MAY 9, 2022 / #BEGINNERS GUIDE

Object-Oriented Programming in JavaScript for Beginners

German Cocca

Hi everyone! In this article we're going to review the main characteristics of object oriented programming (OOP) with practical JavaScript examples.

If you're not familiar with programming paradigms, I recommend you check out the brief intro I recently wrote before diving into this one.

Bring it on!

Table of Contents

Intro to Object-Oriented Programming

As mentioned in my previous article about programming paradigms, the core concept of OOP is to **separate concerns and responsibilities** into **entities.**

Entities are coded as **objects**, and each entity will group a given set of information (**properties**) and actions (**methods**) that can be performed by the entity.

OOP is very useful on large scale projects, as it facilitates code modularity and organization.

By implementing the abstraction of entities, we're able to think about the program in a similar way as our world works, with different actors that perform certain actions and interact with each other.

To better understand how we can implement OOP, we're going to use a practical example in which we're going to code a little video game. We're going to focus on the creation of characters and see how OOP can help us with that.👽 👾 🤖

Classes

So any video game needs characters, right? And all characters have certain **characteristics** (properties) like color, height, name, and so on and **abilities** (methods) like jumping, running, punching, and so on. Objects are the perfect data structure to use to store this kind of information.👌

Say we have 3 different character "species" available, and we want to create 6 different characters, 2 of each species.

A way of creating our characters could be to just manually create the objects using object literals, in this way:

```
const alien1 = {
    name: "Ali",
    species: "alien",
    phrase: () => console.log("I'm Ali the alien!"),
    fly: () => console.log("Zzzzzziiiiiinnnnnggggg!!")
}
const alien2 = {
    name: "Lien",
    species: "alien",
    sayPhrase: () => console.log("Run for your lives!"),
    fly: () => console.log("Zzzzzziiiiiinnnnnggggg!!")
}
const bug1 = {
    name: "Buggy",
    species: "bug",
    sayPhrase: () => console.log("Your debugger doesn't work wit
    hide: () => console.log("You can't catch me now!")
}
const bug2 = {
    name: "Erik",
    species: "bug",
    sayPhrase: () => console.log("I drink decaf!"),
    hide: () => console.log("You can't catch me now!")
}
```

```
    transform: () => console.log("Optimus prime!")
}
const Robot2 = {
    name: "Terminator",
    species: "robot",
    sayPhrase: () => console.log("Hasta la vista, baby!"),
    transform: () => console.log("Optimus prime!")
}
```

See that all characters have the `name` and `species` properties and also the `sayPhrase` method. Moreover, each species has a method that belongs only to that species (for example, aliens have the `fly` method).

As you can see, some data is shared by all characters, some data is shared by each species, and some data is unique to each individual character.

This approach works. See that we can perfectly access properties and methods like this:

```
console.log(alien1.name) // output: "Ali"
console.log(bug2.species) // output: "bug"
Robot1.sayPhrase() // output: "I can cook, swim and dance!"
Robot2.transform() // output: "Optimus prime!"
```

The problem with this is that it doesn't scale well at all and it's error prone. Imagine that our game could have hundreds of characters. We would need to manually set the properties and methods for each of them!

Classes set a blueprint to create objects with predefined properties and methods. By creating a class, you can later on **instantiate** (create) objects from that class, that will inherit all the properties and methods that class has.

Refactoring our previous code, we can create a class for each of our character species, like this:

```
class Alien { // Name of the class
    // The constructor method will take a number of parameters ar
    constructor (name, phrase) {
        this.name = name
        this.phrase = phrase
        this.species = "alien"
    }
    // These will be the object's methods.
    fly = () => console.log("Zzzzzziiiiiinnnnnggggg!!")
    sayPhrase = () => console.log(this.phrase)
}

class Bug {
    constructor (name, phrase) {
        this.name = name
        this.phrase = phrase
        this.species = "bug"
    }
    hide = () => console.log("You can't catch me now!")
    sayPhrase = () => console.log(this.phrase)
}

class Robot {
    constructor (name, phrase) {
        this.name = name
        this.phrase = phrase
        this.species = "robot"
    }
    transform = () => console.log("Optimus prime!")
```

