**Programming With JavaScript Notes**

**Week 3**

**Programming Paradigms**

**Introduction to Functional Programming**

Return Values from Functions

Many functions, by default, return the value of undefined.

Example:

console.log('Hello');

//Hello

//undefined

The console log function is built so as to not have the explicitly set return value, it gets the default return value of undefined.

Here’s an example exemplifying how to remove the undefined value:

function consoleLog(val) {

console.log(val)

return val

}

Now when you run the function, the output is:

Hello

'Hello'

It’s useful because you can return values from one function inside another function.

Here’s another example where a function returns double of a number it’s recieved:

function doubleIt(num) {

return num \* 2

}

Now, I’ll code another function that builds an object with a specific value:

function objectMaker(val) {

return {

prop: val

}

}

You can call the objectMaker() function with any value, such as:

objectMaker(20);

The return value will be an object with a single prop key set to 20:

{prop:20}

Now consider this code:

doubleIt(10).toString()

The above code returns a the number 20 as a string, “20”.

You can even combine the custom function calls as follows:

objectMaker( doubleIt(100) );

Which will return:

{prop: 200}

In summary, by JavaScript allowing me to use the return keyword as described above, I can have multiple function calls, returning data and manipulating values, based on whatever coding challenge I have in front of me.

Being able to return custom values is one of the foundations that makes functional programming possible.

The Functional Programming Paradigm

There are several styles of coding, also known as paradigms. A common style is called functional programming, or FP for short.

In functional programming, lots of functions and variables are used.

function getTotal(a,b) {

return a + b

}

var num1 = 2;

var num2 = 3;

var total = getTotal(num1, num2);

When writing FP code, we keep data and functionality separate and pass data into functions only when we want something computed.

function getDistance(mph, h) {

return mph \* h

}

var mph = 60;

var h = 2;

var distance = getDistance(mph, h);

In functional programming, functions return new values and then use those values somewhere else in the code.

function getDistance(mph, h) {

return mph \* h

}

var mph = 60;

var h = 2;

var distance = getDistance(mph, h);

console.log(distance); // <====== THIS HERE!

Another style is object-oriented programming (OOP). In this style, we group data and functionality as properties and methods inside objects.

For example, if I have a virtualPet object, I can give it a sleepy property and a nap() method:

var virtualPet = {

sleepy: true,

nap: function() {}

}

In OOP, methods update properties stored in the object instead of generating new return values.

For example, if you check the sleepy property on the virtualPet object, you can confirm that it’s set to true.

However, once you’ve ran the nap() method on the virtualPet object, will the sleepy property’s value change?

//creating an object

var virtualPet = {

sleepy: true,

nap: function() {

this.sleepy = false

}

}

console.log(virtualPet.sleepy) // true

virtualPet.nap()

console.log(virtualPet.sleepy) // false

OOP helps model real-life objects. It works best when the grouping of properties and data in an object makes logical sense – meaning the properties and methods “belong together”.

The goal here is not to discuss OOP in depth; instead, this is just the simplest explanation of what it is and how it works, in order to make the single most important distinction between FP and OOP.

To summarize, we can say that Functional Programming paradigm works by keeping the data and functionality separate. It’s counterpart, OOP, works by keeping the data and functionality grouped in meaningful objects.

Here are some more concepts and ideas in functional programming:

* First-class functions
* Higher-order function
* Pure functions and side-effects

There are many more concepts and principles in functional programming.

First-class functions:

A function in JavaScript is just another value that can:

* Pass to other functions.
* Save in a variable.
* Return from other functions.

In other words, a functions in JavaScript is just a value – from this vantage point, almost no different than a string or number.

For example, in JavaScript it’s perfectly normal to pass a function invocation to another function.

To explain how this works consider the following program:

function addTwoNums(a, b) {

console.log(a + b)

}

function randomNum() {

return Math.floor((Math.random() \* 10) + 1);

}

function specificNum() { return 42 };

var useRandom = true;

var getNumber;

if(useRandom) {

getNumber = randomNum

} else {

getNumber = specificNum

}

addTwoNums(getNumber(), getNumber())

The program starts with the addTwoNums() function whose definition I’ve already used earlier in various variations. The reason why this function is a recurring example is because it’s so simple that it helps explain concepts that otherwise might be a bit harder to grasp.

Next, I code a function named randomNum() which returns a random number between 0 and 10.

I then code another function named specificNum() which return a specific number, the number 42.

