Return values from functions

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

An example is the **console.log()** function.

If I run:

1

console.log('Hello');





... here's the output in the console:

1

2

Hello

undefined





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

I'll now code my own implementation of **console.log()**, which doesn't return the value of **undefined**:

1

2

3

4

function consoleLog(val) {

    console.log(val)

    return val

}





I'm using the **console.log()** function inside my custom **consoleLog** function declaration. And I'm specifying it to return the value of its argument.

Now when I run my custom **consoleLog()** function:

1

consoleLog('Hello')





I get the following output:

1

2

Hello

'Hello'





So, the value is output in the console, but it's also returned.

Why is this useful?

It's useful because I can use return values from one function inside another function.

Here's an example.

I'll first code a function that returns a double of a number that it received:

1

2

3

function doubleIt(num) {

    return num \* 2

}





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

1

2

3

4

5

function objectMaker(val) {

    return {

        prop: val

    }

}





I can call the **objectMaker()** function with any value I like, such as:

1

objectMaker(20);





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

1

{prop:20}





Now consider this code:

1

doubleIt(10).toString()





The above code returns the number **20** as a string, that is: **"20"**.

I can even combine my custom function calls as follows:

1

objectMaker( doubleIt(100) );





This will now return the following value:

1

{prop: 200}





What does all of this mean?

It means that 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

## Learning Objectives

* Be able to explain that there are several programming paradigms
* Be able to explain the basic difference between the two predominant programming paradigms: functional programming and object-oriented programming
* Understand, at a very high level, how the functional programming paradigm works

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

In functional programming, we use a lot of functions and variables.

1

2

3

4

5

6

7

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.

1

2

3

4

5

6

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.

1

2

3

4

5

6

7

8

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:

1

2

3

4

var virtualPet = {

    sleepy: true,

    nap: function() {}

}





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

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

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

1

2

3

4

5

6

7

8

9

10

//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 us 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".

Note that the goal here is not to discuss OOP in depth; instead, I just want to show you 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 this point, we can say that the 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.

There are many more concepts and ideas in functional programming.

Here are some of the most important ones:

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

There are many other concepts and priciples in functional programming, but for now, let's stick to these three.

### First-class functions

It is often said that functions in JavaScript are “first-class citizens”. What does that mean?

It means that a function in JavaScript is just another value that we can:

* pass to other functions
* save in a variable
* return from other functions

In other words, a function in JavaScript is just a value - from this vantage point, almost no different then a string or a 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())





I start the program 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 returns 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**. This is where things get 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()** function'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 JavaScript are truly first-class citizens, which can be assigned to variable names and passed around just like I would pass around a string, a number, an object, etc.** Note: most of the code inside the **randomNum()** function declaration comes from a previous lesson, namely the lesson that discussed the Math object in JavaScript. This brings me 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 re-defining the **addTwoNums()** function so that it is a higher-order function:

1

2

3

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 returned from those invocations. For example:

1

2

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

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





### Pure functions and side-effects

Another concept of functional programming are pure functions.

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:

1

2

3

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, a **5**, and a **6**:

1

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 of 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, it's sufficient to know that this concept exists and that it is related to functional programming.

# Additional resources

Here is a list of resources that may be helpful as you continue your learning journey.

[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)

# Object Oriented Programming principles

In this reading, you'll learn about the benefits of object-oriented programming (OOP) and the OOP principles.

## The Benefits of OOP

There are many benefits to using the object-oriented programming (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 the code you write. OOP also:

* Allows you to write modular code,
* Makes your code more flexible and
* Makes your code reusable.

## The Principles of OOP

The four fundamental OOP principles are inheritance, encapsulation, abstraction and polymorphism. You'll learn about each of these principles in turn. The thing to remember about Objects is that they exist in a hierarchal structure. Meaning that 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.

1

2

3

4

5

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

var myDog = Object.create(Animal)

console.log (Animal)





RunReset

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

1

2

3

4

5

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

var myDog = new Animal()

console.log (Animal)





RunReset

This concept is explored within the next section on inheritance

### OOP Principles: Inheritance

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

In essence, it's a very simple concept. It works like this:

1. There is a base class of a "thing".
2. There is one or more sub-classes of "things" that inherit the properties of the base class (sometimes also referred to as the "super-class")
3. There might be some other sub-sub-classes of "things" that inherit from those classes in point 2.

Note that 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.

All of this might sound a bit "computer-sciency", so here's a more practical example:

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

Thus, in the above example, 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 relation between classes in JavaScript, I can use the **extends** keyword, as in **class B extends A**.

Here's an example of an inheritance hierarchy in JavaScript:

1

2

3

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

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

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





### OOP Principles: Encapsulation

In the simplest 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:

1

"abc".toUpperCase();





I don't really need 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 break 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.

It helps to think 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.

For now, it's enough to be aware of their existence in OOP.

### OOP Principles: Polymorphism

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

So, to understand what polymorphism is about, let's consider some 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 a door.

The above concepts can be coded in JavaScript as follows:

1

2

3

4

5

6

7

8

9

10

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:

1

bicycle.bell(); // "Get away, please"





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

1

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 declaration:

1

2

3

function ringTheBell(thing) {

    console.log(thing.bell())

}





Now I have declared a **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 the **bicycle** as its single argument, here's the output:

1

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





However, if I invoke the **ringTheBell()** function and pass it the **door** object, I'll get the following output:

1

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.