And then we can instantiate our characters from those classes like this:

```
const alien1 = new Alien("Ali", "I'm Ali the alien!")
// We use the "new" keyword followed by the corresponding class n
// and pass it the corresponding parameters according to what was

const alien2 = new Alien("Lien", "Run for your lives!")
const bug1 = new Bug("Buggy", "Your debugger doesn't work with me
const bug2 = new Bug("Erik", "I drink decaf!")
const Robot1 = new Robot("Tito", "I can cook, swim and dance!")
const Robot2 = new Robot("Terminator", "Hasta la vista, baby!")
```

Then again, we can access each object properties and methods like this:

```
console.log(alien1.name) // output: "Ali"
console.log(bug2.species) // output: "bug"
Robot1.sayPhrase() // output: "I can cook, swim and dance!"
Robot2.transform() // output: "Optimus prime!"
```

What is nice about this approach and the use of classes in general is that we can use those "blueprints" to create new objects quicker and more securely than if we did it "manually".

Also, our code is better organized as we can clearly identify where each object properties and methods are defined (in the class). And this makes future changes or adaptations much easier to implement.

> *"a class in a program is a definition of a “type” of custom data structure that includes both data and behaviors that operate on that data. Classes define how such a data structure works, but classes are not themselves concrete values. To get a concrete value that you can use in the program, a class must be instantiated (with the "new" keyword) one or more times."*

- Remember that classes aren't actual entities or objects. Classes are the blueprints or molds that we're going to use to create the actual objects.
- Class names are declared with a capital first letter and camelCase by convention. The class keyword creates a constant, so it cannot be redefined afterwards.
- Classes must always have a constructor method that will later on be used to instantiate that class. A constructor in JavaScript is just a plain old function that returns an object. The only thing special about it is that, when invoked with the "new" keyword, it assigns its prototype as the prototype of the returned object.
- The “this” keyword points to the class itself and is used to define the class properties within the constructor method.
- Methods can be added by simply defining the function name and its execution code.
- JavaScript is a prototype-based language, and within JavaScript classes are used only as syntactic sugar. This doesn't make a huge difference here, but it's good to know and keep in mind. You can read this article if you'd like to know more about this topic.

OOP

OOP is normally explained with 4 key principles that dictate how OOP programs work. These are **inheritance, encapsulation, abstraction and polymorphism.** Let's review each of them.

Inheritance

Inheritance is the ability to **create classes based on other classes.** With inheritance, we can define a **parent class** (with certain properties and methods), and then **children classes** that will inherit from the parent class all the properties and methods that it has.

Let's see this with an example. Imagine all the characters we defined before will be the enemies of our main character. And as enemies, they will all have the "power" property and the "attack" method.

One way to implement that would be just to add the same properties and methods to all the classes we had, like this:

```
...

class Bug {
    constructor (name, phrase, power) {
        this.name = name
        this.phrase = phrase
        this.power = power
        this.species = "bug"
    }
    hide = () => console.log("You can't catch me now!")
    sayPhrase = () => console.log(this.phrase)
    attack = () => console.log(`I'm attacking with a power of ${t
}

class Robot {
    constructor (name, phrase, power) {
    }
    transform = () => console.log("Optimus prime!")
    sayPhrase = () => console.log(this.phrase)
    attack = () => console.log(`I'm attacking with a power of ${t
}

const bug1 = new Bug("Buggy", "Your debugger doesn't work with me
const Robot1 = new Robot("Tito", "I can cook, swim and dance!", 1

console.log(bug1.power) //output: 10
Robot1.attack() // output: "I'm attacking with a power of 15!"
```

But you can see we're repeating code, and that's not optimal. A better way would be to declare a parent "Enemy" class which is then extended by all enemy species, like this:

```
class Enemy {
    constructor(power) {
        this.power = power
    }

    attack = () => console.log(`I'm attacking with a power of ${t
}


class Alien extends Enemy {
    constructor (name, phrase, power) {
        super(power)
        this.name = name
        this.phrase = phrase
        this.species = "alien"
    }
    fly = () => console.log("Zzzzzziiiiiinnnnnggggg!!")
    sayPhrase = () => console.log(this.phrase)
}

...
```

properties, and methods are declared like simple functions.

