JAVASCRIPT

Table of Contents

[DAY1 – Getting Started with Javascript 3](#_Toc37870031)

[JavaScript data types and data structures 3](#_Toc37870032)

[Data and Structure types 3](#_Toc37870033)

[Primitive values 3](#_Toc37870034)

[String Type 4](#_Toc37870035)

[Object type 4](#_Toc37870036)

[Data property: 5](#_Toc37870037)

[Symbol type 6](#_Toc37870038)

[Dates ; 6](#_Toc37870039)

[JSON 7](#_Toc37870040)

[DAY 2 – Javascript Object and Prototype chaining 8](#_Toc37870042)

[Introducing JavaScript objects 8](#_Toc37870043)

[Object basics 8](#_Toc37870044)

[Object-oriented JavaScript 9](#_Toc37870045)

[Constructors and object instances 10](#_Toc37870046)

[Object prototypes 11](#_Toc37870047)

[Modifying prototypes 14](#_Toc37870048)

[DAY 3 – ES6 14](#_Toc37870049)

Completed By Diwakar Jha,

Reference Taken from (MDN) <https://developer.mozilla.org/>

# DAY1 – Getting Started with Javascript

## JavaScript data types and data structures

JavaScript is a loosely typed or a dynamic language. Variables in JavaScript are not directly associated with any particular value type, and any variable can be assigned (and re-assigned) values of all types.

let foo = 42; // foo is now a number

foo = 'bar'; // foo is now a string

foo = true; // foo is now a Boolean

### Data and Structure types

The latest ECMAScript standard defines nine types:

* Six **Data Types** that are [primitives](https://developer.mozilla.org/en-US/docs/Glossary/Primitive), checked by [typeof](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/typeof) operator:
  + [undefined](https://developer.mozilla.org/en-US/docs/Glossary/Undefined) : typeof instance === "undefined"
  + [Boolean](https://developer.mozilla.org/en-US/docs/Glossary/Boolean) : typeof instance === "boolean"
  + [Number](https://developer.mozilla.org/en-US/docs/Glossary/Number) : typeof instance === "number"
  + [String](https://developer.mozilla.org/en-US/docs/Glossary/String) : typeof instance === "string"
  + [BigInt](https://developer.mozilla.org/en-US/docs/Glossary/BigInt) : typeof instance === "bigint"
  + [Symbol](https://developer.mozilla.org/en-US/docs/Glossary/Symbol) : typeof instance === "symbol"
* [null](https://developer.mozilla.org/en-US/docs/Glossary/Null) : typeof instance === "object". Special [primitive](https://developer.mozilla.org/en-US/docs/Glossary/Primitive) type having additional usage for it's value: if object is not inherited null is shown at the end of [Prototype Chain](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain);
* [Object](https://developer.mozilla.org/en-US/docs/Glossary/Object) : typeof instance === "object". Special non data but Structural type for any [constructed](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects#The_Constructor) instance instance also used as data structures: new [Object](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object), new [Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array), new [Map](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map), new [Set](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set), new [WeakMap](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WeakMap), new [WeakSet](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WeakSet), new [Date](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date) and almost everything made with [new keyword](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/new);
* [Function](https://developer.mozilla.org/en-US/docs/Glossary/Function) non data structure, though it also answers for typeof operator: typeof instance === "function". This answer is done as a special shorthand for Functions, though every Function constructor is derived from Object constructor.

### Primitive values

All types except objects define immutable values (that is, values which can't be changed). For example (and unlike in C), Strings are immutable. We refer to values of these types as "primitive values".

The number type has only one integer with two representations: 0 is represented as both -0 and +0. (0 is an alias for +0.)

+0===-0

> 42 / +0

Infinity

> 42 / -0

-Infinity

### String Type

JavaScript's [String](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String) type is used to represent textual data. It is a set of "elements" of 16-bit unsigned integer values. Each element in the String occupies a position in the String. The first element is at index 0, the next at index 1, and so on. The length of a String is the number of elements in it.

Unlike some programming languages (such as C), JavaScript strings are immutable. This means that once a string is created, it is not possible to modify it.

However, it is still possible to create another string based on an operation on the original string. For example:

* A substring of the original by picking individual letters or using [String.substr()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/substr).
* A concatenation of two strings using the concatenation operator (+) or [String.concat()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/concat).

### Object type

In computer science, an object is a value in memory which is possibly referenced by an [identifier](https://developer.mozilla.org/en-US/docs/Glossary/Identifier).