Next, I save a variable named useRandom, and I set it to the Boolean value of true. I declare another variable named getNumber.

Where it get’s interesting:

On the next several lines, I have an if else statement. The if condition is executed when the value of useRandom is set to true. If that’s the case, the entire randomNum() functions’s declaration is saved into the getNumber variable. Otherwise, I’m saving the entire specificNum() function’s declaration into the getNumber variable.

In other words, based on the useRandom being set to true or false, the getNumber variable will be assigned either the randomNum() function declaration or the specificNum() function declaration.

With all this code set, I can then invoke the addTwoNums() function, passing it the invocation of the getNumber() variables as its first and second arguments.

\*\* This works because functions in JavScript are truly first-class citizens, which can be assigned to variable names and passed around just like I would pass around a single string, a number, an object, etc.

Bringing us to the second foundational concept of functional programming, which is the concept of higher-order functions.

Higher-order functions:

A higher-order function is a function that has either one or both of the following characteristics:

* It accepts other functions as arguments.
* It returns functions when invoked.

There’s no “special way” of defining higher-order functions in JavaScript. It is simply a feature of the language. The language itself allows me to pass a function to another function, or to return a function from another function.

Continuing from the previous section, consider the following code, in which I’m redefining the addTwoNums() function so that it’s a higher-order function:

function addTwoNums(getNumber1, getNumber2) {

console.log(getNumber1() + getNumber2());

}

You can think of the above function declaration of addTwoNums as describing how it will deal with the getNumber1 and getNumber2 inputs: once it receives them as arguments, it will then attempt invoking them and concatenating the values from those invocations.

For example:

addTwoNums(specificNum, specificNum); // returned number is 84

addTwoNums(specificNum, randomNum); // returned number is 42 + some random number

Pure functions and side-effects:

A pure function returns the exact same result as long as it’s given the same values.

An example of a pure function is the addTwoNums() function from the previous section:

function addTwoNums(a, b) {

console.log(a + b)

}

This function will always return the same output, based on the input. For example, as long as we give it a specific value, say 5 and 6.

addTwoNums(5,6); // 11

The output will always be the same.

Another rule for a function to be considered pure is that it should not have side-effects. A side-effect is any instance where a function makes a change outside itself.

This includes:

* Changing variable values outside of the function itself, or even relying on outside variables.
* Calling a Browser API (even the console itself)
* Calling Math.random() – since the value cannot be reliably repeated

The topic of pure and impure functions can get somewhat complex.

For now, just know that this concept exists and that it is related to functional programming.

Additional Resources

MDN Functions Guide

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions>

MDN Glossary: Recursion

<https://developer.mozilla.org/en-US/docs/Glossary/Recursion>

MDN Glossary: Scope

<https://developer.mozilla.org/en-US/docs/Glossary/Scope>

Functional Programming in JavaScript

<https://www.toptal.com/javascript/functional-programming-javascript>

MDN: First-class Functions

<https://developer.mozilla.org/en-US/docs/Glossary/First-class_Function>

**Introduction to Object Oriented Programming**

Object Oriented Programming Principles

There are many benefits to using the OOP paradigm.

OOP helps developers to mimic the relationship between objects in the real world. In a way, it helps you to reason about relationships between things in your software, just like you would in the real world. Thus, OOP is an effective approach to come up with solutions in thee code you write.

OOP will also:

* Allow you to write modular code.
* Makes your code more flexible.
* Makes your code reusable.

The Principles of OOP

There are four fundamental principles of OOP.

1. Inheritance
2. Encapsulation
3. Abstraction
4. Polymorphism

Objects exist in a hierarchal structure. Meaning the original base or super class for everything is the Object class, all objects derive from this class. This allows us to utilize the Object.create() method to create or instansiate objects of our classes.

class Animal { /\* ...class code here... \*/ }

var myDog = Object.create(Animal)

console.log (Animal)

A more common method of creating objects from classes is to use the new keyword. When using a default or empty constructor method, JavaScript implicitly calls the Object superclass to cerate the instance.

class Animal { /\* ...class code here... \*/ }

var myDog = new Animal()

console.log (Animal)

OOP Principle: Inheritance

Inheritance is one of the foundations of object-oriented programming.

In essence, it’s a very simple concept that works like this:

1. There’s a base class of a “thing”.
2. There is one or more sub-classes of “things” that inherit the porperties of the base class. (sometimes referred to as the “super-class”)
3. There might be some other sub-sub classes of “things” that inherit those classes from point 2.

