JavaScript Additional

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator_precedence

https://developer.mozilla.org/en-US/docs/Glossary/Primitive

https://tc39.es/ecma262/

https://jquery.com/

https://react.dev/

https://survey.stackoverflow.co/2021#technology-most-popular-technologies

https://unicode.org/emoji/charts/full-emoji-list.html#1f600

---

https://developer.mozilla.org/en-US/docs/Glossary/Truthy

https://developer.mozilla.org/en-US/docs/Glossary/Falsy

https://www.javascripttutorial.net/javascript-comparison-operators/

https://developer.mozilla.org/en-US/docs/Learn_web_development/Core/Scripting/Conditionals

---

Math object cheat sheet

JavaScript has handy built-in objects. One of these popular built-in objects is the Math object. By the end of this reading, you'll be able to:

- Outline the built-in properties and methods of the Math object

Number constants

Here are some of the built-in number constants that exist on the Math object:

- The PI number: **`Math.PI`** which is approximately 3.14159
- The Euler's constant: **`Math.E`** which is approximately 2.718

- The natural logarithm of 2: `Math.LN2` which is approximately 0.693

Rounding methods

These include:

- `Math.ceil()` - rounds up to the closest integer
- `Math.floor()` - rounds down to the closest integer
- `Math.round()` - rounds up to the closest integer if the decimal is `.5` or above; otherwise, rounds down to the closest integer
- `Math.trunc()` - trims the decimal, leaving only the integer

Arithmetic and calculus methods

Here is a non-conclusive list of some common arithmetic and calculus methods that exist on the `Math` object:

- `Math.pow(2,3)` - calculates the number `2` to the power of `3`, the result is `8`
- `Math.sqrt(16)` - calculates the square root of `16`, the result is `4`
- `Math.cbrt(8)` - finds the cube root of `8`, the result is `2`
- `Math.abs(-10)` - returns the absolute value, the result is `10`
- Logarithmic methods: `Math.log()`, `Math.log2()`, `Math.log10()`
- Return the minimum and maximum values of all the inputs: `Math.min(9,8,7)` returns `7`, `Math.max(9,8,7)` returns `9`.
- Trigonometric methods: `Math.sin()`, `Math.cos()`, `Math.tan()`, etc.

String cheat sheet

By the end of this reading, you'll be able to:

- Identify examples of String functions and explain how to call them

In this cheat sheet, I'll list some of the most common and most useful properties and methods available on strings.

For all the examples, I'll be using either one or both of the following variables:

```
2
var place = "World"
```

Note that whatever string properties and methods I demo in the following examples, I could have ran it on those strings directly, without saving them to a variable such as the ones I named `greet` and `place`.

In some of the examples that follow, for the sake of clarity, instead of using a variable name, I'll use the string itself.

All strings have at their disposal several built-in properties, but there's a single property that

is really useful: the **`length`** property, which is used like this:

```
greet.length; // 7
```

To read each individual character at a specific index in a string, starting from zero, I can use the **`charAt()`** method:

```
greet.charAt(0); // 'H'
```

The **`concat()`** method joins two strings:

```
"Wo".concat("rl").concat("d"); // 'World'
```

The **`indexOf`** returns the location of the first position that matches a character:

```
"ho-ho-ho".indexOf('h'); // 0
"ho-ho-ho".indexOf('o'); // 1
"ho-ho-ho".indexOf('-'); // 2
```

The **`lastIndexOf`** finds the last match, otherwise it works the same as **`indexOf`**.
The **`split`** method chops up the string into an array of sub-strings:

```
"ho-ho-ho".split("-"); // ['ho', 'ho', 'ho']
```

There are also some methods to change the casing of strings. For example:

```
greet.toUpperCase(); // "HELLO, "
greet.toLowerCase(); // "hello, "
```

Here's a list of all the methods covered in this cheat sheet:

- **`charAt()`**
- **`concat()`**
- **`indexOf()`**
- **`lastIndexOf()`**

- `split()`
- `toUpperCase()`
- `toLowerCase()`

---

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

https://developer.mozilla.org/en-US/docs/Learn_web_development/Core/Scripting/Object_basics

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/typeof

https://developer.mozilla.org/en-US/docs/Learn_web_development/Core/Scripting/Arrays

---

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/function

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/try...catch

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration_protocols

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Errors

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/null

https://developer.mozilla.org/en-

US/docs/Web/JavaScript/Reference/Global_Objects/undefined

---

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

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

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

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

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:

```
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:

```
"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:

```
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 declaration:

```
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:

```
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:

```
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. Here's the result:

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

Consider using 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 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:

```
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
```

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 function, I 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 get back is the current datetime, such as:

`Thu Feb 03 2022 11:24:08 GMT+0100 (Central European Standard Time)`

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

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

Thus, I 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 an 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`);
    }
}
```

RunReset
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}
```

RunReset

I've just built two instance objects of **`Icecream`** type.
The most common use case of **`new`** is to use it with one of the built-in 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 build 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 the **`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 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 built 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/); // ['d', index: 3, input: 'abcd', groups: undefined]
"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-in types, we can use the constructor.
Here are a few examples:

```
new Date();
new Error();
new Map();
new Promise();
new Set();
new WeakSet();
new WeakMap();
```

The above list is inconclusive, but it's just there to give you an idea of some constructor functions you can surely use.
Note that there are links provided about RegExp and regular expression in the lesson item titled *"Additional Reading"*.

