Object Oriented Programming in Javascript

Table of Contents

Module 1: Objects basics

What are objects in JavaScript?

You can model real world things and occurrences in JavaScript by using a concept called objects. This is more popularly known as object-oriented programming, where you try to solve real world problems with a programming language, using objects.

As I said, objects model real world conditions.

In the real world, there are 2 things: objects and properties.

For example, take us human beings.

We are human beings, as in one of the objects, and each object within this category, as in, each person, has a ton of properties with different value.

For example, let us say the object is called a person.

Now, the person can have properties like, name, age, height, hair color, etc. The properties can go on like that, without any end.

Also, there can be more than one person, but with different properties, or in some cases, the same properties.

Let me explain that by writing down.

So, let's say these are the properties of the first person:

Person1:

```
Name: Susan
Age: 35
Height: 160
Hair: blond
```

And let's say these are the properties of the 2nd person.

Person2:

```
Name: Gary
```

```
Age: 40
Height: 180
Hair: blond
```

So, as you can see here, they have the same properties, but the values differ in some cases, but remain the same in others. They both have blond hair.

So, this is how objects are written. In fact, person2, which is a different object, can in fact have more properties or different properties than the person1 object.

Let's add eye color to the 2nd object.

```
eyeColor: blue
```

So, this illustrates what objects are, and how their properties and values are derived.

How to create objects in JavaScript?

But how can we write this in JavaScript?

The syntax is quite simple.

The properties and values are called key:value pair, where the property name is called a key and it holds the reference to the value.

The property should always be a string, but the value can be anything: string, number, Boolean, another object, a function, array etc.

As usual, you need to declare a variable. Variables usually hold only one value, except in the case of arrays, where it can hold more than one value.

But, with objects as well, you can make a variable hold multiple property.

But before that, let's look at how to create empty objects. There are two ways to create them:

```
let person = {};
let person = new Object();
```

The first method is more commonly used, so let's stick to that. We just created an empty object so far. What if we need to insert properties and values into them? The easiest way to do that would be while creating the object.

So, for person1:

```
let person1 = { name: "Susan", age: 35, height: 160, hair:
"blond" };
```

For person2:

```
let person2 = { name: "Gary", age: 40, height: 180, hair:
"blond", eyeColor: "blue" };
```

As you can see here, you declare the variable first, and the variable gets converted to an object based on what is assigned to it, which is in this case property value pairs.

An object's values are created within two flower braces.

The property is written first, and with a color, the value for that property is written, just like in the case of a CSS stylesheet. This is a property value pair as well.

This is called encapsulation. Everything that is related to person 1 is encapsulated inside that object, in the form of properties and its values.

Access property values

Now, how do we access these values?

There are two ways to do this:

Using dot notation, or using square brackets, like you do in arrays.

```
objectName.propertyName or
objectName["propertyName"]
```

Let's try displaying the values on our live editor. Let's say we want to know that person1's height is. To do that, let's alert:

```
alert(person1.height);
```

If we run the above line of code, we get Susan's height, which is 160 cms.

Let's try doing the same with the 2nd method now.

```
alert(person1["height"]);
```

Add & update properties

You can add new properties to your objects too.

To do that, you can again use the dot notation or the square brackets.

```
person1.gender = "female";
console.log(person1);
```

You can give Boolean values for your properties as well.

```
person2["canVote"] = true;
console.log(person2);
```

Delete properties

You can delete the properties from the object as well, by using the pre-defined delete property of the Object.

```
delete person2.canVote;
console.log(person2);
```

Dynamic access

You can make accessing, adding or anything of a property in an object dynamic by declaring a variable and using that as the point of reference.

```
let prop = "hair";
console.log(person1[prop]);
person1[prop] = "brunette";
prop = "name";
console.log(person2[prop]);
```

You can get the value of prop from the user to make it truly dynamic, like this:

```
let prop = prompt("What do you want to see?","hair");
alert(`Person 1's ${prop} is ${person1[prop]}`);
```

This doesn't work with dot notation:

```
alert(`Person 1's ${prop} is ${person1.prop}`);
```

We get undefined because prop isn't an actual property, we need to replace the string inside it in the square brackets for it to work.

Muli-word properties

We can create multi-word properties within square brackets.

```
person1["can vote"] = true;
person1.can vote = true; //doesn't work
```

It thinks person1.can is the correct syntax, and vote will produce an error.

After the dot, you need a valid variable that adheres to variable rules: starts with $ or letters, contains $, letters, numbers or _, no spaces or other special characters, etc.

The following will produce an error too.

```
person1.%vote = true;
```

But square brackets always work with multi-word properties, to set properties, update them, delete them, etc. They should always be within quotes though, single, or double.

Shorthand

Look at the example below:

```
let fName = "John";
let lName = "Doe";
let age = 35;
```

```
let person1 = {
     fName: fName,
     lName: lname,
     age: age
};
```

Sometimes, the property name and their values are the same.

In that case, we can use shorthand to save some space, like this:

```
let person1 = {
     fName,
     lName,
     age
};
console.log(fName);
```

This works with functions too. Sometimes, we receive values as arguments inside a function and we return an object with them, like this:

```
function createPerson(fName, lName, age) {
     let fullName = `${fName} ${lName}`;
     return {
          fName: fName,
          lName: lName,
          age: age,
          fullName: fullName
     };
}
console.log(createPerson("John","Doe",35));
```

Instead, we can use the shorthand:

```
function createPerson(fName, lName, age) {
     let fullName = `${fName} ${lName}`;
     return {
          fName,
          lName,
          age,
          fullName
     };
}
```

And it will still work the same.

An object can have both shorthand as well as normal properties in them:

```
let person1 = {
     fName,
     lName,
     age,
     fullName: fName + " " + lName
};
```

Arrays as objects

You can also create an array of objects. As you know, an object holds the properties and values of a single object. What if you want to hold the properties of multiple objects? Arrays are your best bet.

Let's create a variable called just person this time. This variable should hold and array that has 2 object, one which has the properties we assigned to the person1 object and the other which has the properties we assigned to the person2 object.

```
let person = [{ name: "Susan", age: 35, height: 160, hair:
"blond" }, { name: "Gary", age: 40, height: 180, hair: "blond",
eyeColor: "blue" }];
```

So, all you must do is, create the object's property value pairs within a flower bracket, and separate them by commas.