Note: Each sub-class inherits from its super-class. In turn, a sub-class might also be a super-class, if there are classes inheriting from that sub-class.

Simpler Example:

1. There is a base class of “Animal”.
2. There is another class, a sub-class inheriting from “Animal”, and this class is “Bird”.
3. Next, there is another class, inheriting from “Bird”, and this class is “Eagle”.

Thus, I’m modelling objects from the real world by constructing relationships between Animal, Bird, and Eagle. Each of them are separate classes, meaning, each of them are separate blueprints for specific object instances that can be constructed as needed.

To setup the inheritance relationship between classes in JavaScript, I can use the extends keyword, as in class B extends A.

Inheritance hierarchy example:

class Animal { /\* ...class code here... \*/ }

class Bird extends Animal { /\* ...class code here... \*/ }

class Eagle extends Bird { /\* ...class code here... \*/ }

OOP Principles: Encapsulation

In simple terms, encapsulation has to do with making a code implementation “hidden” from other users, in the sense that they don’t have to know how my code works in order to ”consume” the code.

For example, when I run the following code:

"abc".toUpperCase();

I don’t have to worry or even waste time thinking about how the toUpperCase() method works. All I want is to use it, since I know it’s available to me. Even if the underlying syntax – that is, the implementation of the toUpperCase() method changes – as long as it doesn’t brake my code, I don’t have to worry about what it does in the background, or even how it does it.

OOP Principles: Abstraction

Abstraction is all about writing code in a way that will make it more generalized.

The concepts of encapsulation and abstraction are often misunderstood because their differences can feel blurry.

Try thinking of it in the following terms:

* An abstraction is about extracting the concept of what you’re trying to do, rather than dealing with a specific manifestation of that concept.
* Encapsulation is about you not having access to, or not being concerned with, how some implementation works internally.

While both the encapsulation and abstraction are important concepts in OOP, it requires more experience with programming in general to really delve into this topic.

Principles in OOP: Polymorphism

Polymorphism is a word derived from the Greek language meaning “multiple forms”. An alternative translation might be: “something that can take many shapes”.

To further understand, consider real life objects:

* A door has a bell. It could be said that the bell is a property of the door object. This bell can be rung. When would someone ring a bell on the door? Obviously, to get someone to show up at the door.
* Now consider a bell on a bicycle. A bicycle has a bell. It could be said that the bell is a property of the bicycle object. This bell could also be rung. However, the reason, the intention, and the result of somebody ringing the bell on a bicycle is not the same as ringing the bell on the door.

These concepts can be coded in JavaScript as follows:

const bicycle = {

bell: function() {

return "Ring, ring! Watch out, please!"

}

}

const door = {

bell: function() {

return "Ring, ring! Come here, please!"

}

}

So, I can access the bell() method on the bicycle object, using the following syntax:

bicycle.bell(); // "Get away, please"

I can also access the bell() method on the door object, using this syntax:

door.bell(); // "Come here, please"

At this point one can conclude that the exact same name of the method can have the exact opposite intent, based on what object it is used for.

Now, to make this code truly polymorphic, I will add another function decleration:

function ringTheBell(thing) {

console.log(thing.bell())

}

Now I’ve declared the ringTheBell() function. It accepts a thing parameter – which I expect to be an object, namely, either the bicycle object or the door object.

So now, if I call the ringTheBell() function and pass it to the bicycle as its single argument, here’s the output:

ringTheBell(bicycle); // Ring, ring! Watch out, please!

However, if I invoke the ringTheBell() function and pass it to the door object, you get:

ringTheBell(door); // "Ring, ring! Come here, please!"

You’ve now seen an example of the exact same function producing different results, based on the context in which it is used.

Here’s another example, the concatenation operator, used by calling the built-in concat() method.

If I use the concat() method on two strings, it behaves exactly the same as if I used the + operator.

"abc".concat("def"); // 'abcdef'

I can also use the concat() method on two arrays:

["abc"].concat(["def"]); // ['abc', 'def']

Consider the + operator on two arrays with one member each:

["abc"] + ["def"]; // ["abcdef"]

This means that the concat() method is exhibiting polymorphic behavior since it behaves differently based on the context – in this case, based on what data types I give it.

To reiterate, polymorphism is useful because it allows developers to build objects that can have the exact sane functionality, namely, functions with the exact same name, which behave exactly the same. However, at the same time, you can override some parts of the shared functionality or even the complete functionality, in some other parts of the OOP structure.