Properties:

In JavaScript, objects can be seen as a collection of properties. With the object literal syntax, a limited set of properties are initialized; then properties can be added and removed. Property values can be values of any type, including other objects, which enables building complex data structures. Properties are identified using key values. A key value is either a String or a Symbol value.

Examples:->

### Data property:

Associates a key with a value, and has the following attributes:

| **Attributes of a data property** | | | |
| --- | --- | --- | --- |
| **Attribute** | **Type** | **Description** | **Default value** |
| [[Value]] | Any JavaScript type | The value retrieved by a get access of the property. | undefined |
| [[Writable]] | Boolean | If false, the property's [[Value]] cannot be changed. | false |
| [[Enumerable]] | Boolean | If true, the property will be enumerated in [for...in](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...in) loops. See also [Enumerability and ownership of properties](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Enumerability_and_ownership_of_properties). | False |
| [[Configurable]] | Boolean | If false, the property cannot be deleted, cannot be changed to an  accessor property, and attributes other than [[Value]] and  [[Writable]] cannot be changed. | False |

#### Accessor property

Associates a key with one of two accessor functions (get and set) to retrieve or store a value, and has the following attributes:

| **Attributes of an accessor property** | | | |
| --- | --- | --- | --- |
| **Attribute** | **Type** | **Description** | **Default value** |
| [[Get]] | Function object or undefined | The function is called with an empty argument list and retrieves the property value whenever a get access to the value is performed. See also [get](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/get). | undefined |
| [[Set]] | Function object or undefined | The function is called with an argument that contains the assigned value and is  executed whenever a specified property is attempted to be changed. See also [set](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/set). | Undefined |
| [[Enumerable]] | Boolean | If true, the property will be enumerated in [for...in](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...in) loops. | False |
| [[Configurable]] | Boolean | If false, the property can't be deleted and can't be changed to a data property. | false |

Example->TODO

A JavaScript object is a mapping between keys and values. Keys are strings (or [Symbol](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Symbol)s), and values can be anything. This makes objects a natural fit for [hashmaps](http://en.wikipedia.org/wiki/Hash_table).

Functions are regular objects with the additional capability of being callable.

### Symbol type

Symbols are new to JavaScript in ECMAScript 2015. A Symbol is a unique and immutable primitive value and may be used as the key of an Object property. In some programming languages, Symbols are called "atoms".

### Dates ;

When representing dates, the best choice is to use the built-in [Date utility](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date) in JavaScript.

Example

Indexed Collection : Array

[Arrays](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Array) are regular objects for which there is a particular relationship between integer-key-ed properties and the length property.

Examples ->

The **slice()** method returns a shallow copy of a portion of an array into a new array object selected from begin to end (end not included) where begin and end represent the index of items in that array. The original array will not be modified.

The **splice()** method changes the contents of an array by removing or replacing existing elements and/or adding new elements [in place](https://en.wikipedia.org/wiki/In-place_algorithm).

The **reduce()** method executes a **reducer** function (that you provide) on each element of the array, resulting in a single output value.

arr.reduce(callback( accumulator, currentValue[, index[, array]] )[, initialValue])

### JSON

JSON (**J**ava**S**cript **O**bject **N**otation) is a lightweight data-interchange format, derived from JavaScript, but used by many programming languages. JSON builds universal data structures.

Methods

JSON.parse(text[, reviver])

Parse the string text as JSON, optionally transform the produced value and its properties, and return the value. Any violations of the JSON syntax, including those pertaining to the differences between JavaScript and JSON, cause a SyntaxError to be thrown. The reviver option allows for interpreting what the replacer has used to stand in for other datatypes.

JSON.stringify(value[, replacer[, space]])

Return a JSON string corresponding to the specified value, optionally including only certain properties or replacing property values in a user-defined manner. By default, all instances of undefined are replaced with null, and other unsupported native data types are censored. The replacer option allows for specifying other behavior.

# DAY 2 – Javascript Object and Prototype chaining

## Introducing JavaScript objects

In JavaScript, most things are objects, from core JavaScript features like strings and arrays to the browser [APIs](https://developer.mozilla.org/en-US/docs/Glossary/API) built on top of JavaScript. You can even create your own objects to encapsulate related functions and variables into efficient packages, and act as handy data containers. The object-based nature of JavaScript is important to understand if you want to go further with your knowledge of the language, therefore we've provided this module to help you. Here we teach object theory and syntax in detail, then look at how to create your own objects.