Now, to access the values, use the same method as before, but this time with array indexes so your browser knows which object you are referring to.

So, let's say we want to get the eye color of the 2nd object.

Then, you must do it like this. The variable name, then the array index of that object, which in this case is 1, and follow that up with the dot and the property name, as usual.

```
person[1].eyeColor
```

Objects and const

Generally, anything that is declared with const cannot be modified, as you know, and that's why we declare them with all caps, to denote the difference.

```
const PI = 3.14;
PI = 4; //error
```

But with objects, its different. They can be modified.

```
'use strict';
const person1 = {
     lName: "John",
     fName: "Doe"
};
person1.lName = "Jane"; //No error
```

That's because only person1 as such is fixed, not the properties or methods inside it or their values.

But, if we try to redeclare person1, lets see what happens:

```
person1 = {
     name: "John Doe"
}; //Assignment to constant variable error.
```

Variables as properties – computed properties

We can set variables as properties, as such, when creating them. They are called computed properties because the property name can be dynamic, based on the user's input.

These properties should be specified within square brackets within the object declaration.

```
let person = "Jane Doe";
let voting = {
     [person] : "Can vote"
};
console.log(voting);
```

We can change the property to anything, and it'll affect the object.

The below line of code changes the object.

```
let person = "John Doe";
```

But, if we change the person value after the object declaration,

```
person = "John Murray";
console.log(voting);
```

In the above, the object is still the same because it took the value of the person above the declaration and that's it.

So, if you want a truly dynamic property name, you can get it from the user, with a prompt maybe, or an input box.

```
let person = prompt("What is the name of the person?","Jane
Doe");
let voting = {
     [person] : "Can vote"
};
console.log(`${person} ${voting[person]}`);
```

Accessing the property doesn't work with dot notation though.

We can compute the properties too.

```
let person = "Jane";
let voting = {
     [person + 1] : "Can vote"
};
console.log(voting["Jane1"]);
```

Naming properties – do's and don'ts

There are basically no restrictions with property naming. They can be anything, they don't need to even follow proper variable naming conventions.

The only difference is that, if they don't follow proper variable naming conventions, you can't use the dot notation to access said property, but you can always use the square brackets for anything.

Numbers and string values can be properties too. They'll automatically be converted to strings while assigning.

```
let voting = {
     [person + 1] : "Can vote",
     1: "Number One",
     $1Person: "Can vote",
     for: "Hello",
     "Likes food": "Yes"
};
```

Object methods

Can create methods just as we can create properties. They are basically functions that pertain to that particular object.

```
let person = {
     fName: "Jane",
     lName: "Doe",
     fullName: function() {
          return person.fName + " " + person.lName;
     }
};
console.log(person.fullName());
```

Why person.fName, why not just fName because it is inside the object after all. Let's try.

It says undefined. That's because fName as such wasn't declared, it's a part of the object.

```
fullName() {
     return person.fName + " " + person.lName;
}
```

The above works too because of the ES6 update. We don't have to rely on function expressions anymore.

We can create methods outside of the object too, but in that case, you need to do something like this:

```
person.fullName = function() {
     return person.fName + " " + person.lName;
}
```

But if we tried what we did before, we'll get an error.

```
person.fullName() {
     return person.fName + " " + person.lName;
}
```

So, we'll have to resort to function expressions in this case.

The function must be properly called with ().

If we run the below line of code,

```
console.log(person.fullName);
```
We'll just get the function definition.

We can assign functions that we've defined before to an object's method too.

```
let person = {
     fName: "Jane",
     lName: "Doe",
     fullName: fullName
};
function fullName() {
     return person.fName + " " + person.lName;
}
console.log(person.fullName());
```

Since it is a normal function definition and not a function expression, you can place it after you create an object as well and it'll work the same as placing it before the object.

"this" in methods

In the above examples, we accessed properties that were inside the objects for our method, so we used `person.propertyName` to access them. But its right inside, so why not use "this"?

```
let person = {
     fName: "Jane",
     lName: "Doe",
     fullName() {
          return this.fName + " " + this.lName;
     }
};
```
"this" in this case refers to the object within which the method resides.

It'll work the same if we create the method outside the object as well (show).

Why do we even bother with "this"? Why not just stick to `person.fName`?

Well, because referencing the object name as such is unreliable. What if we copy the object to another variable? Create a copy of it? Then if we try to reference the method, we won't get what we want.

```
let person1 = person;
```

This creates a reference to the object in person1 too. I'll explain about object referencing in a future lesson.

```
console.log(person1.fullName());
console.log(person1);
```

The above works, but what if we change person?

What if we make person null? Technically, person1 should still work because it is still referencing the original object and still has everything, right?

Let's check.

Ok, person1 is still there.

But if we try to access the function, what happens?

```
console.log(person1.fullName());
```

We're getting an error: "Cannot read fName of Null".

But if it were "this", we wouldn't have gotten this problem.

```
return this.fName + " " + this.lName;
```

So now it works.

This is the main advantage of using "this" over the object name in methods like this. It makes the methods more dynamic and not bound to one object name.

It'll only work if you use function expressions. "this" doesn't work with arrow functions.

"this" refers to whatever object calls the function. When we later learn about constructors and creating multiple objects out of the same constructor, you'll understand how truly dynamic "this" is.

In operator and for…in

You can know if a key is a part of the object by using the "in" keyword.

Is lName a key of the object "person"?

```
console.log("lName" in person);
console.log("name" in person); //false
```

In objects, if you try to access a property that's not in an object, you'll just get undefined, and not an error (show). That's why it shows as true and false in here.

Realistically, you can do this as well:

```
console.log(person.name == undefined);
```

But what if name was assigned undefined in the object. Then we would not know for sure if the object has that property or not. That's why we're better off using "in".

That said, in can be used to parse through all the keys in the object in a for loop, by using the for…in loop.

```
for(let prop in person) {
     console.log(person[prop]);
}
```

It shows the values of both the properties and methods.

