Creating a Classes

In this reading, you learn about extending classes using basic inheritance in JavaScript. Here are the key points:

* Inheritance in JavaScript is based on the prototype object.
* You can use the class keyword to define a class and create objects from it using the new keyword.
* The constructor function is used to initialize properties on the object instance.
* You can add methods to classes, which will be shared by all instances of the class.
* The super keyword is used to specify which properties are inherited from the superclass in the subclass.
* You can override inherited methods in the subclass and combine them with custom code.
* The prototype holds shared methods, while the constructor sets up instance-specific values.
* Polymorphism can be achieved by inheriting from a base class and overriding inherited behavior.

Overall, classes in JavaScript allow you to separate object data from shared functionality and create more complex object-oriented relationships.

**Creating classes**

By the end of this reading, you should be able to explain, with examples, the concept of extending classes using basic inheritance to alter behaviors within child classes.

By now, you should know that inheritance in JavaScript is based around the prototype object.

All objects that are built from the prototype share the same functionality.

When you need to code more complex OOP relationships, you can use the *class* keyword and its easy-to-understand and easy-to-reason-about syntax.

Imagine that you need to code a *Train* class.

Once you've coded this class, you'll be able to use the keyword *new* to instantiate objects of the *Train* class.

For now though, you first need to define the *Train* class, using the following syntax:

class Train {}

class name

So, you use the *class* keyword, then specify the name of your class, with the first letter capitalized, and then you add an opening and a closing curly brace.

In between the curly braces, the first piece of code that you need to define is the **constructor**:

The *constructor* will be used to build properties on the future object instance of the *Train* class.

For now, let's say that there are only two properties that each object instance of the *Train* class should have at the time it gets instantiated: *color* and *lightsOn*.

class Train {

    constructor(color, lightsOn) {

        this.color = color;

        this.lightsOn = lightsOn;

    }

}

Notice the syntax of the constructor. The constructor is a special function in my Train class.

First of all, notice that there is no function keyword. Also, notice that the keyword constructor is used to define this function. You give your constructor function parameters inside an opening and closing parenthesis, just like in regular functions. The names of parameters are color and lightsOn.

Next, inside the constructor function's body, you assigned the passed-in color parameter's value to this.color, and the passed-in lightsOn parameter's value to this.lightsOn.

What does this this keyword here represent?

**It's the future object instance of the**Train**class**.

Essentially, this is all the code that you need to write to achieve two things:

1. This code allows me to **build new instances of the**Train**class**.
2. Each object instance of the Train class that I build will have its own custom properties of color and lightsOn.

Now, to actually build a new instance of the Train class, I need to use the following syntax:

new Train()

new keyword make an object or instances of class

Inside the parentheses, you need to pass values such as *"red"* and *false*, for example, meaning that the *color* property is set to *"red"* and the *lightsOn* property is set to *false*.

And, to be able to interact with the new object built this way, you need to assign it to a variable.

Putting it all together, here's your first train:

var myFirstTrain = new Train('red', false);

Just like any other variable, you can now, for example, console log the *myFirstTrain* object:

console.log(myFirstTrain); // Train {color: 'red', lightsOn: false}

You can continue building instances of the *Train* class. Even if you give them exactly the same properties, they are still separate objects.

var mySecondTrain = new Train('blue', false);

var myThirdTrain = new Train('blue', false);

However, this is not all that classes can offer.

You can also add methods to classes, and these methods will then be shared by all future instance objects of my *Train* class.

class Train {

    constructor(color, lightsOn) {

        this.color = color;

        this.lightsOn = lightsOn;

    }

    toggleLights() {

        this.lightsOn = !this.lightsOn;

    }

    lightsStatus() {

        console.log('Lights on?', this.lightsOn);

    }

    getSelf() {

        console.log(this);

    }

    getPrototype() {

        var proto = Object.getPrototypeOf(this);

        console.log(proto);

    }

}

Now, there are four methods on your Train class:  toggleLights(), lightsStatus(),  getSelf() and getPrototype().

1. The toggleLights method uses the logical not operator, !. This operator will change the value stored in the lightsOn property of the future instance object of the Train class; hence the !this.lightsOn. And the = operator to its left means that it will get assigned to this.lightsOn, meaning that it will become the new value of the lightsOn property on that given instance object.
2. The lightsStatus() method on the Train class just reports the current status of the lightsOn variable of a given object instance.
3. The getSelf() method prints out the properties on the object instance it is called on.
4. The getPrototype() console logs the prototype of the object instance of the Train class. The prototype holds all the properties shared by all the object instances of the Train class. To get the prototype, you'll be using JavaScript's built-in Object.getPrototypeOf() method, and passing it this object - meaning, the object instance inside of which this method is invoked.