1

"abc".concat("def"); // 'abcdef'





I can also use the **concat()** method on two arrays. Here's the result:

1

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





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

1

2

["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 same 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:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

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 above code, trycreating some of your own classes. (hint : think about things you know from everyday life)

// create your classes here

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:

1

class Train {}





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**:

1

2

3

4

5

class Train {

    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**.

6

5

3

4

2

1

}

    }

        this.color = color;

        this.lightsOn = lightsOn;

    constructor(color, lightsOn) {

class Train {





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:

1

new Train()





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:

1

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





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

1

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.

1

2

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.

For example:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

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:

1

var train4 = new Train('red', false);





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

1

2

3

4

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:

1

2

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:

1

2

3

4

5

6

7

class HighSpeedTrain extends Train {

    constructor(passengers, highSpeedOn, color, lightsOn) {

        super(color, lightsOn);

        this.passengers = passengers;

        this.highSpeedOn = highSpeedOn;

    }

}





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.

1

2

3

4

5

6

7

8

9

10

11

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.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

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.');

    }

}





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

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

1

2

3

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, namely:

1

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.

1

2

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:

1

2

train5.toggleLights(); // undefined

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





Here's the entire completed code for this lesson:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

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);





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** trains:

1

2

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) //





RunReset

7-22

StationaryBike { position: 'right corner', gears: 8 }

Treadmill { position: 'left corner', modes: 5 }

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:

1

2

3

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:

1

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:

1

2

3

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:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

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 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.

1

2

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:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

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.

Mark as completed

Like

Dislike

Report an issue

Chat with us

# Designing an OO Program

In this reading, I will show you how to create classes in JavaScript, using all the concepts you've learned so far.

Specifically, I'm preparing to build the following inheritance hierarchy:

Animal / \ Cat Bird / \ \ HouseCat Tiger Parrot

There are two keywords that are essential for OOP with classes in JavaScript.

These keywords are **extends** and **super**.

The **extends** keyword allows me to inherit from an existing class.

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

class Animal {

    // ... class code here ...

}

Then I can code, for example, the **Cat** sub-class, like this:

class Cat extends Animal {

    // ... class code here ...

}

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

The **super** keyword allows me to "borrow" functionality from a super-class, in a sub-class. The exact dynamics of how this works will be covered later on in this lesson.

Now I can start thinking about how to implement my OOP class hierarchy.

Before I even begin, I need to think about 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 I've thought it through, I can build my classes.

So, my plan is as follows:

1. The **Animal** class' constructor will have two properties: **color** and **energy**

2. The **Animal** class' prototype will have three methods: **isActive()**, **sleep()**, and **getColor()**.

3. The **isActive()** method, whenever ran, 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 just console log the value in the **color** property.

5. The **Cat** class will inherit from **Animal**, with the additional **sound**, **canJumpHigh**, and **canClimbTrees** properties specific to the **Cat** class. It will also have its own **makeSound()** method.

6. The **Bird** class will also inherit from **Animal**, but is own specific properties will be quite different from **Cat**. Namely, the **Bird** class will have the **sound** and the **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** class' 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. Finally, the **Parrot** 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 have 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 my requirements inside comments of otherwise empty classes, I can start coding each class as per my specifications.

### Coding the Animal class

First, I'll code the base **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: If I didn't use the **super** keyword in our sub-classes, once I'd run the above code, I'd get the following 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 we've set up this entire inheritance structure, we can build various animal objects.

For example, I can build two parrots: one that can talk, and the other 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

Also, I can build a pet cat:

var leo = new HouseCat();

Now I can have **leo** purr:

// leo, no purring please:

leo.makeSound(false); // meow

// leo, both purr and meow now:

leo.makeSound(true); // purr, meow

Additionally, I can build a tiger:

var cuddles = new Tiger();

My **cuddles** tiger can purr and roar, or just roar:

cuddles.makeSound(false); // Roar!

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

Here's the complete code from this lesson, for easier copy-pasting:

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;

    }

}

# Additional resources

Here is a list of resources that may be helpful as you continue your learning journey.

[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)

# For of loops and objects

In this reading, you'll learn how the for of loop works conceptually.

To begin, it's important to know that a for of loop cannot work on an object directly, since **an object is not iterable**. For example:

1

2

3

4

5

6

7

8

const car = {

    speed: 100,

    color: "blue"

}

for(prop of car) {

    console.log(prop)

}





RunReset

Running the above code snippet will throw the following error:

1

Uncaught TypeError: car is not iterable





Contrary to objects, arrays are iterable. For example:

1

2

3

4

const colors = ['red','orange','yellow']

for (var color of colors) {

    console.log(color);

}





RunReset

This time, the output is as follows:

1

2

3

red

orange

yellow





Luckily, you can use the fact that a for of loop can be run on arrays to loop over objects.

But how?

Before you can properly answer this question, you first need to review three built-in methods: **Object.keys()**, **Object.values()**, and **Object.entries()**.

## Built-in methods

### The Object.keys() method

The **Object.keys()** method receives an object as its parameter. Remember, this object is **the object you want to loop over**. It's still too early to explain how you'll loop over the object itself; for now, focus on the returned array of properties when you call the **Object.keys()** method.