## Object basics

An object is a collection of related data and/or functionality (which usually consists of several variables and functions — which are called properties and methods when they are inside objects.)

const person = {

name: ['Bob', 'Smith'],

age: 32,

gender: 'male',

interests: ['music', 'skiing'],

bio: function() {

alert(this.name[0] + ' ' + this.name[1] + ' is ' + this.age + ' years old. He likes ' + this.interests[0] + ' and ' + this.interests[1] + '.');

},

greeting: function() {

alert('Hi! I\'m ' + this.name[0] + '.');

}

};

What is "this"?

The this keyword refers to the current object the code is being written inside — so in this case this is equivalent to person.

const person1 = {

name: 'Chris',

greeting: function() {

alert('Hi! I\'m ' + this.name + '.');

}

}

const person2 = {

name: 'Deepti',

greeting: function() {

alert('Hi! I\'m ' + this.name + '.');

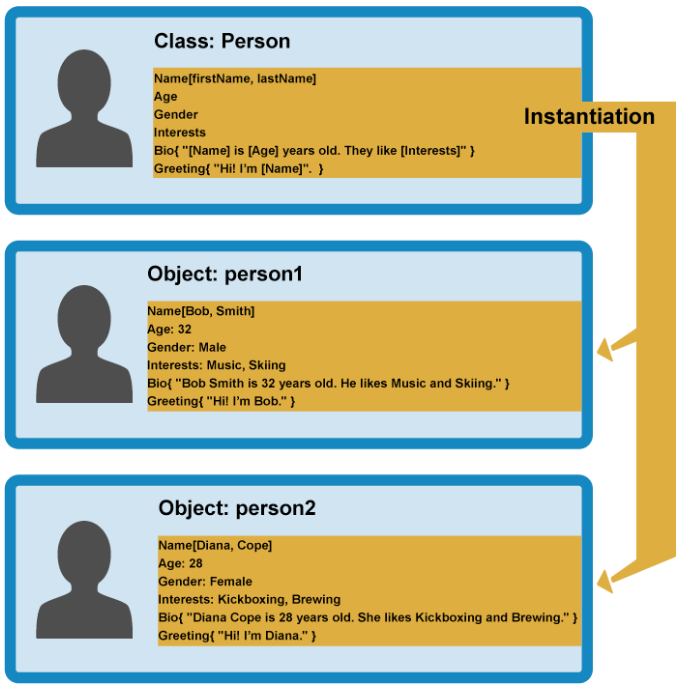
}

}

## Object-oriented JavaScript

The basic idea of OOP is that we use objects to model real world things that we want to represent inside our programs, and/or provide a simple way to access functionality that would otherwise be hard or impossible to make use of.

Let's consider a simple program that displays information about the students and teachers at a school.



When an object instance is created from a class, the class's **constructor function** is run to create it. This process of creating an object instance from a class is called **instantiation** — the object instance is **instantiated** from the class.

## Constructors and object instances

JavaScript uses special functions called **constructor functions** to define and initialize objects and their features.

function Person(name) {

this.name = name;

this.greeting = function() {

alert('Hi! I\'m ' + this.name + '.');

};

}

The constructor function is JavaScript's version of a class. Notice that it has all the features you'd expect in a function, although it doesn't return anything or explicitly create an object — it basically just defines properties and methods. Notice also the this keyword being used here as well — it is basically saying that whenever one of these object instances is created, the object's name property will be equal to the name value passed to the constructor call, and the greeting() method will use the name value passed to the constructor call too.

Now how to create object-

let person1 = new Person('Bob');

let person2 = new Person('Sarah');

In each case, the new keyword is used to tell the browser we want to create a new object instance, followed by the function name with its required parameters contained in parentheses, and the result is stored in a variable .

// See the objects in console

let person1 = new Object({

name: 'Chris',

age: 38,

greeting: function() {

alert('Hi! I\'m ' + this.name + '.');

}

});

Using the create() method

JavaScript has a built-in method called [create()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/create) With it, you can create a new object based on any existing object.

let person2 = Object.create(person1);

## Object prototypes

Prototypes are the mechanism by which JavaScript objects inherit features from one another.

An object's prototype object may also have a prototype object, which it inherits methods and properties from, and so on. This is often referred to as a **prototype chain**, and explains why different objects have properties and methods defined on other objects available to them.

 the properties and methods are defined on the prototype property on the Objects' constructor functions, not the object instances themselves.