Now you can build a brand new train using this updated Train class:

var train4 = new Train('red', false);

train4.toggleLights(); // undefined

train4.lightsStatus(); // Lights on? true

train4.getSelf(); // Train {color: 'red', lightsOn: true}

train4.getPrototype(); // {constructor: f, toggleLights: f, ligthsStatus: f, getSelf: f, getPrototype: f}

The result of calling *toggleLights()* is the change of true to false and vice-versa, for the *lightsOn* property.

The result of calling *lightsStatus()* is the console logging of the value of the *lightsOn* property.

The result of calling *getSelf()* is the console logging the entire object instance in which the *getSelf()* method is called. In this case, the returned object is the *train4* object. Notice that this object gets returned only with the properties ("data") that was build using the *constructor()* function of the *Train* class. That's because all the methods on the *Train* class do not "live" on any of the instance objects of the *Train* class - instead, they live on the prototype, as will be confirmed in the next paragraph.

Finally, the result of calling the *getPrototype()* method is the console logging of all the properties on the *prototype*. When the *class* syntax is used in JavaScript, this results in **only shared methods being stored on the prototype**, while the *constructor()* function sets up the mechanism for saving instance-specific values ("data") at the time of object instantiation.

Thus, in conclusion, the class syntax in JavaScript allows us to clearly separate individual object's data - which exists on the object instance itself - from the shared object's functionality (methods), which exist on the prototype and are shared by all object instances.

However, this is not the whole story.

It is possible to implement polymorphism using classes in JavaScript, by inheriting from the base class and then overriding the inherited behavior. To understand how this works, it is best to use an example.

In the code that follows, you will observe another class being coded, which is named *HighSpeedTrain* and inherits from the *Train* class.

This makes the *Train* class a base class, or the super-class of the *HighSpeedTrain* class. Put differently, the *HighSpeedTrain* class becomes the sub-class of the *Train* class, because it inherits from it.

To inherit from one class to a new sub-class, JavaScript provides the *extends* keyword, which works as follows:

class HighSpeedTrain extends Train {

}

As in the example above, the sub-class syntax is consistent with how the base class is defined in JavaScript. The only addition here is the *extends* keyword, and the name of the class from which the sub-class inherits.

Now you can describe how the *HighSpeedTrain* works. Again, you can start by defining its constructor function:

Notice the slight difference in syntax in the constructor of the *HighSpeedTrain* class, namely the use of the *super* keyword.

In JavaScript classes, *super* is used to specify what property gets inherited from the super-class in the sub-class.

In this case, I choose to inherit both the properties from the *Train* super-class in the *HighSpeedTrain* sub-class.

These properties are *color* and *lightsOn*.

Next, you add the additional properties of the HighSpeedTrain class inside its constructor, namely, the passengers and highSpeedOn properties.

Next, inside the constructor body, you use the *super* keyword and pass in the inherited *color* and *lightsOn* properties that come from the *Train* class. On subsequent lines you assign *passengers* to *this.passengers*, and *highSpeedOn* to *this.highSpeedOn*.

Notice that in addition to the inherited properties, you also **automatically inherit** all the methods that exist on the *Train* prototype, namely, the *toggleLights()*, *lightsStatus()*, *getSelf()*, and *getPrototype()* methods.

Now let's add another method that will be specific to the *HighSpeedTrain* class: the *toggleHighSpeed()* method.

class HighSpeedTrain extends Train {

    constructor(passengers, highSpeedOn, color, lightsOn) {

        super(color, lightsOn);

        this.passengers = passengers;

        this.highSpeedOn = highSpeedOn;

    }

    toggleHighSpeed() {

        this.highSpeedOn = !this.highSpeedOn;

        console.log('High speed status:', this.highSpeedOn);

    }

}

Additionally, imagine you realized that you don't like how the *toggleLights()* method from the super-class works, and you want to implement it a bit differently in the sub-class. You can add it inside the *HighSpeedTrain* class.

So, how did you override the behavior of the original *toggleLights()* method?