On the children class, we use the `extends` keyword to declare the parent class we want to inherit from. Then on the constructor method, we have to declare the "power" parameter and use the `super` function to indicate that property is declared on the parent class.

When we instantiate new objects, we just pass the parameters as they were declared in the corresponding constructor function and *voilà!* We can now access the properties and methods declared in the parent class.😎

```
const alien1 = new Alien("Ali", "I'm Ali the alien!", 10)
const alien2 = new Alien("Lien", "Run for your lives!", 15)

alien1.attack() // output: I'm attacking with a power of 10!
console.log(alien2.power) // output: 15
```

Now let's say we want to add a new parent class that groups all our characters (no matter if they're enemies or not), and we want to set a property of "speed" and a "move" method. We can do that like this:

```
class Character {
    constructor (speed) {
        this.speed = speed
    }

    move = () => console.log(`I'm moving at the speed of ${this.s
}

class Enemy extends Character {
    attack = () => console.log(`I'm attacking with a power of ${t
}


class Alien extends Enemy {
    constructor (name, phrase, power, speed) {
        super(power, speed)
        this.name = name
        this.phrase = phrase
        this.species = "alien"
    }
    fly = () => console.log("Zzzzzziiiiiinnnnnggggg!!")
    sayPhrase = () => console.log(this.phrase)
}
```

First we declare the new "Character" parent class. Then we extend it on the Enemy class. And finally we add the new "speed" parameter to the `constructor` and `super` functions in our Alien class.

We instantiate passing the parameters as always, and *voilà* again, we can access properties and methods from the "grandparent" class.👴

```
const alien1 = new Alien("Ali", "I'm Ali the alien!", 10, 50)
const alien2 = new Alien("Lien", "Run for your lives!", 15, 60)

alien1.move() // output: "I'm moving at the speed of 50!"
console.log(alien2.speed) // output: 60
```

Now that we know more about inheritance, let's refactor our code so we avoid code repetition as much as possible:

```
    }
    move = () => console.log(`I'm moving at the speed of ${this.s
}

class Enemy extends Character {
    constructor(name, phrase, power, speed) {
        super(speed)
        this.name = name
        this.phrase = phrase
        this.power = power
    }
    sayPhrase = () => console.log(this.phrase)
    attack = () => console.log(`I'm attacking with a power of ${t
}


class Alien extends Enemy {
    constructor (name, phrase, power, speed) {
        super(name, phrase, power, speed)
        this.species = "alien"
    }
    fly = () => console.log("Zzzzzziiiiiinnnnnggggg!!")
}

class Bug extends Enemy {
    constructor (name, phrase, power, speed) {
        super(name, phrase, power, speed)
        this.species = "bug"
    }
    hide = () => console.log("You can't catch me now!")
}

class Robot extends Enemy {
    constructor (name, phrase, power, speed) {
        super(name, phrase, power, speed)
        this.species = "robot"
    }
    transform = () => console.log("Optimus prime!")
}


const alien1 = new Alien("Ali", "I'm Ali the alien!", 10, 50)
const alien2 = new Alien("Lien", "Run for your lives!", 15, 60)
const bug1 = new Bug("Buggy", "Your debugger doesn't work with me
const bug2 = new Bug("Erik", "I drink decaf!", 5, 120)
```

See that our species classes look much smaller now, thanks to the fact that we moved all shared properties and methods to a common parent class. That's the kind of efficiency inheritance can help us with.😉

Some things to keep in mind about inheritance:

- A class can only have one parent class to inherit from. You can't extend multiple classes, though there're are hacks and ways around this.
- You can extend the inheritance chain as much as you want, setting parent, grandparent, great grandparent classes and so on.
- If a child class inherits any properties from a parent class, it must first assign the parent properties calling the `super()` function before assigning its own properties.