Module 2: Objects – Constructors, getters, setters and more

Object referencing

Unlike other variables, objects are not stored in the variables they are created in. When you assign an object to a variable, that variable just holds the memory address of where the object resides. That is, it stores the reference to that object and not the object itself.

This is called object referencing and it saves a lot of memory space, when you think about it. Let me show you how.

```
let str = "Hello World";
let str1 = str;
```

Even though we assign the value of str to str1, if we check them in the console, they'll have the same values, but 2 memory boxes were created, and each holds the value "Hello World" inside it. So, we've used 2 memory spaces for the same thing. That's redundant.

If we did the same with objects, on the other hand:

```
let person = {
     fName: "Jane",
     lName: "Doe"
};
let person1 = person;
```

If we check in the console, both have the same property and values, but they don't actually store them separately. Let me prove that to you.

Let me change the value of one of the properties of person1.

```
person1.fName = "John";
console.log(person);
```

We just changed person1, but person has changed as well. This proves that both the variables are just referencing a common property.

If you compare these objects, you'll know the truth.

```
console.log(person === person1);

let person2 = {};
let person3 = {};
console.log(person2 === person3); //false
```

Cloning & merging objects & Object.assign

For…in is best for cloning every property and value.

```
let person = {
     fName: "Jane",
     lName: "Doe"
};
let person1 = {};
for(let prop in person) {
     person1[prop] = person[prop];
}
console.log(person1);
person1.fName = "John";
console.log(person.fName);
```

It's proper cloning now. Changing one object doesn't change the other one.

Object.assign is a pre-defined method and it can be used to clone and merge objects into another root object.

The syntax is as follows:

```
Object.assign(root, obj1,obj2,obj3...);
```

If there is just one object to clone:

```
let objectName = Object.assign({}, obj);
```

Example:

```
let person1 = Object.assign({}, person);
console.log(person1);
```

You can use this to merge multiple objects into one object as well, in which case you don't have to assign it to a variable.

After the person object, let's create more objects called voting and career.

```
let voting = {
     canVote: true,
     gender: "female"
};
let career = {
     graduate: true,
     hasJob: true;
}
Object.assign(person,voting,career);
console.log(person);
```

Now we've merged everything.

Constructors

JavaScript constructors is used to set the initial properties of the object that it creates.

If you want multiple objects to have the same set of properties and methods, but save typing the same thing over and over again, constructors are the best way to go about it.

They look just like functions, but the difference in how you call the function.

It's good practice to name the constructor function with a starting capital letter.

```
function People(fName, lName) {
     this.fName: fName;
     this.lName: lName;
}
let person1 = new People("John","Doe");
let person2 = new People("Susan", "Dee");
```

When the new keyword is used, it creates an empty object, replaces "this" with the new object (person1 or person2), then assigns the attribute values and returns them back to the object.

```
console.log(person1);
console.log(person2);
console.log(person1.fName);
console.log(person2.lName);
```

We can create methods here as well.

```
this.fullName = function() {
     return `${this.fName} ${this.lName}`;
}
console.log(person1.fullName());
```

We should not call objects without new, then it'll take "this" as window, and this.fName would become window.fName – that's not good practice.

To prevent that, can use new.target. It returns empty if called without new and the function (true) if returned with new.

```
function People(fName, lName) {
     if(!new.target) {
          return new People(fName, lName); //call it back with
new so its rectified
     }
     this.fName = fName;
     this.lName = lName;
}
```

If called without new in 'use strict' mode, then it'll show an error. So, in use strict, if you don't want error, use the if to circumvent the call so you don't get, "can't set property fName of undefined".

Can get return from constructors though constructors automatically return the property and method values without mentioning.

If you return an object, then it'll be returned, and the rest will be overridden. If you return primitive, it'll be ignored.

```
function People(fName, lName) {
     this.fName = fName;
     this.lName = lName;
     return {
          name: "Jane Doe",
          age: 30
     };
```

```
}
console.log(person1);
Output: {name: "Jane Doe", age: 30}
```

But, if you just the following, your browser will ignore it:

```
return "Jane Doe";
```
Was ignored.

Constructor expressions:

Just like functions, you can create constructors as function expressions as well. Let's see how:

```
let People = function(fName,lName) {
     this.fName = fName;
     this.lName = lName;
}
let person1 = new People("John","Doe");
console.log(person1);
```

Private properties with closures

If you want properties to be private and not accessed outside of the object, you can do that with closures and getters and setters.

Receive the arguments to start with in the constructors with a prefix _, while calling with new.

Then, to set the arguments, call again with setName, but this time, receive normally without prefix, and set the variable with prefix with the variable without. _fName will now become an inner variable and person.fName will return undefined now.

Now, when called with getName, it'll return the name as described.

```
function People(_fName, _lName) {
     this.setName = function(fName,lName) {
          _fName = fName;
          _lName = lName;
     }
     this.getName = function() {
```

```
            return `${_fName} ${_lName}`;
        }
}
let person1 = new People("John","Doe");
let person2 = new People("Susan", "Dee");
console.log(person1);
console.log(person2);
person1.setName();
console.log(person2.getName());
console.log(Object.getOwnPropertyNames(person1));
```

The above lines of code returns the list of properties in an array: ["fName", "lName"]

This will not return the symbols (I'll teach this in a later lesson) or methods.

Property flags, descriptors

When you create a property, on the outside, you just see the values, but that's not the only thing happening here. Other descriptors for the property, apart from "value" are being added.

They are the following:

Writable, enumerable and configurable
Writable – This is the read only descriptor – if this descriptor is true, the value of the property can be changed, otherwise it is fixed

Enumerable – This descriptor gives or takes visibility for the property in loops – If its true, then this descriptor will be visible in loops, otherwise it won't be

Configurable – If this is true, the above descriptors can be modified. If not, they are fixed. This property determines whether the property can be deleted or not.

So, while accessing a property, the only thing we see is the value. Where do we see the rest of the descriptors?

Well, just like the getOwnPropertyNames method, there is also the getOwnPropertyDescriptor method which gives all of the descriptor values for that particular property. Let's check.