Here’s an example of polymorphism using classes in JavaScript:

class Bird {

useWings() {

console.log("Flying!")

}

}

class Eagle extends Bird {

useWings() {

super.useWings()

console.log("Barely flapping!")

}

}

class Penguin extends Bird {

useWings() {

console.log("Diving!")

}

}

var baldEagle = new Eagle();

var kingPenguin = new Penguin();

baldEagle.useWings(); // "Flying! Barely flapping!"

kingPenguin.useWings(); // "Diving!"

The Penguin and Eagle sub-classes both inherit from the Bird super-class. The Eagle sub-class inherits the useWings() method from the Bird class, but extends it with an additional console log. The Penguin sub-class doesn’t inherit the useWings() class – instead it has its own implementation, although the Penguin class itself does extend the Bird class.

Do some practice with the code above, try creating some of your own classes. (Try using real life examples)

Constructors

JavaScript has a number of built in object types, such as:

Math, Date, Object, Function, Boolean, Symbol, Array, Map, Set, Promise, JSON, etc.

These objects are sometimes referred to as “native objects”.

Constructor functions, commonly referred to as just “constructors”, are special functions that allow us to build instances of these built-in native objects. All the constructors are capitalized.

To use a constructor, you must prepend it with the operator new.

For example, to create a new instance of the Date object, I can run: new Date(). What I will get back is the current datetime.

However, not all built-in objects come with a constructor function. An example of such an object type is the built-in Match object.

Running new Math() throws an Uncaught TypeError, informing us that Math is not a constructor.

Thus, you can conclude that some built-in objects do have constructors, when they serve a particular purpose: to allow us to instantiate a specific instance of a given object’s constructor. The built-in Date object is perfectly suited for having a constructor because each new date object instance I build should have unique data by definition, since it’s going to be a different timestamp – it’s going to be built at a different moment in time.

Other built-in objects that don’t have constructors, such as the Math object, don’t need a constructor. They’re just static objects whose properties and methods can be accessed directly, from the built-in object itself. In other words, there is no point in building an instance of the built-in Math object to be able to use its functionality.

For example, if I want to use the pow method of the Math object to calculate exponential values, there’s no need to build in instance of the Math object to do so. For example, to get the number 2 to the power of 5, I’d run:

Math.pow(2,5); // 🡪 32

There’s no need to build an instance of the Math object since there would be nothing that needs to be stored in that specific object’s instance.

Besides constructor functions for the built-in objects, I can also define custom constructor functions. Here’s an example:

function Icecream(flavor) {

this.flavor = flavor;

this.meltIt = function() {

console.log(`The ${this.flavor} icecream has melted`);

}

}

Now, I can make as many icecreams as I want

function Icecream(flavor) {

this.flavor = flavor;

this.meltIt = function() {

console.log(`The ${this.flavor} icecream has melted`);

}

}

let kiwiIcecream = new Icecream("kiwi");

let appleIcecream = new Icecream("apple");

kiwiIcecream; // --> Icecream {flavor: 'kiwi', meltIt: ƒ}

appleIcecream; // --> Icecream {flavor: 'apple', meltIt: ƒ}

I’ve just built two instance objects of Icecream type.

The most common case of new is to use it with one of the built-on object types.

Note that using constructor functions on all built-in objects is sometimes not the best approach.

This is especially true for object constructors of primitive types, namely: String, Number, and Boolean.

For example, using the built-in String constructor, I can built new strings:

let apple = new String("apple");

apple; // --> String {'apple'}

The apple variable is an object of type String.

Let’s see how the apple object differs from the following pear variable:

let pear = "pear";

pear; // --> "pear"

The pear variable is a string literal, that is, a primitive JavaScript value.

The pear variable, being a primitive value, will always be more performant than the apple variable, which is an object.

Besides being more performant, due to the fact that each object in JavaScript is unique, you can’t compare a String object with another String object, even when their values are identical.

In other words, if you compare new String(‘plum’) === new String ( ‘plum’ ), you’ll get back false, while “plum === “plum” returns true. You’re getting false when comparing objects because it is not the values that you pass to the constructor that are being compared, but rather the memory location where the objects are saved.

Besides not using constructors to build object versions of primitives, you are better off not using constructors when constructing plain, regular objects.

Instead of new Object, you should stick to the object literal syntax: {}.