An example:

```
// This works:
class Alien extends Enemy {
    constructor (name, phrase, power, speed) {
        super(name, phrase, power, speed)
        this.species = "alien"
    }
    fly = () => console.log("Zzzzzziiiiiinnnnnggggg!!")
}

// This throws an error:
class Alien extends Enemy {
    constructor (name, phrase, power, speed) {
}
```

- When inheriting, all parent methods and properties will be inherited by the children. We can't decide what to inherit from a parent class (same as we can't choose what virtues and defects we inherit from our parents. 😅 We'll get back to this when we talk about composition).
- Children classes can override the parent's properties and methods.

To give an example, in our previous code, the Alien class extends the Enemy class and it inherits the `attack` method which logs `I'm attacking with a power of ${this.power}!`:

```
class Enemy extends Character {
    constructor(name, phrase, power, speed) {
        super(speed)
        this.name = name
        this.phrase = phrase
        this.power = power
    }
    sayPhrase = () => console.log(this.phrase)
    attack = () => console.log(`I'm attacking with a power of ${t
}


class Alien extends Enemy {
    constructor (name, phrase, power, speed) {
        super(name, phrase, power, speed)
        this.species = "alien"
    }
    fly = () => console.log("Zzzzzziiiiiinnnnnggggg!!")
}
```

Let's say we want the `attack` method to do a different thing in our Alien class. We can override it by declaring it again, like this:

```
class Enemy extends Character {
    constructor(name, phrase, power, speed) {
        super(speed)
        this.name = name
        this.phrase = phrase
        this.power = power
    }
    sayPhrase = () => console.log(this.phrase)
    attack = () => console.log(`I'm attacking with a power of ${t
}


class Alien extends Enemy {
    constructor (name, phrase, power, speed) {
        super(name, phrase, power, speed)
        this.species = "alien"
    }
    fly = () => console.log("Zzzzzziiiiiinnnnnggggg!!")
    attack = () => console.log("Now I'm doing a different thing,
}

const alien1 = new Alien("Ali", "I'm Ali the alien!", 10, 50)
alien1.attack() // output: "Now I'm doing a different thing, HA!'
```

Encapsulation

Encapsulation is another key concept in OOP, and it stands for an object's capacity to "decide" which information it exposes to "the outside" and which it doesn't. Encapsulation is implemented through **public and private properties and methods.**

```
// Here's our class
class Alien extends Enemy {
    constructor (name, phrase, power, speed) {
        super(name, phrase, power, speed)
        this.species = "alien"
    }
    fly = () => console.log("Zzzzzziiiiiinnnnnggggg!!")
}

// Here's our object
const alien1 = new Alien("Ali", "I'm Ali the alien!", 10, 50)

// Here we're accessing our public properties and methods
console.log(alien1.name) // output: Ali
alien1.sayPhrase() // output: "I'm Ali the alien!"
```

To make this clearer, let's see how private properties and methods look like.

Let's say we want our Alien class to have a `birthYear` property, and use that property to execute a `howOld` method, but we don't want that property to be accessible from anywhere else other than the object itself. We could implement that like this:

```
class Alien extends Enemy {
    #birthYear // We first need to declare the private property,

    constructor (name, phrase, power, speed, birthYear) {
        super(name, phrase, power, speed)
        this.species = "alien"
        this.#birthYear = birthYear // Then we assign its value w
    }
    fly = () => console.log("Zzzzzziiiiiinnnnnggggg!!")
    howOld = () => console.log(`I was born in ${this.#birthYear}
```

Then we can access the `howOld` method, like this:

```
alien1.howOld() // output: "I was born in 10000"
```

But if we try to access the property directly, we'll get an error. And the private property won't show up if we log the object.

```
console.log(alien1.#birthYear) // This throws an error
console.log(alien1)
// output:
// Alien {
//     move: [Function: move],
//     speed: 50,
//     sayPhrase: [Function: sayPhrase],
//     attack: [Function: attack],
//     name: 'Ali',
//     phrase: "I'm Ali the alien!",
//     power: 10,
//     fly: [Function: fly],
//     howOld: [Function: howOld],
//     species: 'alien'
//   }
```

Encapsulation is useful in cases where we need certain properties or methods for the inner working of the object, but we don't want to expose that to the exterior. Having private properties/methods ensures we don't "accidentally" expose information we don't want.

information that is relevant to the problem's context. In plain English, only expose to the outside the properties and methods that you're going to use. If it's not needed, don't expose it.