Syntax is as follows:

```
Object.getOwnPropertyDescriptor(objectName,"propertyName");
```

Property name should be within quotes.

```
console.log(Object.getOwnPropertyDescriptor(person1,"fName"))
{value: "John", writable: true, enumerable: true, configurable:
true}
```

As you can see, by default, the values for writable, enumerable, and configurable are true. That is, by default, a property's value can be changed, it'll be shown on loops and it can be deleted, or the descriptor values can be changed.

But what if we want to change one of these?

We can use the defineProperty method.

Its syntax is as follows:

```
Object.defineProperty(obj,property,descriptor);
```

But that's not how it's written, not exactly:

```
Object.defineProperty(obj,property,{
     descriptor: value
});
```

The property should be within quotes.

Examples:

If we want to change values:

```
Object.defineProperty(person1,"fName",{
     value: "Gary"
});
console.log(person1.fName);
```

Make the property non-writable – cant be edited:

```
Object.defineProperty(person1,"fName",{
     writable: false
});
person1.fName = "Gary"; //error
```

Non enumerable:

```
Object.defineProperty(person1,"fName",{
     enumerable: false
});
for(let prop in person1) {
     console.log(prop);
}
```

Non configurable:

```
Object.defineProperty(person1,"fName",{
     configurable: false
});
delete person1.fName;
```

We can change multiple descriptors at the same time as well:

```
Object.defineProperty(person1,"fName",{
     configurable: false,
     value: Gary,
     writable: false
});
```

As you can see with defineProperty, we can actually use it to create a new property as well, or update the value of a property, instead of using the usual methods that involve the dot notation or the square brackets.

Doing it one property at a time is very time consuming though. You can define multiple properties at the same time by using the defineProperties method.

```
Object.defineProperties(objName,{
     property1 : {descriptor : value...},
     property2 : {descriptor : value...},
     property3 : {descriptor : value...}
});
```

Getters and setters

There are two ways or creating and assigning values to a property in objects. One of them is what we've been seeing so far, the data method.

The next is using getters and setters, called the accessor method.

It's as the name implies. Setter sets the value of a property and get gets the value of said property.

So what happens is, once we create these two methods, whenever you change the value of a property, it'll call the setter code, and whenever you try to access the property, it'll call the getter code.

Let me show you.

Getters and setters should have commas between them or they won't work - set should get the variable name inside it, get should

Setter and getter for a property should have the same name as the property name you want to set

When you call the property with dot notation or square brackets, the getter will return it. When you assign a value to the property outside the object, setters will be called, and it'll create the property and assign that value

The this.propertyName inside of the setter should be different from the actual property name. It's an inner variable, and the setter's name would be the property name accessible from outside the object.

```
The properties won't be created until you trigger the setters
let person = {
     set fName(f) {
          this.f = f; //inner variable/property
     },
     get fName() {
          return this.f;
     },
     set lName(l) {
          this.l = l;
     },
     get lName() {
          return this.l;
```

```
    }
};
```

When you do a console now:

```
console.log(person);
```

This is what you get: {}

Nothing was created yet.

How do we do that ?

Setter first:

```
person.fName = "John";
person.lName = "Doe";
console.log(person);
{f: "John", l: "Doe"}
```

Now, try to call the properties, then getter will be fired.

```
console.log(`${person.fName}'s full name is ${person.fName}
${person.lName}`);
```

You can do the same inside Object.define too.

Symbols

Symbols are another data type of JavaScript which are mostly used with respect to objects.

So, this is how you create them:

```
let sym = Symbol("id");
```

“id” here is just a label. It doesn’t mean anything. It’s mostly used to identify the symbol, for debugging purposes etc.

But, every symbol is unique. If you create another symbol, even if it has the same label, then it’ll still be unique. Let me prove that to you.

```
let sym = Symbol("id");
let sym1 = Symbol("id");
```

```
console.log(sym === sym1); //false
```

So, why are symbols useful?

Well, they are mostly used to create identifiers, things that cannot be changed and so on.

For instance, if your object has a social security number feature, then you don't want that changed outside of the object, right? Or changed by another script outside of your code.

So, in that case, place that number inside of your symbol.

If someone else wants to create a new id, they can just create a new symbol, and as you know, symbols are always unique, even if they have the same label.

Let me show you an example:

Let's create a symbol with label id and assign it to the variable id first.

```
let id = Symbol("id");
let person = {
     name : "Jane Doe",
```

Let's place that id inside of the object. To use symbols as properties, you should always place the variable name inside the square brackets.

```
     [id] : 30654,
};
```

Now, even if you accidentally create another id and give a new name, this is what'll happen.

```
person.id1 = 1253;
console.log(person);
```

So, symbols are basically hidden properties that can be used for unique value storage.

You can't just print out a symbol in an alert box. It won't be automatically converted to a string while printing out, like the other data types.

```
alert(id); //error
```

So, convert it to a string:

```
alert(id.toString()); //Symbol(id)
```

Symbols are skipped by for…in and object.keys

```
for(let key in person) {
     console.log(key);
}
console.log(Object.keys(person));
```

But Object.assign will still clone it.

```
let person1 = Object.assign({},person);
console.log(person1);
```

You can create global symbols as well, that is, symbols that are the same, if created with the same label.

```
let sym = Symbol.for("id");
let sym1 = Symbol.for("id");
console.log(sym === sym1);
```

You can get the label for a symbol by using the keyFor property.

```
console.log(Symbol.keyFor(sym));
console.log(Symbol.keyFor(sym1));
```

Module 3: Prototypes & Prototype Inheritance

The __proto__ you see when you look at an object in the console is the creator of that object.

```
function People(fName, lName) {
     this.fName = fName;
     this.lName = lName;
     this.fullName = function() {
          return `${this.fName} ${this.lName}`;
     }
}
let person1 = new People("John","Doe");
console.log(person1);
```

if you look at the console, and open the object, you'll see the property and method, and at the end, the __proto__. It has the constructor inside, and another __proto__ with a lot of methods.

That __proto__ points to the Object.prototype. It's the creator of this object and it'll have a lot of pre-defined methods you can use.

Everything has such prototypes because everything is an object in Javascript.

Arrays, functions, date etc.

```
let arr = [1,2,3];
console.log(arr);
```

Open the arrow, and it has proto, with lots of properties that you can use on that array.