A RegExp object is another built-in object in JavaScript. It’s used to pattern-match strings using what’s known as “Regular Expressions.” Regular Expressions exist in many languages, not just JavaScript.

In JavaScript, you can build an instance of the RegExp constructor using new RegExp.

Alternatively, you can use a pattern literal instead of RegExp. Here’s an example of using /d/ as a pattern literal, passed-in as an argument to the match method on a string.

"abcd".match(/d/); // null

"abcd".match(/a/); // ['a', index: 0, input: 'abcd', groups: undefined]

Instead of using Array, Function, and RegExp constructors, you should use their array literal, function literal, and pattern literal varieties: [], (), {}, and /()/.

However, when building objects of other built-on types, we can use the constructor.

Examples:

new Date();

new Error();

new Map();

new Promise();

new Set();

new WeakSet();

new WeakMap();

Creating Classes

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

Here’s an example for creating a class:

class Train {}

Now let’s define a constructor in the class:

class Train {

constructor() {

}

}

The constructor will be used to build properties on the future object that each object instance of the train class.

Let’s add two properties that each object instance of the Train class should have at the time:

class Train {

constructor(color, lightsOn) {

this.color = color;

this.lightsOn = lightsOn;

}

}

The constructor is a special function in my Train class.

Notice that there is no function keyword. Also notice, 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 functions body, you assigned the passed-in color parameters value to this.color, and the passed-in lightsOn parameters value to this.lightsOn.

With this code you can build new instances of the train class, and each of those objects will have their own properties.

new Train()

Inside the parenthesis, you’ll need to pass values such as, “red” and false, so that the color property and the lightsOn property is set.

To interact with the new object built this way, you need to assign it to a variable:

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

Just like any other variable, you can now console log the myFirstTrain object.

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

You can keep building new instances of the train class, even if they have the same properties they are separate objects.

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

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

Methods can also be added to classes. Methods will be shared by all future instance objects of the Train class.

For example:

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 the Train class:

toggleLights(), lightStatus(), getSelf(), and getPrototype().

1. 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.lightOn. 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. lightsStatus() method on the Train class just reports the current status of the lightsOn variable of a given object instance.
3. getSelf() method prints out the properties on the object instance it’s called on.
4. getPrototype() method 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 can use JavaScript’s built-in Object.getPrototypeOf() method, and passing it this object – meaning, the object instance inside of which this method is invoked.

Build a brand new train using this updated Train class:

var train4 = new Train('red', false);

And now, you can run each of its methods, one after another, to confirm the behavior:

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}

Calling toggleLights() results in the change of true to false and vice versa, for the lightsOn property.

Calling lightStatus() results in a console log of the value of the lightsOn property.

Calling getSelf() results in console logging the entire object instance in which the command 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 built 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.

getPrototype() results in console logging all the properties of 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.

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 objects functionality (methods), which exist on the prototype and are shared by all object instances.

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

Another class will be coded, named HighSpeedTrain and inherits from the Train class.

This makes the Train class a base class, or the super-class of the HighSpeedTrain class. Or the HighSpeedTrain class becomes the sub-class of the Train class.

To inherit from one class to a new sub-calss, JavaScript provides the keyword, extends:

class HighSpeedTrain extends Train {

constructor(passengers, highSpeedOn, color, lightsOn) {

super(color, lightsOn);

this.passengers = passengers;

this.highSpeedOn = highSpeedOn;

}

}

Notice the difference in syntax in the constructor of the HighSpeedTrain class, especially the use of the keyword, super.

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

Next, you can add additional properties for the sub-class inside its constructor. (passengers and highSpeedOn)

After adding the super keyword and passing the properties that come from the Train class, you can assign passengers using this.passengers, and highSpeedOn to this.HighSpeedOn.

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

Now, add another method that will be specific to the HighSpeedTrain class:

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);

}

}

You might’ve 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.

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.toggleLigths();

super.lightsStatus();

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

}

}

To override the original toggleLights() method…

In the super-class the toggleLights() method was defined like:

toggleLights() {

this.lightsOn = !this.lightsOn;

}

First, we reused the existing behavior of the toggleLights() method by, using the super.toggleLights() syntax to inherit the entire super-class’ method.

Also, inherit the behavior of the super-class’ lightsStatus() method – because we want to have the updated status of the LightsOn property logged in the console, whenever the toggleLights() methos in the cub-class in invoked.