Well in the super-class, the *toggleLights()* method was defined as follows:

toggleLights() {

    this.lightsOn = !this.lightsOn;

}

You realized that the *HighSpeedTrain* method should reuse the existing behavior of the original *toggleLights()* method, and so you used the *super.toggleLights()* syntax to inherit the entire super-class' method.

Next, you also inherit the behavior of the super-class' *lightsStatus()* method - because you realize that you want to have the updated status of the *lightsOn* property logged to the console, whenever you invoke the *toggleLights()* method in the sub-class.

Finally, you also add the third line in the re-implemented *toggleLights()* method, name

console.log('Lights are 100% operational.');

You've added this third line to show that I can combine the "borrowed" method code from the super-class with your own custom code in the sub-class.

Now you're ready to build some train objects.

var train5 = new Train('blue', false);

var highSpeed1 = new HighSpeedTrain(200, false, 'green', false);

You've built the *train5* object of the *Train* class, and set its *color* to *"blue"* and its *lightsOn* to *false*.

Next, you've built the *highSpeed1* object to the *HighSpeedTrain* class, setting *passengers* to *200*, *highSpeedOn* to *false*, *color* to *"green"*, and lightsOn to false.

Now you can test the behavior of *train5*, by calling, for example, the *toggleLights()* method, then the *lightsStatus()* method:

train5.lightsStatus(); // Lights on? true

Here's the entire completed code for this lesson:

class Train {

    constructor(color, lightsOn) {

        this.color = color;

        this.lightsOn = lightsOn;

    }

    toggleLights() {

        this.lightsOn = !this.lightsOn;

    }

    lightsStatus() {

        console.log('Lights on?', this.lightsOn);

    }

    getSelf() {

        console.log(this);

    }

    getPrototype() {

        var proto = Object.getPrototypeOf(this);

        console.log(proto);

    }

}

class HighSpeedTrain extends Train {

    constructor(passengers, highSpeedOn, color, lightsOn) {

        super(color, lightsOn);

        this.passengers = passengers;

        this.highSpeedOn = highSpeedOn;

    }

    toggleHighSpeed() {

        this.highSpeedOn = !this.highSpeedOn;

        console.log('High speed status:', this.highSpeedOn);

    }

    toggleLights() {

        super.toggleLights();

        super.lightsStatus();

        console.log('Lights are 100% operational.');

    }

}

var myFirstTrain = new Train('red', false);

console.log(myFirstTrain); // Train {color: 'red', lightsOn: false}

var mySecondTrain = new Train('blue', false);

var myThirdTrain = new Train('blue', false);

var train4 = new Train('red', false);

train4.toggleLights(); // undefined

train4.lightsStatus(); // Lights on? true

train4.getSelf(); // Train {color: 'red', lightsOn: true}

train4.getPrototype(); // {constructor: f, toggleLights: f, ligthsStatus: f, getSelf: f, getPrototype: f}

var train5 = new Train('blue', false);

var highSpeed1 = new HighSpeedTrain(200, false, 'green', false);

train5.toggleLights(); // undefined

train5.lightsStatus(); // Lights on? true

highSpeed1.toggleLights(); // Lights on? true, Lights are 100% operational.

Notice how the *toggleLights()* method behaves differently on the *HighSpeedTrain* class than it does on the *Train* class.

Additionally, it helps to visualize what is happening by getting the prototype of both the *train5* and the *highSpeed1*

train5.getPrototype(); // {constructor: ƒ, toggleLights: ƒ, lightsStatus: ƒ, getSelf: ƒ, getPrototype: ƒ}

highSpeed1.getPrototype(); // Train {constructor: ƒ, toggleHighSpeed: ƒ, toggleLights: ƒ}

The returned values in this case might initially seem a bit tricky to comprehend, but actually, it is quite simple:

1. The prototype object of the train5 object was created when you defined the class Train. You can access the prototype using Train.prototype syntax and get the prototype object back.
2. The prototype object of the highSpeed1 object is this object: {constructor: ƒ, toggleHighSpeed: ƒ, toggleLights: ƒ}. In turn this object has its own prototype, which can be found using the following syntax: HighSpeedTrain.prototype.\_\_proto\_\_. Running this code returns: {constructor: ƒ, toggleLights: ƒ, lightsStatus: ƒ, getSelf: ƒ, getPrototype: ƒ}.

Prototypes seem easy to grasp at a certain level, but it's easy to get lost in the complexity. This is one of the reasons why class syntax in JavaScript improves your developer experience, by making it easier to reason about the relationships between classes. However, as you improve your skills, you should always strive to understand your tools better, and this includes prototypes. After all, JavaScript is just a tool, and you've now "peeked behind the curtain".

