**JavaScript Basics #4: Object-Oriented Programming**

In the previous article, we talked about a new data type called the objects. In computer programming, objects are very commonly used as a way to organize code. Programmers would group values and functions with close relationships to each other, and put them in the same object, which makes them easier to access. This method of organizing your code is called object-oriented programming. In this article, we’ll discuss how these ideas could be applied in JavaScript.

Table of Contents

* [Encapsulation](https://www.ericsdevblog.com/index.php/2021/10/javascript-basics-4-object-oriented-programming/#Encapsulation)
* [Methods](https://www.ericsdevblog.com/index.php/2021/10/javascript-basics-4-object-oriented-programming/#Methods)
* [Prototypes](https://www.ericsdevblog.com/index.php/2021/10/javascript-basics-4-object-oriented-programming/#Prototypes)
* [Classes](https://www.ericsdevblog.com/index.php/2021/10/javascript-basics-4-object-oriented-programming/#Classes)
* [Getters, Setters, and Statics](https://www.ericsdevblog.com/index.php/2021/10/javascript-basics-4-object-oriented-programming/#Getters_Setters_and_Statics)
* [Inheritance](https://www.ericsdevblog.com/index.php/2021/10/javascript-basics-4-object-oriented-programming/#Inheritance)

**ENCAPSULATION**

The core idea of object-oriented programming is to split a program into small pieces, and each piece only minds its own business. People working on other pieces of code don’t need to know how this piece of code is written, or that it even exists.

Sometimes the different pieces need to communicate with each other to perform a more complicated task. These pieces of code can “talk” to each other through interfaces. An interface is a set of functions or bindings that work on a more abstract level, and they are made **public**, meaning they can be “seen” by the code outside of the object. While the actual implementation is hidden inside the object as **private** properties, meaning they cannot be seen or accessed by the outside code. This way of separating the interface from the implementation is called encapsulation.

Most programming languages have very distinctive methods of denoting public properties and private properties, usually with keywords public and private. JavaScript, however, does not have this functionality built-in, at least not yet. But JavaScript programmers still follows this idea of encapsulation, by putting an underscore character (\_) at the beginning of the properties that should be made **private**. But since this is not JavaScript’s built-in functionality, technically you could still access these properties from the outside, but that is something you should never do, for security reasons.

**METHODS**

As you know, methods are just properties with functions as their values. This is a simple method:

// Create a new empty object

let rabbit = {};

// Add a method named speak() to the empty object

rabbit.speak = function(line) {

console.log(`The rabbit says '${line}'`);

}

// Excute the mathod

rabbit.speak("I'm alive.");

Sometimes, the method needs to do something to the object it was called on, such as taking two numbers that are stored in the object, and add them up, or taking a string value from the object and process it. To do this, we can use the this keyword, which is a binding that automatically points to the object that was called on. Let’s take a look at an example:

// Create the method named speak()

function speak(line) {

console.log(`The ${this.type} rabbit says '${line}'`);

}

/\*

Create an object named whiteRabbit, with two properties, "type"

and "speak". By using the "this" keyword in the method "speak",

we are able to access the "type" property in the same object.

\*/

// In this case, this.type = "white".

let whiteRabbit = { type: "white", speak };

// In this case, this.type = "hungry".

let hungryRabbit = { type: "hungry", speak };

**PROTOTYPES**

Look at the following code:

// Create an empty object

let empty = {};

console.log(empty.toString); // -> function toString(){...}

console.log(empty.toString); // -> [object Object]

Notice that even though we defined an empty object, we still manage to pull a property from it. Well, technically, that property is not from the object, it’s from the object’s prototype. A prototype is basically another object on which our empty object is based, and it acts as a fallback source of properties. If you are trying to access a property that does not exist in the object, then its prototype will be searched for that property.

JavaScript offers a method (Object.getPrototypeOf()) that returns the prototype of a data type. For example, let’s try finding out the prototype of that empty object we just created:

console.log(Object.getPrototypeOf(empty)); // -> {..., constructor: Object(), ...}

console.log(Object.getPrototypeOf(empty) == Object.prototype); // -> true

The Object.prototype is the ancestral root of all objects that we create, but not all data types share the same prototype. For instance, the functions derive from Function.prototype, and arrays derive from Array.prototype.

console.log(Object.getPrototypeOf([]) == Array.prototype);

// -> true

console.log(Object.getPrototypeOf(Math.max) == Function.prototype);

// -> true

However, since those prototypes are still just objects, they also have a prototype, and that is usually Object.project. This is why almost all of the data types we’ve talked about have a toString method that converts objects into a string representation.