Here's an example of running the **Object.keys()** method on a brand new **car2** object:

1

2

3

4

5

const car2 = {

    speed: 200,

    color: "red"

}

console.log(Object.keys(car2)); // ['speed','color']





RunReset

So, when I run **Object.keys()** and pass it my **car2** object, **the returned value is an array of strings**, where each string is a property key of the properties contained in my **car2** object.

### The Object.values() method

Another useful method is **Object.values()**:

1

2

3

4

5

const car3 = {

    speed: 300,

    color: "yellow"

}

console.log(Object.values(car3)); // [300, 'yellow']





### The Object.entries() method

Finally, there's another useful method, **Object.entries()**, which returns an array listing both the keys and the values.

1

2

3

4

5

const car4 = {

    speed: 400,

    color: 'magenta'

}

console.log(Object.entries(car4));





What gets returned from the invocation of the **Object.entries()** method is the following:

1

[ ['speed', 400], ['color', 'magenta'] ]





This time, the values that get returned are 2-member arrays nested inside an array. In other words, you get an array of arrays, where each array item has two members, the first being a property's key, and the second being a property's value.

Effectively, it's as if you was listing all of a given object's properties, a bit like this:

1

2

3

4

5

[

    [propertyKey, propertyVal],

    [propertyKey, propertyVal],

    ...etc

]





To summarise, you learned that you can loop over arrays using the **for of** loop. You also learned that you can extract object's keys, values, or both, using the **Object.keys()**, **Object.values()** and **Object.entries()** syntax.

## Examples

You now have all the ingredients that you need to **loop over any object's own property keys and values**.

Here's a very simple example of doing just that:

1

2

3

4

5

6

7

8

9

10

var clothingItem = {

    price: 50,

    color: 'beige',

    material: 'cotton',

    season: 'autumn'

}

for( key of Object.keys(clothingItem) ) {

    console.log(keys, ":", clothingItem[key])

}





The trickiest part to understand in this syntax is probably the **clothingItem[key]**.

Luckily, this is not too hard to comprehend, especially since you've already covered the concept previously when you were learning **how to access an object's member using the brackets notation**.

Recall that you also learned how you can dynamically access a property name.

To revisit this concept and show a practical demo of how that works, let's code a function declaration that randomly assigns either the string **speed** or the string **color** to a variable name, and then build an object that has only two keys: a **speed** key and a **color** key.

After this setup, you will be able to dynamically access either one of those properties on a brand new **drone** object, using the brackets notation.

Here's the example's code:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

function testBracketsDynamicAccess() {

  var dynamicKey;

  if(Math.random() > 0.5) {

    dynamicKey = "speed";

   }else{

     dynamicKey = "color";

   }

    var drone = {

      speed: 15,

      color: "orange"

    }

    console.log(drone[dynamicKey]);

}

testBracketsDynamicAccess();





RunReset

This example might feel a bit convoluted, but its purpose is to demo the fact that you're getting either one or the other value from an object's key, based on the string that got assigned to the **dynamicKey** variable, and accessed without issues, using the brackets notation.

Feel free to run the **testBracketsDynamicAccess()** function a few times, and you'll notice that sometimes the value that gets output is **15**, and sometimes it's **orange**, although I'm always accessing the **drone[dynamicKey]** key. Since the value of the **dynamicKey** is populated on the **Math.random()** invocation, sometimes that expression evaluates to **drone["speed"]**, and other times that expression evaluates to **drone["color"]**.

You have now learned about the building blocks that make the for of loop useful to iterate over objects - although objects are not iterables.

Next, you'll have a go at a practical example of working with both the for of and the for in loop. Each loops have their place and can be considered useful in different situations.

# Template literals examples

The aim of this reading is to help you understand how template literals work.

## What are template literals?

Template literals are an alternative way of working with strings, which was introduced in the ES6 addition to the JavaScript language.

Up until ES6, the only way to build strings in JavaScript was to delimit them in either single quotes or double quotes:

1

2

'Hello, World!'

"Hello, World!"





Alongside the previous ways to build strings, ES6 introduced the use of backtick characters as delimiters:

1

`Hello, World!`





The above code snippet is an example of a template string, which is also known as a template literal.

Note: On most keyboards, the backtick character can be located above the TAB key, to the left of the number 1 key.

## Differences between a template and regular string

There are several ways in which a template string is different from a regular string.

* First, it allows for **variable interpolation**:

1

2

3

let greet = "Hello";

let place = "World";

console.log(`${greet} ${place} !`) //display both variables using template literals

The above console log will output:

Hello World !

Essentially, using template literals allows programmers to embed variables directly in between the backticks, without the need to use the **+** operator and the single or double quotes to delimit string literals from variables. In other words, in ES5, the above example would have to be written as follows:

var greet = "Hello";

var place = "World";

console.log(greet + " " + place + "!"); //display both variables without using template literals

* Besides variable interpolation, template strings can span multiple lines.

For example, this is perfectly good syntax:

`Hello,

World

!`

Notice that this can't be done using **string literals** (that is, strings delimited in single or double quotes):

"Hello,

World"

The above code, when run, will throw a syntax error.

