. Translated to ordinary people language, **the logic behind the inheritance/prototypes is still the same**.

Before, we used to use constructor functions like so:

```
function BankAccount(clientName, currency) {
  this.clientName = clientName;
  this.currency = currency;
  this.balance = 0.0;
}
```

Explain this code

As we can see:

- this is a function statement (declaration)
- the constructor function name should be capitalized (*BankAccount*)
- The body of the function `{...}` has all properties that future object instances will inherit, and each of them is set to be equal to the parameters that are passed into the function.

The `this` keyword refers to an object so that you can access the properties within an object. Thanks to it, we could set the value of a property within an object.

Every instance of the BankAccount constructor function will have the same properties (*clientName*, *currency*, *balance*).

To add methods to the constructor function, we use the following syntax:

```
BankAccount.prototype.showBalance = function () {
  return `${this.currency} ${this.balance}`;
};

BankAccount.prototype.withdrawMoney = function (amount) {
  if (amount <= this.balance) {
    this.balance -= amount;
  } else {
    throw new Error('Not enough funds!');
  }
};

BankAccount.prototype.depositMoney = function (amount) {
  this.balance += amount;
};
```

Explain this code

To create new instances of the constructor function, we use the keyword `new` (the same as we do using the `class` syntax):

```
const account1 = new BankAccount('mike', '$');
```

Explain this code

Now `account1` is an object that has all the properties inherited from the BankAccount constructor function, as well as all the methods:

```
account1.depositMoney(100);
account1.withdrawMoney(25);
account1.showBalance();
// $ 75
```

Explain this code

Inheritance

To inherit from another constructor function, we use the `call` method, passing the object's context, and the attributes we wanted to another object.

```
function BusinessBankAccount(clientName, currency, companyName) {
  BankAccount.call(this, clientName, currency);
  this.companyName = companyName;
}
```

Explain this code

Let's test:

```
const sandbox = new BusinessBankAccount('ana', 'eur', 'sandbox');
console.log(sandbox);

// BusinessBankAccount {
//   clientName: 'ana',
//   currency: 'eur',
//   balance: 0,
//   companyName: 'sandbox'
// }
```

Explain this code

As we can see, `BusinessBankAccount` constructor function inherited *clientName, currency, and balance* from the `BankAccount` constructor function.

But what happens with methods, are they inherited as well? Let's check it out.

```
console.log(sandbox.showBalance());
// TypeError: sandbox.showBalance is not a function
```

Explain this code

To get to that point, we have to use `Object.create()` method to create a `prototype` of the newly created *BusinessBankAccount* constructor function based on the prototype of the *BankAccount* constructor function. And we have to make sure this prototype has its own constructor. Let's see:

```
BusinessBankAccount.prototype = Object.create(BankAccount.prototype);
BusinessBankAccount.prototype.constructor = BusinessBankAccount;

console.log(sandbox.showBalance()); // => eur 0
```

Explain this code

The `class`

And today, we are using `class` syntax to achieve pretty much the same as above:

```
class BankAccount {
  constructor(clientName, currency) {
    this.clientName = clientName;
    this.currency = currency;
    this.balance = 0.0;
  }

  showBalance() {
    return `${this.currency} ${this.balance}`;
  }

  withdrawMoney(amount) {
    if (amount <= this.balance) {
      this.balance -= amount;
    } else {
      throw new Error('not enough funds');
    }
  }

  depositMoney(amount) {
    this.balance += amount;
  }
}

let account1 = new BankAccount('mike', '$');
account1.depositMoney(100);
account1.withdrawMoney(25);
account1.showBalance();
// $ 75
```


As we can see, and already know, 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.

Inheritance

```
class BusinessBankAccount extends BankAccount {
  constructor(clientName, currency, companyName) {
    super(clientName, currency);
    this.companyName = companyName;
  }
}

const sandbox = new BusinessBankAccount('ana', 'eur', 'sandbox');

console.log(sandbox.showBalance()); // => eur 0
```


A constructor can use the `super` keyword to call the constructor of the superclass.

`extends` is a keyword we use to ‘substitute’ the steps we have done with the first version (super confusing *NewFunction.prototype = Object.create(OldFunction.prototype)* and *NewFunction.prototype.constructor = NewFunction*) but in a much cleaner and easier way. So the `extends` keyword is used in class declarations or class expressions to **create a class as a child of another class**.

If there is a constructor present in the subclass, it needs to first call `super()` before using `this` keyword.

> 💡 It's important to know the ES5 way of doing things because the ES6 way is simply syntactic sugar on top of that to make things prettier. This being said, the logic stays the same.

To explore a bit more on the topic of the constructor functions, look for the *self-guided* lessons that follow as well as the resources in the *Extra resources* section.

Extra Resources

- MDN Object-oriented JavaScript for beginners
- W3Schools - JS Object Prototypes

Mark as completed

PREVIOUS
← JS | OOP - class and inheritance

NEXT
LAB | JavaScript Vikings →