Finally, also add the third line in the re-implemented toggleLights() method:

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

The third line is added to show that you can combine the “borrowed” method code from the super-class with your own custom code in the sub-class.

Now, build some train objects:

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

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

The train5 object of the Train class has been built, and its color is set to “blue” and lightsOn to false.

Next, the highSpeed1 object of the HighSpeedTrain class has been built, setting passengers to 200, highSpeedOn to false, color to “green”, and lightsOn to false.

Now, test the behavior of train5 by calling, for example, the toggleLights() method, then the lightStatus() method:

train5.toggleLights(); // undefined

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

For the entire completed code of 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 HighSpeedTrain class than it does on the Train class.

Additionally, it helps to visualize what’s happening by getting the prototype of both the train5 and highSpeed1 trains:

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 may look complex but they’re really 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: f, toggleHighSpeed: f, toggleLights: f}. In turn this object has its own prototype, which can be found using the following syntax: HighSpeedTrain.prototype.\_\_proto\_\_. Running this code returns: {constructor: f, toggleLights: f, lightsStatus: f, getSelf: f, getPrototype: f}.

Prototypes are easy to grasp at a certain level, but it’s easy to get lost in the complexity. It’s just one reason why class syntax in JavaScript improves your developer experience, by making it easier to reason about the relationship 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 another tool.

These mostly have to do with coding classes so that it’s easier to create object instances of those classes.

Using class instances as another classes 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 describes where the stationary bike will be placed inside the gym, and the gears property gives the number of gears that that stationary bike should have.

The Treadmill class alos 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 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 ES6 feature allows a default parameter set inside a function definition.

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 zero effort to implement.

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

function noDefaultParams(number) {

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

}

Obviously, the noDefaultParams function should return whatever number it receives, *squared*. However, what if it’s called 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 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 no arguments are passed to it, while still being flexible enough to allow me to pass custom argument values and deal with them accordingly.

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

Consider the following class definition:

class NoDefaultParams {

constructor(num1, num2, num3, string1, bool1) {

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"

}

}

Now, 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:

var fail = new NoDefaultParams(1,2,3,false);

fail.calculate(); // 'The value of bool1 is incorrect'

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 shines when using 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.

Designing an OOP Program

Prepare to build the following inheritance hierarchy:

Animal

/ \

Cat. Bird

/ \ \

HouseCat Tiger Parrot

Two keywords essential for OOP with classes in JS:

1. extends
2. super

Based on the above hierarchy, I can code the Animal class like this:

class Animal {

// ... class code here ...

}

Then, I can code a sub-class, for example, cat:

class Cat extends Animal {

// ... class code here ...

}

This is how the extends keyword is used to setup inheritance of relationships.

The super keyword allows you to “borrow” functionality from a super-class, in a sub-class.

Now, start thinking of how you’d like to implement the OOP class hierarchy.

Before you even begin think of things like: What should go into the base class of Animal? In other words, what will all the sub-classes inherit from the base class? What are the specific properties and methods that separate each class from others? Generally, how will my classes relate to one another?

Once considered, take the plan as follows:

1. The Animal classes constructor will have two properties: color and energy.
2. The animal classes prototype will have three methods: isActive(), sleep(), and getColor().
3. The isActive() method, will lower the value of energy until it hits 0. The isActive() method will also report the updated value of energy. If energy is at zero, the animal object will immediately go to sleep, by invoking the sleep() method based on the said condition.
4. The getColor() method will console log the value in the color property.
5. That Cat class will inherit from Animal, with the additional sound, canJumpHigh, and canClimgTrees properties specific to the Cat class. It will also have its own makeSound() method.
6. The Bird class wil also inherit from Animal, but its own specific properties will be different form Cat. The Bird class will have the sound and canFly properties, and the makeSound method too.
7. The HouseCat class will extend the Cat class, and it will have its own houseCatSound as its special property. Additionally, it will override the makeSound() method from the Cat class, but it will do so in an interesting way. If the makeSound() method, on invocation, receives a single option argument – set to true, then it will run super.makeSound() – in other words, run the code from the parent class (Cat) with the addition of running the console.log(this.houseCatSound). Effectively, this means that the makeSound() method on the HouseCat classes instance object will have two separate behaviors, based on whether we pass it true or false.
8. The Tiger class will also inherit from Cat, and it will come with its own tigerSound property, while the rest of the behavior will be pretty much the same as in the HouseCat class.
9. The Parrat class will extend the Bird class, with its own canTalk property, and its own makeSound() method, working with two conditionals: one that checks if the value of true was passed to makeSound during invocation, and another that checks the value stored inside this.canTalk property.