Put simply, template literals allow for multi-line strings - something that simply isn't possible with string literals.

* Additionally, the reason why it's possible to interpolate variables in template literals is because this syntax actually allows for **expression evaluation**.

In other words, this:

//it's possible to perform arithmetic operation inside a template literal expression

console.log(`${1 + 1 + 1 + 1 + 1} stars!`)

The above example will console log the following string: **5 stars!**.

This opens up a host of possibilities. For example, it's possible to evaluate a ternary expression inside a template literal.

Some additional use cases of template literals are **nested template literals** and **tagged templates**. However, they are a bit more involved and are beyond the scope of this reading.

If you're curious about how they work, please refer to the additional reading provided at the end of this lesson.

# Data Structures examples

In this reading, you'll learn about some of the most common examples of data structures.

The focus will be on working with the Object, Array, Map and Set data structures in JavaScript, through a series of examples.

Note that this reading will not include a discussion of some data structures that exist in some other languages, such as Queues or Linked Lists. Although these data structures (and other data structures too!) can be custom-coded in JavaScript, the advanced nature of these topics and the difficulty with implementing related exercises means they are beyond the scope of this reading.

## Working with arrays in JavaScript

Previously, you've covered a lot of concepts related to how to work with JavaScript arrays.

However, there are still a few important topics that can be covered, and one of those is, for example, working with some built-in methods.

In this reading, the focus is on three specific methods that exist on arrays:

1. **forEach**
2. **filter**
3. **map**

Let's explore these methods.

### The forEach() method

Arrays in JavaScript come with a handy method that allows you to loop over each of their members.

Here's the basic syntax:

1

2

3

4

5

const fruits = ['kiwi','mango','apple','pear'];

function appendIndex(fruit, index) {

    console.log(`${index}. ${fruit}`)

}

fruits.forEach(appendIndex);





The result of running the above code is this:

1

2

3

4

0. kiwi

1. mango

2. apple

3. pear





To explain the syntax, the **forEach()** method accepts **a function that will work on each array item**. That function's first parameter is the current array item itself, and the second (optional) parameter is the index.

Very often, the function that the **forEach()** method needs to use is passed in directly into the method call, like this:

1

2

3

4

const veggies = ['onion', 'garlic', 'potato'];

veggies.forEach( function(veggie, index) {

    console.log(`${index}. ${fruit}`);

});





This makes for more compact code, but perhaps somewhat harder to read. To increase readability, sometimes arrow functions are used. You can find out more about arrow functions in the additional reading.

### The filter() method

Another very useful method on the array is the **filter()** method. It filters your arrays **based on a specific test**. Those array items that pass the test are returned.

Here's an example:

1

2

3

4

const nums = [0,10,20,30,40,50];

nums.filter( function(num) {

    return num > 20;

})





Here's the returned array value:

1

[30,40,50]





Similar to the **forEach()** method, the **filter()** method also accepts a function and that function performs some work on each of the items in the array.

### The map method

Finally, there's a very useful **map** method.

This method is used to map each array item over to another array's item, based on whatever work is performed inside the function that is passed-in to the map as a parameter.

For example:

1

2

3

[0,10,20,30,40,50].map( function(num) {

    return num / 10

})





The return value from the above code is:

1

[0,1,2,3,4,5]





As already discussed, choosing a proper data structure affects the very code that you can write. This is because the data structure itself comes with some built-in functionality that makes it easier to perform certain tasks or makes it harder or even impossible without converting your code to a proper data structure.

Now that you've covered the methods, let's explore how to work with different built-in data structures in JavaScript.

## Working with Objects in JavaScript

A lot of the information on how to work with objects in JavaScript has already been covered in this course.

The example below demonstrates how to use the object data structure to complete a specific task. This task is to convert an object to an array:

1

2

3

4

5

6

7

8

9

10

const result = [];

const drone = {

    speed: 100,

    color: 'yellow'

}

const droneKeys = Object.keys(drone);

droneKeys.forEach( function(key) {

    result.push(key, drone[key])

})

console.log(result)





This is the result of executing the above code:

1

['speed',100,'color','yellow']





Although this is possible and works, having to do something like this might mean that you haven't chosen the correct data structure to work with in your code.

On the flip side, sometimes you don't get to pick the data structure you're working with. Perhaps that data comes in from a third-party data provider and all you can do is code your program so that it consumes it. You'll learn more about the interchange of data on the web when you learn about JSON (JavaScript Object Notation).

## Working with Maps in JavaScript

To make a new Map, you can use the **Map** constructor:

1

new Map();





A map can feel very similar to an object in JS.

However, it doesn't have inheritance. No prototypes! This makes it useful as a data storage.

For example:

1

2

3

4

5

6

let bestBoxers = new Map();

bestBoxers.set(1, "The Champion");

bestBoxers.set(2, "The Runner-up");

bestBoxers.set(3, "The third place");

console.log(bestBoxers);





Here's the console output:

1

Map(3) {1 => 'The Champion', 2 => 'The Runner-up', 3 => 'The third place'}





To get a specific value, you need to use the **get()** method. For example:

1

bestBoxers.get(1); // 'The Champion'





## Working with Sets in JavaScript

A set is a collection of unique values.