You must have used some of them already without knowing they come from the proto.

```
console.log(arr.reverse());
```

When using one of the properties of the prototype you don't have to say arr.prototype.reverse(), you can just use it as it is.

The same goes for functions, dates, strings, numbers etc. Everything can be created as an object.

```
let str = new String("hello"); //create like this to make the
primitive an object
console.log(str);
```

When opened, it has a lot of methods and properties inside the prototype as well.

So, everything pretty much inherits from objects, especially Object.

So why use the prototype?

Why not create the methods inside the constructor in itself? What makes the difference?

Well, if you create methods inside the constructor, every time you create an object with it, it'll hard code the methods as well. That's a lot of space and memory used.

If prototype is used, then it'll just create them once, put it in the prototype area, and you can just access it for the object you want to access it for. Create it once and use it many times for many objects. Very efficient. Time as well as space.

That's why it's better to just put the properties inside the constructor and create methods using the prototype.

I want to clarify that the prototype contains the methods and the __proto__ points to that Object.prototype and knows how to use those methods.

```
People.prototype.fullName = function() {
     return `${this.fName} ${this.lName}`;
}
```

Now, look inside and you'll see this method inside __proto__ and it won't be hard coded every time an object is created.

Check this to know what each are:

```
console.log(person1.__proto__ === People.prototype); //true
```

So, __proto__ is for the object instance that gets creator and prototype is for the Object constructor in itself. The object instance inherits from the constructor and the __proto__ inherits from the Object.prototype

[1,2,3] inherits from the Array and Array inherits from the Object.

```
let arr = [1,2,3];
console.log(arr.__proto__ === Array.prototype); //true
console.log(arr.__proto__.__proto__ === Object.prototype);
//true
console.log(arr.__proto__.__proto__.__proto__ === null); //true
```

Above the Object, in the chain, is null.

For primitives, the methods are available in String.prototype, Number.prototype and Boolean.prototype and chain goes up from there.

We can borrow methods from native prototypes for our objects as well.

Prototype inheritance

You can inherit the properties and methods of one object to another. That's called prototype inheritance.

You can inherit the prototype, especially, so all the methods under the prototype will be accessible by the inherited object.

An object can only inherit from one object, not many, and the inheritance cannot be cyclic. That is, if obj2 inherits from obj1, and obj3 inherits from obj2, then obj1 cannot inherit from either obj2 or obj3.

```
function Person(fName, lName) {
     this.fName = fName;
     this.lName = lName;
}
Person.prototype.fullName = function() {
     return `${this.fName} ${this.lName}`;
}
//So Runner inherits from Prototype
Runner.prototype = Person.prototype;
function Runner(speed,...args) {
     //lets get the property values of People inside an array
and apply that using the apply pre-defined function - bind this
to that array and apply it to Person
     Person.apply(this,args);
     this.speed = speed;
}
Runner.prototype.isTop10 = function() {
     if(this.speed < 5) {
          return `${this.fullName()} is in the top 10`;
     }
     else {
          return `${this.fullName()} is not in the top 10`;
     }
}
let person1 = new Person("John", "Doe");
```

```
let runner1 = new Runner(4.5,"John","Doe"); //since we receive
speed first and then the rest parameter values (as an array),
give the value of speed first, and then first and last names
console.log(runner1.isTop10());
console.log(runner1.fName);
```

You can inherit only one of the object instance and not everything as well. That's up to you.

Let's remove this line:

```
Runner.prototype = Person.prototype;
```

Let's retain the apply because we are just getting the rest parameter. If it has nothing, it'll have nothing to bind (lets check).

And also change the dependence code like this:

```
Runner.prototype.isTop10 = function() {
     if(this.speed < 5) {
          return `$The runner is in the top 10`;
     }
     else {
          return `$The runner is not in the top 10`;
     }
}
let person1 = new Person("John","Doe");
let runner1 = new Runner(4.5); //don't have to give fName and
lName because we are directly inheriting from an object instance
which already has those values assigned
let person2 = new Person("Gary","Smith");
let runner2 = new Runner(6);
runner1.__proto__ = person1;
console.log(runner1);
console.log(runner1.fullName());
console.log(runner2.fullName()); //error
```

Prototype chain:

```
let person = {
     fName : "John",
     lName : "Doe"
};
let sportsPerson = {
     sports : "Runner",
     __proto__ : person
```

```
};
let runner = {
     speed : 4.5,
     __proto__ : sportsPerson
};
console.log(runner);
```

Look at the protos - 3 protos - first for runner, then sportsPerson, then person and finally goes to the main Object __proto__ which has all the pre-defined methods - this is called prototype chaining

If you use for…in over an inherited object ,it'll iterate over the inherited properties as well:

```
for(let key in runner) {
     console.log(key);
}
```

Module 3: Classes (ES6) & Class Inheritance

If you're familiar with any other programming languages, then you must have come across classes. Classes are the core of object-oriented programming in a lot of languages.

But JavaScript didn't have them until the ES6 update.

Truth be told, they still don't have them.

They just have the keyword class now, which is a sugarcoated way of working with prototypes. Whenever you use classes, in the background, your prototypes will do all the work. That's what's really working.

But it's a great way of creating objects, setting up constructors, properties and methods, so if you don't want to worry about what's actually going on, I'd recommend directly using classes and not bothering with prototypes unless you're working with a code that uses prototypes.

So, how do you create them? It's pretty simple actually. We've already looked at constructors and prototypes, so it must be easy for you.

Instead of function, use the class keyword, and name the class. It's good practice to capitalize the first letter of the class name.

Then, create a constructor with the constructor function inside the class and then initialize all the properties inside it.

Instead of using function constructorName() like we usually do to create constructors, we'll be directly using constructor() to create the same here.

Then, create your methods below the constructor. That's it. Let me show you.

```
class People {
     constructor(fName,lName) {
          this.fName = fName;
          this.lName = lName;
     }
     fullName() {
          return `${this.fName} ${this.lName}`;
     }
}
```

```
let person1 = new People("John","Doe");
let person2 = new People("Susan","Dee");
console.log(person1);
console.log(person2);
console.log(person1.lName);
console.log(person2.fullName());
```

When you call the new keyword, the object is created and the constructor is called and the properties are initialized. Then you can call the methods as needed.

```
console.log(typeof People); //function
```

Function comes from the constructor method. The constructor method is not compulsory. You can write a class without one as well. If there's no constructor, it'll be considered empty.