In this reading, you've learned the very essence of how OOP with classes works in JavaScript. However, this is not all.

In the lesson on designing an object-oriented program, you'll learn some more useful concepts. These mostly have to do with coding your classes so that it's even easier to create object instances of those classes in JavaScript.

**Using class instance as another class' constructor's property**

Consider the following example:

class StationaryBike {

    constructor(position, gears) {

        this.position = position

        this.gears = gears

    }

}

class Treadmill {

    constructor(position, modes) {

        this.position = position

        this.modes = modes

    }

}

class Gym {

    constructor(openHrs, stationaryBikePos, treadmillPos) {

        this.openHrs = openHrs

        this.stationaryBike = new StationaryBike(stationaryBikePos, 8)

        this.treadmill = new Treadmill(treadmillPos, 5)

    }

}

var boxingGym = new Gym("7-22", "right corner", "left corner")

console.log(boxingGym.openHrs) //

console.log(boxingGym.stationaryBike) //

console.log(boxingGym.treadmill) //

In this example, there are three classes defined: StationaryBike, Treadmill, and Gym.

The StationaryBike class is coded so that its future object instance will have the position and gears properties. The position property describes where the stationary bike will be placed inside the gym, and the gears propery gives the number of gears that that stationary bike should have.

The Treadmill class also has a position, and another property, named modes (as in "exercise modes").

The Gym class has three parameters in its constructor function: openHrs, stationaryBikePos, treadmillPos.

This code allows me to instantiate a new instance object of the Gym class, and then when I inspect it, I get the following information:

* the openHrs property is equal to "7-22" (that is, 7am to 10pm)
* the stationaryBike property is an object of the StationaryBike type, containing two properties: position and gears
* the treadmill property is an object of the Treadmill type, containing two properties: position and modes

…………………………………………………………………………………………………………………………………………………………………

# Default Parameters

A useful a ES6 feature allows me to set a default parameter inside a function definition First, .

What that means is, I'll use an ES6 feature which allows me to set a default parameter inside a function definition, which goes hand in hand with the defensive coding approach, while requiring almost no effort to implement.

For example, consider a function declaration without default parameters set:

function noDefaultParams(number) {

    console.log('Result:', number \* number)

}

Obviously, the *noDefaultParams* function should return whatever number it receives, *squared*.

However, what if I call it like this:

noDefaultParams(); // Result: NaN

JavaScript, due to its dynamic nature, doesn't throw an error, but it does return a non-sensical output.

Consider now, the following improvement, using default parameters:

function withDefaultParams(number = 10) {

    console.log('Result:', number \* number)

}

Default params allow me to build a function that will run with default argument values even if I don't pass it any arguments, while still being flexible enough to allow me to pass custom argument values and deal with them accordingly.

This now allows me to code my classes in a way that will promote easier object instantiation.

Consider the following class definition:

Now I'll instantiate an object of the *NoDefaultParams* class, and run the *calculate()* method on it. Obviously, the *bool1* should be set to *true* on invocation to make this work, but I'll set it to false on purpose, to highlight the point I'm making.

This example might highlight the reason sometimes weird error messages appear when some software is used - perhaps the developers just didn't have enough time to build it better.

However, now that you know about default parameters, this example can be improved as follows:

class WithDefaultParams {

    constructor(num1 = 1, num2 = 2, num3 = 3, string1 = "Result:", bool1 = true) {

        this.num1 = num1;

        this.num2 = num2;

        this.num3 = num3;

        this.string1 = string1;

        this.bool1 = bool1;

    }

    calculate() {

        if(this.bool1) {

            console.log(this.string1, this.num1 + this.num2 + this.num3);

            return;

        }

        return "The value of bool1 is incorrect"

    }

}

var better = new WithDefaultParams();

better.calculate(); // Result: 6

This approach improves the developer experience of my code, because I no longer have to worry about feeding the WithDefaultParameters class with all the arguments. For quick tests, this is great, because I no longer need to worry about passing the proper arguments.

Additionally, this approach really shines when building inheritance hierarchies using classes, as it makes it possible to provide only the custom properties in the sub-class, while still accepting all the default parameters from the super-class constructor.

In conclusion, in this reading I've covered the following:

* How to approach designing an object-oriented program in JavaScript
* The role of the extends and super keywords
* The importance of using default parameters.