To build a new set, you can use the **Set** constructor:

1

new Set();





The **Set** constructor can, for example, accept an array.

This means that we can use it to quickly filter an array for unique members.

1

2

3

const repetitiveFruits = ['apple','pear','apple','pear','plum', 'apple'];

const uniqueFruits = new Set(repetitiveFruits);

console.log(uniqueFruits);





The above code outputs the following in the console:

1

{'apple', 'pear', 'plum'}





## Other data structures in JavaScript

Besides the built-in data structures in JavaScript, it's possible to build non-native, custom data structures.

These data structures come built-in natively in some other programming languages or even those other programming languages don't support them natively.

Some more advanced data structures that have not been covered include:

* Queues
* Linked lists (singly-linked and doubly-linked)
* Trees
* Graphs

For resources on building these data structures, please refer to the additional reading.

# Using Spread and Rest

In this reading, you'll learn how to join arrays, objects using the rest operator. You will also discover how to use the spread operator to:

* Add new members to arrays without using the **push()** method,
* Convert a string to an array and
* Copy either an object or an array into a separate object

#### Recall that the *push()* and *pop()* methods are used to add and remove items from the end of an array.

## Join arrays, objects using the rest operator

Using the spread operator, it's easy to concatenate arrays:

1

2

3

4

const fruits = ['apple', 'pear', 'plum']

const berries = ['blueberry', 'strawberry']

const fruitsAndBerries = [...fruits, ...berries] // concatenate

console.log(fruitsAndBerries); // outputs a single array





Here's the result:

1

['apple', 'pear', 'plum', 'blueberry', 'strawberry']





It's also easy to join objects:

1

2

3

4

const flying = { wings: 2 }

const car = { wheels: 4 }

const flyingCar = {...flying, ...car}

console.log(flyingCar) // {wings: 2, wheels: 4}





## Add new members to arrays without using the push() method

Here's how to use the spread operator to easily add one or more members to an existing array:

1

2

3

4

let veggies = ['onion', 'parsley'];

veggies = [...veggies, 'carrot', 'beetroot'];

console.log(veggies);





Here's the output:

1

['onion', 'parsley', 'carrot', 'beetroot']





## Convert a string to an array using the spread operator

Given a string, it's easy to spread it out into separate array items:

1

2

3

const greeting = "Hello";

const arrayOfChars = [...greeting];

console.log(arrayOfChars); //  ['H', 'e', 'l', 'l', 'o']





## Copy either an object or an array into a separate one

Here's how to copy an object into a completely separate object, using the spread operator.

1

2

3

4

5

6

7

8

9

const car1 = {

    speed: 200,

    color: 'yellow'

}

const car 2 = {...car1}

car1.speed = 201

console.log(car1.speed, car2.speed)





The output is **201, 200**.

You can copy an array into a completely separate array, also using the spread operator, like this:

1

2

3

4

const fruits1 = ['apples', 'pears']

const fruits2 = [...fruits]

fruits1.pop()

console.log(fruits1, "not", fruits2)





This time, the output is:

['apples'] 'not' ['apples','pears']

Note that the spread operator only performs a shallow copy of the source array or object. For more information on this, please refer to the additional reading.

There are many more tricks that you can perform with the spread operator. Some of them are really handy when you start working with a library such as React.

# Additional resources

Here is a list of resources that may be helpful as you continue your learning journey.

[Template literals](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals)

[Arrow functions](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions)

[Spread syntax](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax)

[Rest parameters](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/rest_parameters)

[JavaScript data structures](https://data-flair.training/blogs/javascript-data-structures/)

JavaScript interactivity

The purpose of this reading is to introduce you to a simple explanation of web page manipulation and some examples of it.

Did you know that JavaScript's initial purpose was to **provide interactivity in the browser?**

In other words, it was the "set of controls" that would allow web developers to control the behavior of the webpages and even the browsers that these webpages worked on.

This is still the case today.

As the web ecosystem developed and the world became ever more digitized, so did the myriad of ways that one can use JavaScript as a web developer to manipulate websites.

Initially, in the late 1990s, there was plain JavaScript that had to be tweaked to suit individual browsers.

Then, by the mid-2000s, the jQuery library came out, with the idea of writing less code, but doing more with it. It "leveled the playing field" as it allowed developers to use a single code-base for various browsers.

This trend continued and many other frameworks such as React, Vue, Angular, D3, and more came along.

Together with npm and Node.js, the JavaScript ecosystem is not slowing down.

Even though it has gone a long way, sometimes it's good to go back to basics. JavaScript is still the king when it comes to making our websites interactive.

Although CSS has developed significantly over the years, it is still JavaScript that allows users to:

* Get their geolocation,
* Interact with maps,
* Play games in the browser,
* Handle all kinds of user-triggered events, regardless of the device,
* Verify form input before sending it to the backend of a webapp,
* and more!

There are many, many ways in which JavaScript allows you to build rich, interactive experiences on the web.

Exercise: Web page content update

In this reading, you will learn how to capture user input and process it. You'll be introduced to a simple example that demonstrates how to manipulate information displayed based on user input.

To capture user input, you can use the built-in **prompt()** method, like this:

1

let answer = prompt('What is your name?');





Once you have the user-provided input inside the **answer** variable, you can manipulate it any way you need to.