All the methods are automatically stored in Person.prototype. You don't have to separately create them like you did when working with constructors and prototypes. So, classes are a higher form of them, but it uses them in the background.

So, whenever we call a method from an object created with the class, it takes the method from the prototype.

```
//Person is actually a constructor and it's a function because
of the constructor method
console.log(Person === Person.prototype.constructor);


//The methods in People's prototype
console.log(Object.getOwnPropertyNames(People.prototype));
//constructor,fullName
```

It might look like syntactic sugar, but it actually isn't.

Class methods are non-enumerable by default.

```
let person1 = new People("John","Doe");
for(let key in person1) {
     console.log(key);
}
//lName and fName and not fullName()
```

It uses strict mode by default as well. So, if we don't use new to call the class constructor, even if your code is not in strict mode, you'll get an error, unlike with normal constructors.

Remove use strict statement.

Now,

```
let person1 = People("John","Doe");
Error: Class constructor People cannot be invoked without 'new'
```

So, classes have some differences to normal constructors and prototypes. It's not just syntactic sugar. It made some improvements to Javascript.

We can use class expressions to create classes, just like functions:

```
let People = class {
```

We can return classes, just like functions and objects:

```
function createPeople(fName,lName) {
     return class {
          constructor() {
               this.fName = fName;
               this.lName = lName;
          }
          fullName() {
               return `${this.fName} ${this.lName}`;
          }
     }
}
let Person = createPeople("John","Doe"); //Person is now a class
- lets create a new object instance from it
console.log(new Person().fullName()); //call the Person()
class's constructor first, and then the fullName() function on
it.
```

Class fields

We can create properties outside of the class constructor as well. Can use it to get input from user and other complicated things. These are called class fields.

```
let People = class {
     age = prompt("What is the person's age?",18);
     constructor(fName,lName) {
          this.fName = fName;
          this.lName = lName;
     }
     fullName() {
          return `${this.fName} ${this.lName}`;
     }
     isMajor() {
          if(this.age > 18) {
               return `${this.fullName()} is a major`;
          }
          else {
               return `${this.fullName()} is a minor`;
          }
     }
}
let person1 = new People("John","Doe");
console.log(person1.isMajor());
console.log(person1.age);
```

Getters and setters

You can create getters and setters in constructors, just like you would for normal constructor functions, but with a difference. You need to create the constructor here first, and that'll invoke the getters, not the new object instance call.

```
class People {
     constructor(fName,lName) {
          this.fName = fName; //invokes fName setter
          this.lName = lName; //invokes lname setter
     }
     set fName(f) {
          this.f = f;
     }
     get fName() {
          return this.f;
     }
     set lName(l) {
          this.l = l;
     }
     get lName() {
          return this.l;
     }
```

```
     fullName() {
          return `${this.f} ${this.l}`;
     }
}
let person1 = new People("John","Doe");
console.log(person1.fullName());
console.log(person1.fName); //invokes the setter for fName
console.log(person1); //look inside proto, getters and setters
were created inside the prototype
```

Method chaining

You can chain method calls one after the other to save some line of space.

```
class Person {
     constructor(fName,lName,age) {
          this.fName = fName;
          this.lName = lName;
          this.age = age;
     }
     isMajor() {
          if(this.age > 18) {
               console.log("Major");
          }
          else {
               console.log("Minor");
          }
     }
     fullName() {
          console.log(`${this.fName} ${this.lName}`);
     }
}
let person1 = new Person("John","Doe",40);
person1.isMajor().fullName(); //get an error saying cannot call
fullName() of undefined.
```

This is because isMajor() returns undefined.

So, what do we do? Return this, so then, person1.isMajor() will return this and the next function call would become again person1.fullName().

Return this for both methods so we can call any method at the start and it'll all work the same.

```
return this; //end of isMajor and fullName
```

```
person1.fullName().isMajor(); //works the same
```

Class inheritance

Classes have inheritance like prototypes do.

We must use the extends keyword with the class that we want to inherit from the parent class.

```
class Person{
     constructor(fName,lName,age) {
          this.fName = fName;
          this.lName = lName;
          this.age = age;
     }
     fullName() {
          return `${this.fName} ${this.lName}`;
     }
}
class Major extends Person {
     isMajor() {
          if(this.age > 18) {
               return true;
          }
          else {
               return false;
          }
     }
}
let person1 = new Major("John","Doe",40);
console.log(person1.isMajor());
```

Can have multiple classes inheriting like this, like a chain.

Overriding methods & properties in Classes

What if the child class has the same properties or methods as the parent class? Which one will get executed? The one immediately next to it will be executed, of course.

```
class Person{
     constructor(fName,lName,age) {
```

```
            this.fName = fName;
            this.lName = lName;
            this.age = age;
      }
      fullName() {
            return `${this.fName} ${this.lName}`;
      }
      isMajor() {
            return (this.age > 18) ? `Major` : `Minor`;
      }
}
class Major extends Person {
      isMajor() {
            return this.age > 18 ? `true` : `false`;
      }
}
let person1 = new Major("John","Doe",40);
console.log(person1.isMajor()); //true
```

So, class Major's isMajor() function was executed, not the Person class's. So, the one nearest in the inheritance tree will be executed.

This is called method overriding.

But sometimes, we might want the child class's method to be executed, after which, or before which, the parent class's method to ALSO be executed.

So, in those classes, use the super keyword, and call the parent method from the child method to get around method overriding.

```
class Person{
      constructor(fName,lName,age) {
            this.fName = fName;
            this.lName = lName;
            this.age = age;
      }
      fullName() {
            return `${this.fName} ${this.lName}`;
      }
      isMajor() {
            if(this.age > 18) {
                  return `${this.fullName()} is a Major`;
            }
```

```
          else {
               return `${this.fullName()} is a Minor`;
          }
     }
}
class Major extends Person {
     isMajor() {
          console.log(this.age > 18 ? `true` : `false`);
          return super.isMajor(); //calling parent isMajor()
after the child isMajor is executed. Can't give a return
statement before this though, then it would never reach this
line. So changing it to a cosnsole. But I'm returning
super.isMajor() or else the parent will return nothing and it'll
be undefined (show)
     }
}
let person1 = new Major("John","Doe",40);
console.log(person1.isMajor());
```

Place super.isMajor() on top of the console and try.