Now that I’ve fully explained how all the code in my class hierarchy should work, I might start implementing it by adding all the requirements as comments inside the code structure.

At this stage, with all the requirements written down as comments, my code should be as follows:

class Animal {

// constructor: color, energy

// isActive()

// if energy > 0, energy -=20, console log energy

// else if energy <= 0, sleep()

// sleep()

// energy += 20

// console.log energy

}

class Cat extends Animal {

// constructor: sound, canJumpHigh, canClimbTrees, color, energy

// makeSound()

// console.log sound

}

class Bird extends Animal {

// constructor: sound, canFly, color, energy

// makeSound()

// console.log sound

}

class HouseCat extends Cat {

// constructor: houseCatSound, sound, canJumpHigh, canClimbTrees, color, energy

// makeSound(option)

// if (option)

// super.makeSound()

// console.log(houseCatSound)

}

class Tiger extends Cat {

// constructor: tigerSound, sound, canJumpHigh, canClimbTrees, color, energy

// makeSound(option)

// if (option)

// super.makeSound()

// console.log(tigerSound)

}

class Parrot extends Bird {

// constructor: canTalk, sound, canJumpHigh, canClimbTrees, color, energy

// makeSound(option)

// if (option)

// super.makeSound()

// if (canTalk)

// console.log("talking!")

}

Now that I’ve coded the requirements inside comments of otherwise empty classes, I can start coding each class as per my specifications.

Let’s start with the Animal class:

class Animal {

constructor(color = 'yellow', energy = 100) {

this.color = color;

this.energy = energy;

}

isActive() {

if(this.energy > 0) {

this.energy -= 20;

console.log('Energy is decreasing, currently at:', this.energy)

} else if(this.energy == 0){

this.sleep();

}

}

sleep() {

this.energy += 20;

console.log('Energy is increasing, currently at:', this.energy)

}

getColor() {

console.log(this.color)

}

}

Each animal object, no matter which one it is, will share the properties of color and energy.

Now I can code the Cat and Bird classes:

class Cat extends Animal {

constructor(sound = 'purr', canJumpHigh = true, canClimbTrees = true, color, energy) {

super(color, energy);

this.sound = sound;

this.canClimbTrees = canClimbTrees;

this.canJumpHigh = canJumpHigh;

}

makeSound() {

console.log(this.sound);

}

}

class Bird extends Animal {

constructor(sound = 'chirp', canFly = true, color, energy) {

super(color, energy);

this.sound = sound;

this.canFly = canFly;

}

makeSound() {

console.log(this.sound);

}

}

Note that if I didn’t use the super keyword in our sub-classers, once I’d run the code, I’d get an error: Uncaught ReferenceError: Must call super constructor in derived class before accessing ‘this’ or returning from derived constructor.

And now I can code the three remaining classes: HouseCat, Tiger, and Parrot.

class HouseCat extends Cat {

constructor(houseCatSound = "meow", sound,canJumpHigh,canClimbTrees, color,energy) {

super(sound,canJumpHigh,canClimbTrees, color,energy);

this.houseCatSound = houseCatSound;

}

makeSound(option) {

if (option) {

super.makeSound();

}

console.log(this.houseCatSound);

}

}

class Tiger extends Cat {

constructor(tigerSound = "Roar!", sound,canJumpHigh,canClimbTrees, color,energy) {

super(sound,canJumpHigh,canClimbTrees, color,energy);

this.tigerSound = tigerSound;

}

makeSound(option) {

if (option) {

super.makeSound();

}

console.log(this.tigerSound);

}

}

class Parrot extends Bird {

constructor(canTalk = false, sound,canFly, color,energy) {

super(sound,canFly, color,energy);

this.canTalk = canTalk;

}

makeSound(option) {

if (option) {

super.makeSound();

}

if (this.canTalk) {

console.log("I'm a talking parrot!");

}

}

}

Now that I’ve setup the entire inheritance structure, I can build various animal objects.