For example, you can output the typed-in information on the screen, as an **<h1>** HTML element.

Here's how you'd do that:

1

2

3

4

5

6

7

let answer = prompt('What is your name?');

if (typeof(answer) === 'string') {

    var h1 = document.createElement('h1')

    h1.innerText = answer;

    document.body.innerText = '';

    document.body.appendChild(h1);

}





This is probably the quickest and easiest way to capture user input on a website, but doing it this way is not the most efficient approach, especially in more complex scenarios.

This is where HTML forms come in.

You can code a script which will take an input from an HTML form and display the text that a user types in on the screen.

Here's how this is done.

You'll begin by coding out a "test solution" to the task at hand:

1

2

3

4

5

6

7

8

9

var h1 = document.createElement('h1')

h1.innerText = "Type into the input to make this text change"

var input = document.createElement('input')

input.setAttribute('type', 'text')

document.body.innerText = '';

document.body.appendChild(h1);

document.body.appendChild(input);





So, you're essentially doing the same thing that you did before, only this time you're also dynamically adding the **input** element, and you're setting its HTML **type** attribute to **text**. That way, when you start typing into it, the letters will be showing in the **h1** element above.

However, you're not there quite yet. At this point, the code above, when run on a live website, will add the **h1** element with the text "Type into the input to make this text change", and an empty input form field under it.

You can try this code out yourself, for example, by pointing your browser to the **example.com** website, and running the above code in the console.

**Remember you can access the console from the developer tools in your browser.**

Another opinionated thing that you did in the code above is: setting my variables using the **var** keyword.

Although it's better to use either **let** or **const**, you're just running a quick experiment on a live website, and you want to use the most lenient variable keyword, the one which will not complain about you having already set the **h1** or the **input** variables.

If you had a complete project with a modern JavaScript tooling setup, you'd be using **let** or **const**, but this is just a quick demo, so using **var** in this case is ok.

The next thing that you need to do is: set up an event listener. The event you're listening for is the **change** event. In this case, the change event will fire after you've typed into the input and pressed the ENTER key.

Here's your updated code:

1

2

3

4

5

6

7

8

9

10

11

12

13

var h1 = document.createElement('h1')

h1.innerText = "Type into the input to make this text change"

var input = document.createElement('input')

input.setAttribute('type', 'text')

document.body.innerText = '';

document.body.appendChild(h1);

document.body.appendChild(input);

input.addEventListener('change', function() {

    console.log(input.value)

})





This time, when you run the above code on the said **example.com** website, subsequently typing some text into the input field and pressing the enter key, you'll get the value of the typed-in text logged to the console.

Now, the only thing that you still need to do to complete my code is to update the text content of the **h1** element with the value you got from the **input** field.

Here's the complete, updated code:

1

2

3

4

5

6

7

8

9

10

11

12

13

var h1 = document.createElement('h1')

h1.innerText = "Type into the input to make this text change"

var input = document.createElement('input')

input.setAttribute('type', 'text')

document.body.innerText = '';

document.body.appendChild(h1);

document.body.appendChild(input);

input.addEventListener('change', function() {

    h1.innerText = input.value

})





After this update, whatever you type into the input, after pressing the ENTER key, will be shown as the text inside the **h1** element.

Although this completes this lesson item, it's important to note that combining DOM manipulation and event handling allows for some truly remarkable interactive websites.

# Exercise: Capture Data

## Description

The aim of this exercise is to access the content of an element, specifically to use a button click to replace text.

## Task 1: The example.com website

Open the [example.com](https://exampledomain.github.io/capture-data/index.html) website in your browser. Open the developer tools and focus on the Console tab.

Example.com is a domain that can be used as an example in documents, papers and websites.

If you navigate in your browser to [http://www.example.com](http://www.example.com/)  you will see a webpage with a simple message:

**Example Domain**

This domain is established to be used for illustrative examples in documents. You may use this domain in examples without prior coordination or asking for permission.

## Task 2: Get h1 into a variable

Use the **document.querySelector()** method to query the h1 element on the page and assign it to the variable named **h1**.

## Task 3: Code an array

Declare a new variable, name it **arr**, and save the following array into it:

1

2

3

4

5

6

[

    'Example Domain',

    'First Click',

    'Second Click',

    'Third Click'

]





RunReset

## Task 4: Write a click-handling function

Write a new function declaration, named **handleClicks**. It should not accept any parameters.

Inside of it, code a **switch** statement, and pass a single parameter to it, **h1.innerText**.

The body of the switch statement should have a total of 4 cases (the fourth being the default case).

The first case should start with **case arr[0]:**. It should set the **h1.innerText** to **arr[1]**. In other words, it should assign the value of **arr[1]** to the **h1.innerText** property. The next line should have only the **break** keyword.

The second case should start with case **arr[1]**:. It should set the **h1.innerText** to **arr[2]**. In other words, it should assign the value of **arr[2]** to the **h1.innerText** property. The next line should have only the **break** keyword.

The third case should start with case **arr[2]**:. It should set the **h1.innerText** to **arr[3]**. In other words, it should assign the value of **arr[3]** to the **h1.innerText** property. The next line should have only the **break** keyword.

The **default** case should set the value of the **h1.innerText** property to **arr[0]**.