This principle is closely related to encapsulation, as we can use public and private properties/methods to decide what gets exposed and what doesn't.

Polymorphism

Then there is polymorphism (sounds really sophisticated, doesn't it? OOP names are the coolest... 🙃). Polymorphism means "many forms" and is actually a simple concept. It's the ability of one method to return different values according to certain conditions.

For example, we saw that the Enemy class has the `sayPhrase` method. And all our species classes inherit from the Enemy class, which means they all have the `sayPhrase` method as well.

But we can see that when we call the method on different species, we get different results:

```
const alien2 = new Alien("Lien", "Run for your lives!", 15, 60)
const bug1 = new Bug("Buggy", "Your debugger doesn't work with me

alien2.sayPhrase() // output: "Run for your lives!"
bug1.sayPhrase() // output: "Your debugger doesn't work with me!'
```

And that's because we passed each class a different parameter at instantiation. That's one kind of polymorphism, **parameter-based**.

to when we have a parent class that sets a method and the child overrides that method to modify it in some way. The example we saw previously applies perfectly here as well:

```
class Enemy extends Character {
    constructor(name, phrase, power, speed) {
        super(speed)
        this.name = name
        this.phrase = phrase
        this.power = power
    }
    sayPhrase = () => console.log(this.phrase)
    attack = () => console.log(`I'm attacking with a power of ${t
}


class Alien extends Enemy {
    constructor (name, phrase, power, speed) {
        super(name, phrase, power, speed)
        this.species = "alien"
    }
    fly = () => console.log("Zzzzzziiiiiinnnnnggggg!!")
    attack = () => console.log("Now I'm doing a different thing,
}

const alien1 = new Alien("Ali", "I'm Ali the alien!", 10, 50)
alien1.attack() // output: "Now I'm doing a different thing, HA!'
```

This implementation is polymorphic because if we commented out the `attack` method in the Alien class, we would still be able to call it on the object:

```
alien1.attack() // output: "I'm attacking with a power of 10!"
```

Object Composition

Object composition is a technique that works as an alternative to inheritance.

When we talked about inheritance we mentioned that child classes always inherit all parent methods and properties. Well, by using composition we can assign properties and methods to objects in a more flexible way than inheritance allows, so objects only get what they need and nothing else.

We can implement this quite simply, by using functions that receive the object as a parameter and assign it the desired property/method. Let's see it in an example.

Say now we want to add the flying ability to our bug characters. As we've seen in our code, only aliens have the `fly` method. So one option could be to duplicate the exact same method in the `Bug` class:

```
class Alien extends Enemy {
    constructor (name, phrase, power, speed) {
        super(name, phrase, power, speed)
        this.species = "alien"
    }
    fly = () => console.log("Zzzzzziiiiiinnnnnggggg!!")
}

class Bug extends Enemy {
    constructor (name, phrase, power, speed) {
        super(name, phrase, power, speed)
        this.species = "bug"
    }
```

Another option would be to move the `fly` method up to the `Enemy` class, so it can be inherited by both the `Alien` and `Bug` classes. But that also makes the method available to classes that don't need it, like `Robot`.

```
class Enemy extends Character {
    constructor(name, phrase, power, speed) {
        super(speed)
        this.name = name
        this.phrase = phrase
        this.power = power
    }
    sayPhrase = () => console.log(this.phrase)
    attack = () => console.log(`I'm attacking with a power of ${t
    fly = () => console.log("Zzzzziiiiinnnnngggg!!")
}


class Alien extends Enemy {
    constructor (name, phrase, power, speed) {
        super(name, phrase, power, speed)
        this.species = "alien"
    }
}

class Bug extends Enemy {
    constructor (name, phrase, power, speed) {
        super(name, phrase, power, speed)
        this.species = "bug"
    }
    hide = () => console.log("You can't catch me now!")
}

class Robot extends Enemy {
    constructor (name, phrase, power, speed) {
        super(name, phrase, power, speed)
        this.species = "robot"
    }
    transform = () => console.log("Optimus prime!")
```

As you can see, inheritance causes problems when the starting plan we had for our classes changes (which in the real world is pretty much always). Object composition proposes an approach in which objects get properties and methods assigned only as they need them.