For example, I can build two parrots: one that can talk and another that can’t:

var polly = new Parrot(true); // we're passing `true` to the constructor so that polly can talk

var fiji = new Parrot(false); // we're passing `false` to the constructor so that fiji can't talk

polly.makeSound(); // 'chirp', 'I'm a talking parrot!'

fiji.makeSound(); // 'chirp'

polly.color; // yellow

polly.energy; // 100

polly.isActive(); // Energy is decreasing, currently at: 80

var penguin = new Bird("shriek", false, "black and white", 200); // setting all the custom properties

penguin; // Bird {color: 'black and white', energy: 200, sound: 'shriek', canFly: false }

penguin.sound; // 'shriek'

penguin.canFly; // false

penguin.color; // 'black and white'

penguin.energy; // 200

penguin.isActive(); // Energy is decreasing, currently at: 180

Or a pet cat  
  
var leo = new HouseCat();

You can have the cat purr or not:

// leo, no purring please:

leo.makeSound(false); // meow

// leo, both purr and meow now:

leo.makeSound(true); // purr, meow

Or build a tiger:

var cuddles = new Tiger();

He can purr and roar, or just purr:

cuddles.makeSound(false); // Roar!

cuddels.makeSound(true); // purr, Roar!

Here’s all the code put together:

class Animal {

constructor(color = 'yellow', energy = 100) {

this.color = color;

this.energy = energy;

}

isActive() {

if(this.energy > 0) {

this.energy -= 20;

console.log('Energy is decreasing, currently at:', this.energy)

} else if(this.energy == 0){

this.sleep();

}

}

sleep() {

this.energy += 20;

console.log('Energy is increasing, currently at:', this.energy)

}

getColor() {

console.log(this.color)

}

}

class Cat extends Animal {

constructor(sound = 'purr', canJumpHigh = true, canClimbTrees = true, color, energy) {

super(color, energy);

this.sound = sound;

this.canClimbTrees = canClimbTrees;

this.canJumpHigh = canJumpHigh;

}

makeSound() {

console.log(this.sound);

}

}

class Bird extends Animal {

constructor(sound = 'chirp', canFly = true, color, energy) {

super(color, energy);

this.sound = sound;

this.canFly = canFly;

}

makeSound() {

console.log(this.sound);

}

}

class HouseCat extends Cat {

constructor(houseCatSound = "meow", sound,canJumpHigh,canClimbTrees, color,energy) {

super(sound,canJumpHigh,canClimbTrees, color,energy);

this.houseCatSound = houseCatSound;

}

makeSound(option) {

if (option) {

super.makeSound();

}

console.log(this.houseCatSound);

}

}

class Tiger extends Cat {

constructor(tigerSound = "Roar!", sound,canJumpHigh,canClimbTrees, color,energy) {

super(sound,canJumpHigh,canClimbTrees, color,energy);

this.tigerSound = tigerSound;

}

makeSound(option) {

if (option) {

super.makeSound();

}

console.log(this.tigerSound);

}

}

class Parrot extends Bird {

constructor(canTalk = false, sound,canFly, color,energy) {

super(sound,canFly, color,energy);

this.canTalk = canTalk;

}

makeSound(option) {

if (option) {

super.makeSound();

}

if (this.canTalk) {

console.log("I'm a talking parrot!");

}

}

}

var fiji = new Parrot(false); // we're passing `false` to the constructor so that fiji can't talk

var polly = new Parrot(true); // we're passing `true` to the constructor so that polly can talk

fiji.makeSound(); // undefined

fiji.makeSound(true); // chirp

polly.makeSound(); // I'm a talking parrot!

polly.makeSound(true); // chirp, I'm a talking parrot!

polly.color; // yellow

polly.energy; // 100

polly.isActive(); // Energy is decreasing, currently at: 80

var penguin = new Bird("shriek", false, "black and white", 200); // setting all the custom properties

penguin; // Bird {color: 'black and white', energy: 200, sound: 'shriek', canFly: false }

penguin.sound; // 'shriek'

penguin.canFly; // false

penguin.color; // 'black and white'

penguin.energy; // 200

penguin.isActive(); // Energy is decreasing, currently at: 180

var leo = new HouseCat();

// leo, no purring please:

leo.makeSound(false); // meow

// leo, both purr and meow now:

leo.makeSound(true); // purr, meow

var cuddles = new Tiger();

cuddles.makeSound(false); // Roar!

cuddles.makeSound(true); // purr, Roar!

Additional Resources

Constructor

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/constructor>

Classes

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes>

Object-oriented programming

<https://css-tricks.com/the-flavors-of-object-oriented-programming-in-javascript/>

Regular Expressions in JavaScript

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions>

RegExp object in JavaScript

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/RegExp>