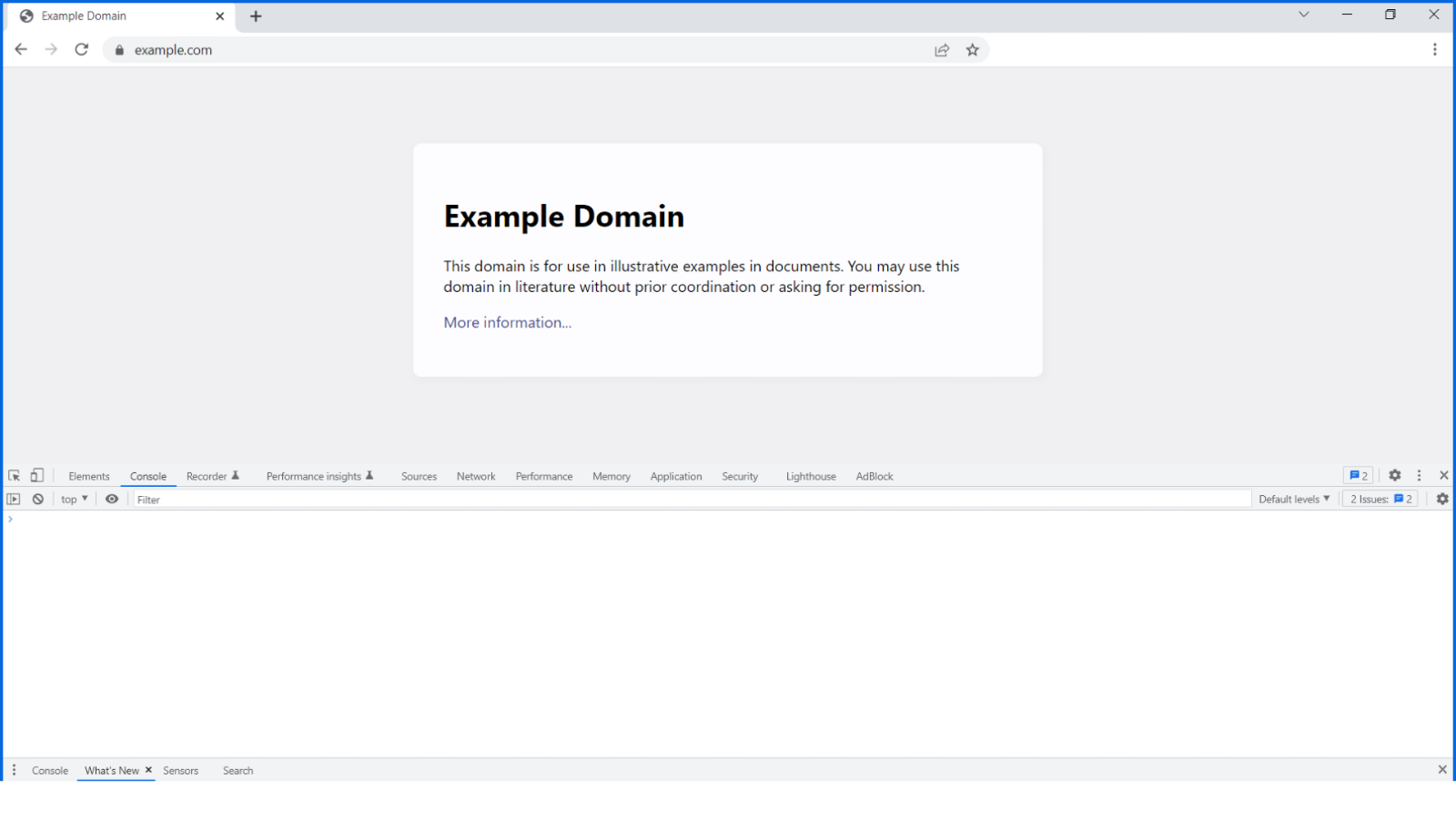
## Task 5: Add an event listener

You've created an **h1** variable in Task 2. Now, use that variable to run the **addEventListener()** method on it. Pass two arguments to the **addEventListener()** method: **'click'** and **handleClicks**.

# Solution: Capture Data

## Task 1 solution: The example.com website

1. Open your favorite browser and navigate to <https://example.com/>.
2. Next open its developer tools using either the F12 key, or right-clicking onto the page and selecting the contextual menu's Inspect command, or by pressing CTRL SHIFT I or COMMAND SHIFT I.
3. Next, click on the Console tab to open it in a dedicated tab, or press the ESC key to have the console open while any tab is in focus.



## Task 2 solution: Get h1 into a variable

1

2

var h1 = document.querySelector('h1')





RunReset

## Task 3 solution: Code an array

1

2

3

4

5

6

7

var arr = [

    'Example Domain',

    'First Click',

    'Second Click',

    'Third Click'

]





RunReset

## Task 4 solution: Write a click-handling function

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

function handleClicks() {

    switch(h1.innerText) {

        case arr[0]:

            h1.innerText = arr[1]

            break

        case arr[1]:

            h1.innerText = arr[2]

            break

        case arr[2]:

            h1.innerText = arr[3]

            break

        default:

            h1.innerText = arr[0]

    }

}





RunReset

## Task 5 solution: Add an event listener

h1.addEventListener('click', handleClicks);

# Moving data around on the web

The modern web consists of millions and millions of web pages, connected services and databases.