In our example, we could create a function and its only responsibility would be to add the flying method to any object that receives as parameter:

```
const bug1 = new Bug("Buggy", "Your debugger doesn't work with me

const addFlyingAbility = obj => {
    obj.fly = () => console.log(`Now ${obj.name} can fly!`)
}

addFlyingAbility(bug1)
bug1.fly() // output: "Now Buggy can fly!"
```

And we could have very similar functions for each power or ability we may want our monsters to have.

As you can surely see, this approach is a lot more flexible than having parent classes with fixed properties and methods to inherit. Whenever an object needs a method, we just call the corresponding function and that's it.👌

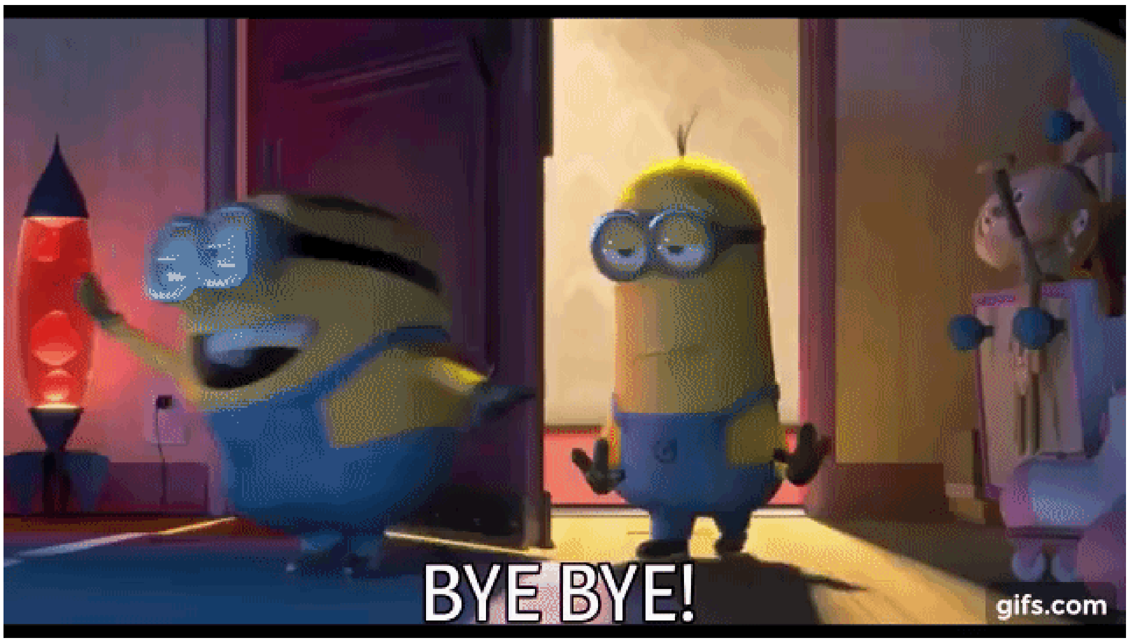
Here's a nice video that compares inheritance with composition.

tackle huge projects by creating the abstraction of entities. Each entity will be responsible for certain information and actions, and entities will be able to interact with each other too, much like how the real world works.

In this article we learned about classes, inheritance, encapsulation, abstraction, polymorphism and composition. These are all key concepts in the OOP world. And we've also seen various examples of how OOP can be implemented in JavaScript.

As always, I hope you enjoyed the article and learned something new. If you want, you can also follow me on LinkedIn or Twitter.

Cheers and see you in the next one! ✌️



along my path to becoming the best developer I can be.

If this article was helpful, share it .

Learn to code for free. freeCodeCamp's open source curriculum has helped more than 40,000 people get jobs as developers.

Get started

Trending Books and Handbooks

REST APIs
Clean Code
TypeScript
JavaScript
AI Chatbots
Command Line
GraphQL APIs
CSS Transforms
Access Control
REST API Design
PHP
Java
Linux
React
CI/CD
Docker
Golang
Python
Node.js
Todo APIs
JavaScript Classes
Front-End Libraries
Express and Node.js
Python Code Examples