Overriding a constructor

So far, our child classes never had constructors. But what if they did, because they wanted to assign properties while creating the object like a normal class would? Then how would you assign the properties of your parent constructor?

By using super again. Let's see how.

Let's receive age in the major class now and have just one isMajor function in the inherited class, and see what happens.

I'm getting this error:

```
Uncaught ReferenceError: Must call super constructor in derived
class before accessing 'this' or returning from derived
constructor
    at new Major
```

So, let's add super. Call super first to instanciate the parent constructor first and then works on the child constructor. You need to receive all the property values in

the child constructor and send the corresponding ones to the parent constructor, like this:

```
class Person{
     constructor(fName,lName,age) {
          this.fName = fName;
          this.lName = lName;
     }
     fullName() {
          return `${this.fName} ${this.lName}`;
     }
}
class Major extends Person {
     constructor(fName, lName, age) {
          super(fName,lName);
          this.age = age;
     }
     isMajor() {
          console.log(this.age > 18 ? `true` : `false`);
          return super.isMajor();
     }
}
let person1 = new Major("John","Doe",40);
console.log(person1);
```

Static properties and methods

Sometimes, we might need properties or methods that are particular to just the class, which can only be accessed by the class name and not the object instance's name.

We can directly call these properties and methods without creating an object instance.

These properties and methods can be created by pre-pending them with the static keyword.

Let's see some examples of how to use them.

```
class Person{
     static species = "Human being";
     constructor(fName,lName,age) {
          this.fName = fName;
          this.lName = lName;
```

```
          this.age = age;
     }
     fullName() {
          return `${this.fName} ${this.lName}`;
     }
}
let person1 = new Person("John","Doe",40);
let person2 = new Person("Susa","Smith",36);
console.log(`${person1.fullName()} is a ${Person.species}`);
```

That's how you use properties.

Now, to create methods:

```
static hello() {
          console.log("Hello");
}
Person.hello(); //Hello
```

You can send the object as a parameter to that:

```
static hello(person) {
          console.log(`Hello ${person.fName}`);
}
Person.hello(person1);
```

You can use methods to compare and operate on multiple object's property values from within the class.

For example, let's say we want to know who's older between the two objects we sent. We can do that comparison within the Person class by using the static keyword.

```
static whoIsOlder(person1,person2) {
     if(person1.age1 > person2.age2) {
          console.log(`${person1.fullName()} is older than
${person2.fullName()}`);
          }
          else {
               console.log(`${person2.fullName()} is older than
${person1.fullName()}`);
          }
     }
Person.whoIsOlder(person1,person2);
```

You can also inherit the static properties and methods.

```
class SportsMan extends Person {
     constructor(fName,lName,age,sports) {
          super(fName,lName,age);
          this.sports = sports;
     }
}

let person3 = new SportsMan("Jane","Dee",30,"Tennis");
console.log(`${person3.fullName()} is a ${SportsMan.species} who
plays ${person3.sports}`);
```

Private and protected properties in classes

Unlike the other OOPS languages like Java or C++, you don't have readymade ways to create private and protected class properties and methods in javascript.

Well, what is it, by the way?

As the name implies, protected properties are protected. They cannot be accessed outside of the class. They are hidden from the outside. We looked at creating properties like these in a previous module. Let's look at doing the same with classes.

Now private properties are pretty much the same, but the difference between private and protected is that protected properties can be accessed by child classes while private properties cannot be.

Protected property

So, let's look at protected first. We can use getters and setters to do that. But we can also use normal functions/methods as well, so let's look at both ways of doing that.

It's usually good practice to prefix the name of the protected property with an _. It's not compulsory, but it's done.

```
class People {
```

```
     _name = ''; //protected
     set name(n) {
          this._name = n;
     }
     get name() {
          return this._name;
     }
}
let person1 = new People();
person1.name = "John Doe";
console.log(person1.name);
console.log(person1);
```

Protected properties can be inherited. Let's look at a very simple example.

```
class Runner extends People{

}
let runner1 = new Runner();
runner1.name = "Susan Dee";
console.log(runner1); //accesses _name which is a hidden
property
console.log(runner1.name);
```

But there is a problem with this method. If you know the name of the hidden property, you can actually access it from the outside, so it is not really hidden, after all.

```
console.log(person1._name);
```

Instead of getters and setters, you can use functions as well.

```
class People {
     _name = ''; //protected
     setName(n) {
          this._name = n;
     }
     getName() {
          return this._name;
     }
}
let person1 = new People();
person1.setName("John Doe");
console.log(person1.getName());
console.log(person1);
```

Read only property

You can also make a property read only, as in, it can only be set when the object is created, probably with the constructor, but it cannot be changed after. To do this, only set a getter for the property, and not a setter. Also, use the constructor function to initialize the value.

```
class People {
     constructor(name) {
          this._name = name;
     }
     get name() {
          return this._name;
     }
}
let person1 = new People("John Doe");
console.log(person1.name); //invokes the getter function
console.log(person1);
```

There's an actual syntax for private properties now. It's a latest addition to the Javascript language so there is not much browser support for it yet, but let's look at it.

So, you just prefix the property with # to make it private. The color changes in your editor, and it'll be have as a private property should.

```
class People {
     #name = ''; //protected
     set name(n) {
          this.#name = n;
     }
     get name() {
          return this.#name;
     }
}
let person1 = new People();
person1.name = "John Doe";
console.log(person1.name);
console.log(person1);
console.log(person1.#name); //error - private field #name must
be declared in an enclosing class
```

Now let's look at inheriting this private property from inside the child class.

```
class Runner extends People {
     this.#name = "Susan Dee"; //doesn't work
}
let runner1 = new Runner();
runner1.name = "Susan Dee";
console.log(runner1);
console.log(runner1.#name); //error
```

Factories in Javascript

At the base of it all, factories are just functions that return objects. In most cases, you can use factory functions instead of classes and they are much simpler than classes.