There are websites communicating with other websites, getting data from data feeds and data providers, both paid and free.

All of these data flows must be facilitated with some kind of data format.

Around 2001, Douglas Crockford came up with a data interchange format based on JavaScript objects. The name given to this format was JSON, which is JavaScript Object Notation.

Before JSON, the most common data interchange file format was **XML** (Extensible Markup Language). However, due to XML's syntax, it required more characters to describe the data that was sent. Also, since it was a specific stand-alone language, it wasn't as easily inter-operable with JavaScript.

Thus, the two major reasons for the JSON format becoming the dominant data interchange format that it is today is two-fold:

* First, it's very lightweight, with syntax very similar to "a stringified JavaScript object". You'll learn more about the specifics of this later.
* Second, it's easier to handle in JavaScript code, since, JSON, after all, is just JavaScript.

It is often said that JSON is a subset of JavaScript, meaning it adheres to syntax rules of the JavaScript language, but it's even more strict in how proper JSON code should be formatted. In other words, all JSON code is JavaScript, but not all JavaScript code is JSON.

Besides being a data interchange format, JSON is also a file format. It's not uncommon to access some third-party data from a third-party website into our own code in the form of a **json** file.

For example, if you had a website with the data on stock price movements, you might want to get the data of the current stock prices from a data vendor. They might offer their data service by giving you access to the file named, say **stockPrices.json**, that you could access from their servers.

Once you'd downloaded that stringified JSON data into your own code, you could then convert that data to a plain JavaScript object.

That would mean that you could use your web application's code to "dig into" the third-party data-converted-to-a-JavaScript-object, so as to obtain specific information based on a given set of criteria.

For example, if the stringified JSON data was converted to an object that had the following structure:

1

2

3

4

5

6

7

8

9

10

11

12

13

const currencyInfo = {

    [

        USD: {

            // ...

        },

        GBP: {

            // ...

        },

        EUR: {

            // ...

        }

    ]

}





 You could then access only the data on the **USD** property, if that's what was needed by you app at a given point in time.

Hopefully, with this explanation, you understand, at a high level, how and why you might want to use JSON in your own code.

It's all about getting stringified JSON data from a server, converting ("parsing") that data into JS objects in your own code, working with the converted data in your own code, and perhaps even stringifying the result into JSON, so that this data is then ready to, for example, be sent back to a server after your code has processed it locally.

## JSON is just a string - but there are rules that it must follow

JSON is a string, but it must be a properly-formatted string. In other words, it must adhere to some rules.

If a JSON string is not properly formatted, JavaScript would not be able to parse it into a JavaScript object.

JSON can work with some primitives and some complex data types, as described below.

Only a subset of values in JavaScript can be properly stringified to JSON and parsed from a JavaScript object into a JSON string.

These values include:

* primitive values: strings, numbers, bolleans, null
* complex values: objects and arrays (no functions!)
* Objects have double-quoted strings for all keys
* Properties are comma-delimited both in JSON objects and in JSON arrays, just like in regular JavaScript code
* String properties must be surrounded in double quotes. For example:

**"fruits"**,

**"vegetables"**

* Number properties are represented using the regular JavaScript number syntax; e.g

**5**,

**10**,

**1.2**

* Boolean properties are represented using the regular JavaScript boolean syntax, that is:

**true**

and

**false**

* Null as a property is the same as in regular JavaScript; it's just a

**null**

You can use object literals and array literals, as long as you follow the above rules

What happens if you try to stringify a data type which is not accepted in JSON syntax?

For example, what if you try to stringify a function? **The operation will silently fail**.

If you try to stringify some other data types, such as a BigInt number, say **123n**, you'd get the following error: **Uncaught TypeError: Do not know how to serialize a BigInt**.

### Some examples of JSON strings

Finally, here is an example of a stringified JSON object, with a single key-value pair:

**'{"color":"red"}'**

Here's a bit more complex JSON object:

**'{"color":"red", "nestedObject": { "color": "blue" } }'**

The above JSON object encodes two properties:

* **"color":"red"**
* **"nestedObject": { "color": "blue" }**

It's also possible to have a JSON string encoding just an array:

**'["one", "two", "three"]'**

The above JSON string encodes an array holding three items, three values of the string data type. Obviously, just like objects, arrays can nest other simple or complex data structures.

For example:

**'[{ "color": "blue" }, {"color: "red"}]'**

In the above example, the JSON string encodes an array which holds two objects, where each object consists of a single key-value pair, where both values are strings.

# Additional resources

Here is a list of resources that may be helpful as you continue your learning journey.

[MDN: Modules](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules)

[Nodejs.org official docs on CommonJS](https://nodejs.org/api/modules.html#modules-commonjs-modules)

[MDN: DOM](https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model)

[MDN: Document.querySelector](https://developer.mozilla.org/en-US/docs/Web/API/Document/querySelector)

[MDN: Event](https://developer.mozilla.org/en-US/docs/Web/API/Event)

[MDN: EventTarget.addEventListener](https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/addEventListener)

[MDN: Working with JSON](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/JSON)