Creating classes
By the end of this reading, you should be able to explain, with examples, the concept of extending classes using basic inheritance to alter behaviors within child classes.
By now, you should know that inheritance in JavaScript is based around the prototype object. All objects that are built from the prototype share the same functionality.
When you need to code more complex OOP relationships, you can use the **`class`** keyword and its easy-to-understand and easy-to-reason-about syntax.
Imagine that you need to code a **`Train`** class.
Once you've coded this class, you'll be able to use the keyword **`new`** to instantiate objects of the **`Train`** class.

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

```
class Train {}
```

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**:

```
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`**.

```
        this.color = color;
        this.lightsOn = lightsOn;
    }
}
```

Notice the syntax of the constructor. The constructor is a special function in my **`Train`** class. First of all, notice that there is no **`function`** keyword. Also, notice that the keyword **`constructor`** is used to define this function. You give your **`constructor`** function parameters inside an opening and closing parenthesis, just like in regular functions. The names of parameters are **`color`** and **`lightsOn`**.
Next, inside the **`constructor`** function's body, you assigned the passed-in **`color`** parameter's value to **`this.color`**, and the passed-in **`lightsOn`** parameter's value to **`this.lightsOn`**.
What does this **`this`** keyword here represent?
It's the future object instance of the `Train` class.
Essentially, this is all the code that you need to write to achieve two things:

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

properties of **`color`** and **`lightsOn`**.

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

```
new Train()
```

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

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

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

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

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

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

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

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

However, this is not all that classes can offer.

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

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 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:

```
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:

```
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.

```
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.

```
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:

```
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.

```
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:

```
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:

```
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

In this example, there are three classes defined: **`StationaryBike`**, **`Treadmill`**, and **`Gym`**.
The **`StationaryBike`** class is coded so that its future object instance will have the **`position`** and **`gears`** properties. The **`position`** property describes where the stationary bike will be placed inside the gym, and the **`gears`** propery gives the number of gears that that stationary bike should have.
The **`Treadmill`** class also has a position, and another property, named **`modes`** (as in "exercise modes").
The **`Gym`** class has three parameters in its constructor function: **`openHrs`**, **`stationaryBikePos`**, **`treadmillPos`**.

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

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

Default Parameters
A useful a ES6 feature allows me to set a default parameter inside a function definition First, .
What that means is, I'll use an ES6 feature which allows me to set a default parameter inside a function definition, which goes hand in hand with the defensive coding approach, while requiring almost no effort to implement.
For example, consider a function declaration without default parameters set:

```
function noDefaultParams(number) {
    console.log('Result:', number * number)
}
```

Obviously, the **`noDefaultParams`** function should return whatever number it receives, *squared*.
However, what if I call it like this:

```
noDefaultParams(); // Result: NaN
```

JavaScript, due to its dynamic nature, doesn't throw an error, but it does return a non-sensical output.
Consider now, the following improvement, using default parameters:

```
}
```

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:

```
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.

```
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 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.

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.

```
                                                                    21
}
```

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:

```
22
}
```

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:

```
1

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:

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

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

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

---

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/constructor
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes
https://css-tricks.com/the-flavors-of-object-oriented-programming-in-javascript/

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

---

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:

```
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:

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

It's also easy to join objects:

```
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:

```
let veggies = ['onion', 'parsley'];
veggies = [...veggies, 'carrot', 'beetroot'];
```

```
console.log(veggies);
```

Here's the output:

```
['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:

```
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.

```
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:

```
const fruits1 = ['apples', 'pears']
const fruits2 = [...fruits1]
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.

---

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/rest_parameters
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures

---

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:

```
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:

```
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:


```
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:

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:

```
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.

---

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules
https://nodejs.org/api/modules.html#modules-commonjs-modules
https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model
https://developer.mozilla.org/en-US/docs/Web/API/Document/querySelector
https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/addEventListener
https://developer.mozilla.org/en-US/docs/Learn_web_development/Core/Scripting/JSON

---

https://developer.mozilla.org/en-US/docs/Learn_web_development/Extensions/Server-side
https://nodejs.org/api/documentation.html
https://jestjs.io/
https://www.cypress.io/

https://www.npmjs.com/
https://www.browserstack.com/guide/unit-testing-in-javascript
---