In fact, we can create our own prototype and use Object.create() method to create objects using a specific prototype.

// Create an object, which we'll use as a prototype

let protoRabbit = {

speak(line) {

console.log(`The ${this.type} rabbit says '${line}'`);

}

};

// Create a new object using the protoRabbit as the prototype

let killerRabbit = Object.create(protoRabbit);

killerRabbit.type = "killer";

// Try to access the speak() method from the killerRabbit object

killerRabbit.speak("SKREEEE!");

// -> The killer rabbit says 'SKREEE!'

**CLASSES**

In object-oriented programming, there is a concept called class, which works just like the prototypes. A class defines the shape of a type of object (just like prototypes), what kind of properties and methods it has. Such an object is called an instance of the class.

To create an instance of the class, we need to make a new object, which derives from the prototype/class. But you also have to make sure that the object has the properties that an instance of the class is supposed to have, not just the ones derived from the prototype/class. This is what a constructor function does.

// An example of a constructor function

function makeRabbit(type) {

// Create a new object using protoRabbit as prototype

let rabbit = Object.create(protoRabbit);

// Add a property named "type".

// Note that the senond type is the variable that is passed to the function

rabbit.type = type;

// returns the newly created object

return rabbit;

}

If you are familiar with other programming languages that follow the idea of object-oriented programming, you’ll see that this is a very awkward way of defining a class and constructor function, but I think it does help you understand what a constructor function is. Luckily, after 2015, JavaScript offered us a new and more standard way of making a class, by using the keyword class.

let Rabbit = class Rabbit {

constructor(type) {

this.type = type;

}

speak(line) {

console.log(`The ${this.type} rabbit says '${line}'`);

}

}

To create an instance of this class, we can use the keyword new.

let killerRabbit = new Rabbit("killer");

let blackRabbit = new Rabbit("black");

The constructor() function which we defined in the class will be automatically executed when you run this code.

**GETTERS, SETTERS, AND STATICS**

Now, let’s focus on the interface part of object-oriented programming. In case you forgot, the interface is the part of the object that can be “seen” from the outside. Programmers use the interface to make different pieces of code work together to solve a complex problem.

There are typically two types of these interface methods, getters and setters. Getters retrieves information from the object, and setters write information to the object. Let’s consider this example of a temperature converter.

class Temperature {

constructor(celsius) {

this.celsius = celsius;

}

get fahrenheit() {

return this.celsius \* 1.8 + 32;

}

set fahrenheit(value) {

this.celsius = (value - 32) / 1.8;

}

static fromFahrenheit(value) {

return new Temperature((value - 32) / 1.8);

}

}

let temp = new Temperature(22);

Notice that we have a static method in this example. Statics are not part of the interface, they are in charge of attaching additional properties to your constructor function, instead of the prototype. In our example, it is used to provide a different way of creating a class instance.

**INHERITANCE**

JavaScript also provides us with an easy way to create a class based on another class, with new definitions of some of its properties. For example, the following class defines a matrix. In case you don’t know, a matrix is a two-dimensional array.

class Matrix {

constructor(width, height, element = (x, y) => undefined) {

this.width = width;

this.height = height;

this.content = [];

for (let y = 0; y < height; y++) {

for (let x = 0; x < width; x++) {

this.content[y \* width + x] = element(x, y);

}

}

}

get(x, y) {

return this.content[y \* this.width + x];

}

set(x, y, value) {

this.content[y \* this.width + x] = value;

}

}

There is another type of matrix that is called a symmetric matrix. It has all the characteristics of a regular matrix, except it is symmetric along its diagonal. To create such a matrix and avoid rewriting the same code all over again, we can make the SymmetricMatrix extends the Matrix class like this:

class SymmetricMatrix extends Matrix {

constructor(size, element = (x, y) => undefined) {

super(size, size, (x, y) => {

if (x < y) return element(y, x);

else return element(x, y);

});

}

set(x, y, value) {

super.set(x, y, value);

if (x != y) {

super.set(y, x, value);

}

}

}

let matrix = new SymmetricMatrix(5, (x, y) => `${x},${y}`);

console.log(matrix.get(2, 3));

// → 3,2