So why use factories instead of classes? Well, first, it's yet another way of creating objects. Also, you don't have to mess around with "this" and "new" anymore (you'll see how) and so you can avoid a lot of the problems that come with "this" and "new".

Also, you can make properties and their values truly private without having to jump through a lot of loops like you would with classes.

So, let's look at an example of a factory function with private properties now.

You don't have to capitalize the first letter of a factory function because it is just a function.

```
function people(f,l) {
     let fName = f; //private - not accessible from outside the
function, even by the object instances
     let lName = l; //both can't have the same name or you'll
get an error
     return {
          fullName() {
               return `${fName} ${lName}`;
          }
     }
}
let person1 = people("John","Doe");
console.log(person1.fullName());
console.log(person1.fName); //undefined - not accessible
```

Of course, you can make the properties public like you would with a normal class as well:

```
function people(fName,lName) {
     return {
          fName : fName, //can be the same name now since one of
them is a property name and not a variable name
          lName : lName,
          fullName() {
               return `${fName} ${lName}`;
          }
     }
}
```

Now you can access them from the outside.

Module 4: Modules (ES6)

Modules basically let you separate your code into different parts, called modules. Each module is usually in a different folder. This makes code more readable and easier to organize it. It makes your code clean and maintainable.

You can break your code into smaller codes and let them interact with each other so later on you can easily understand each code.

Script tags should be written like this:

```
<script type="module" src="hello.js"></script>
<script type="module" src="imp.js"></script>
```

Both the exporter and the importer should have the type as module, otherwise you won't be able to use the export and import keywords on them.

The function or variable you want exported should start with the export keyword.

```
export let name = "hello";
```

You can import it into the other file by using the import keyword.

```
import {name} from './hello.js';
```

Then you can use it in your code:

```
alert("The variable contains " + name);
```

The same goes for functions.

Modules always use 'use strict' by default.

There is also scope between modules. It's called a module level scope. Unless a variable is exported to another module, it can't be accessed by that module (show example).

Examples of export:

You can export anything.

We looked at normal variables before. It can be anything: Boolean, string, number, etc.

Let's look at more now.

You can export functions:

```
export function sayHi() {
     console.log("Hi");
}

import {sayHi} from "./hello.js";
sayHi();
```

You can import arrays and use them in any way we want.

```
export let arr = [1,2,3,"Hello"];

import {arr} from "./hello.js";
arr[4] = true; //add new values
console.log(arr);
console.log(arr[3] + " there!"); //access values
arr = [5,4,7,3,9,2,1]; //completely rewrite it - error because
it was exported - can't completely change it
arr[1] = 5;
console.log(arr); //can change current values because it is not
complete reassignment
```

Reassigning doesn't work between modules, no matter what type of variable that is. The outside module can't reassign a variable that wasn't originally its.

Changing array elements worked because we aren't completely changing the array, but that isn't the case with primitive data types like string, Boolean or numbers, so you can't change them in any way outside of the module. The minute it is exported, your browser will import it as a constant in the other modules.

```
export let str = "Hello";
import {str} from "./hello.js";
str = "hi"; //error
```

The same rule follows for objects, constructors, and classes. Let's take a look at them now.

```
export let person = {
     fName : "John",
     lName : "Doe",
     age : 35
};
import {person} from "./hello.js";
console.log(person);
console.log(person.fName);
person.fullName = function() {
     return `${this.fName} ${this.lName}`;
}
console.log(person.fullName());
person = {}; //error
delete person.age;
console.log(person);
```

Now for constructors:

```
export function Person(fName,lName) {
     this.fName = fName;
     this.lName = lName;
}
import {Person} from "./hello.js";
console.log(Person);
let person1 = new Person("John","Doe");
console.log(person1);
```

You can use prototypes inside your imported modules too:

```
Person.prototype.fullName = function() {
     return `${this.fName} ${this.lName}`;
}
console.log(person1); //open proto to see fullName method in
prototype
Can inherit prototype within the module too:
function Sports() {

}
Sports.prototype = Person.prototype;
let sports1 = new Sports();
console.log(sports1); //look into proto and you'll see fullName
method inside prototype.
//Let's create a simple method inside Person prototype and see
if that is inherited within sports.
Person.prototype.sayHi = function() {
```

```
     console.log("Hello there!");
}
sports1.sayHi();
```

Now finally for classes:

```
export class Person {
     constructor(fName,lName) {
          this.fName = fName;
          this.lName = lName;
     }
     sayHi() {
          console.log(`Hi ${this.fName}`);
     }
}

import {Person} from "./hello.js";

let person1 = new Person("John","Doe");
person1.sayHi();
//Now for inheritance
class Child extends Person {
     constructor(fName,lName,age) {
          super(fName,lName);
     }
     fullName() {
          return `${this.fName} ${this.lName}`;
     }
}
let child1 = new Child("Susan","Smith",5);
console.log(child1);
child1.sayHi();
console.log(child1.fullName());
```

Obviously, you can export and import functions as well. We've already looked at some examples of doing the same when we looked at objects, constructors and classes.

Works with constants as well. Usually constants cannot be changed, and that holds true for modules as well:

```
export const PI = 3.14;
```

```
import {PI} from "./hello.js";
PI = 4; //error
```

Multiple ways of importing variables in modules

You can import things with a different name as well:

```
export let name = "DigiFisk";
import {name as digi} from './hello.js';
console.log(digi);
```

Once you import the variable with another name, you can't use the original name inside the imported module:

```
console.log(name); //nothing
```

You can export multiple things at the same time as well:

```
const PI = 3.14;
function sayHi() {
     return "Hi";
}
let person = {
     fName : "John",
     lName : "Doe"
};
export {PI,sayHi,person};
```

If you export multiple things at the same time, then you can import multiple things at the same time too.

```
import {PI,sayHi,person} from "./hello.js";
console.log(`${person.fName} says ${sayHi()} and ${PI}`);
```

You can export everything you can from a module with the asterisk symbol, at which point it'll be imported as an object, and this is how you make it work:

```
import * as all from "./hello.js";
console.log(`${all.person.fName} says ${all.sayHi()} and
${all.PI}`);
```