In JavaScript, a link is made between the object instance and its prototype (its \_\_proto\_\_ property, which is derived from the prototype property on the constructor), and the properties and methods are found by walking up the chain of prototypes.

Lets start with previous example again.

function Person(first, last, age, gender) {

// property and method definitions

this.name = {

'first': first,

'last' : last

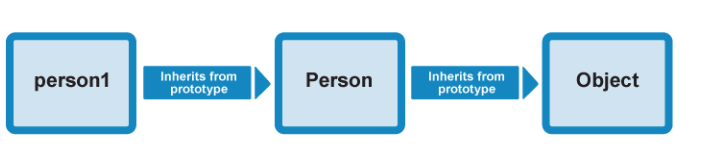
};

this.age = age;

this.gender = gender;

}

let person1 = new Person('Bob', 'Smith', 32, 'male');



See the property available on person1 object. // console or ide

Lets call

person1.valueOf()

This method — [Object.valueOf()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/valueOf) is inherited by person1 because its constructor is Person(), and Person()'s prototype is Object(). valueOf() returns the value of the object it is called on — try it and see! In this case, what happens is:

* The browser initially checks to see if the person1 object has a valueOf() method available on it, as defined on its constructor, Person().
* It doesn't, so the browser then checks to see if the Person() constructor's prototype object (Object()) has a valueOf() method available on it. It does, so it is called, and all is good!

1. You can check out existing prototype properties for yourself — go back to our previous example and try entering the following into the JavaScript console:

Person.prototype

1. The output won't show you very much because we haven't defined anything on our custom constructor's prototype! By default, a constructor's prototype always starts empty. Now try the following:

Object.prototype

You'll see a large number of methods defined on Object's prototype property, which are then available on objects that inherit from Object, as shown earlier.

You'll see other examples of prototype chain inheritance all over JavaScript — try looking for the methods and properties defined on the prototype of the [String](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String), [Date](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date), [Number](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number), and [Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array) global objects

Earlier on we showed how the [Object.create()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/create) method can be used to create a new object instance.

1. For example, try this in your previous example's JavaScript console:

let person2 = Object.create(person1);

1. What create() actually does is to create a new object from a specified prototype object. Here, person2 is being created using person1 as a prototype object. You can check this by entering the following in the console:

person2.\_\_proto\_\_

This will return the person1 object.

person1.constructor

person2.constructor

These should both return the Person() constructor, as it contains the original definition of these instances.

## Modifying prototypes

Person.prototype.farewell = function() {

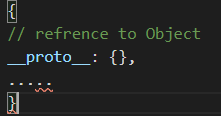
alert(this.name.first + ' has left the building. Bye for now!');

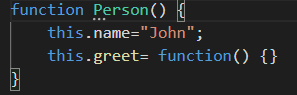
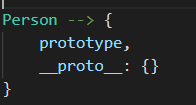
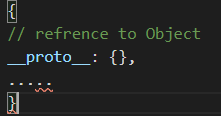
};

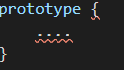
person1.farewell();

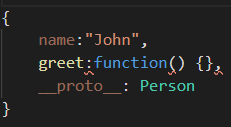
**You will see a alert,**

This new method will be available on all instance derived from person.

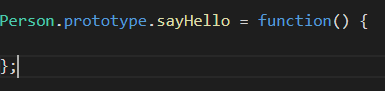
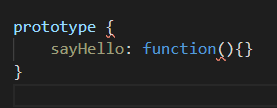
 



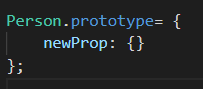
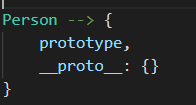
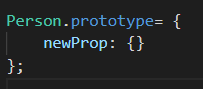
 

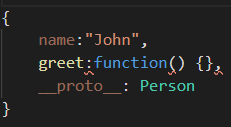
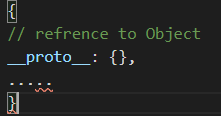
Updated to



When we de reference the Person prototype object



When we dereferenced the prototype of function is now pointing to new prototype object and for new instances \_\_proto\_\_ property will point to new prototype (object).

But for previously created objects they are still pointing to old prototype Object they will have access to previous version of prototype

# DAY 3 – ES6 (ECMA Script 2015)

## Let Const and Block scoping

No variable hoisting, no more depend on function scope, scope defined within curly braces.

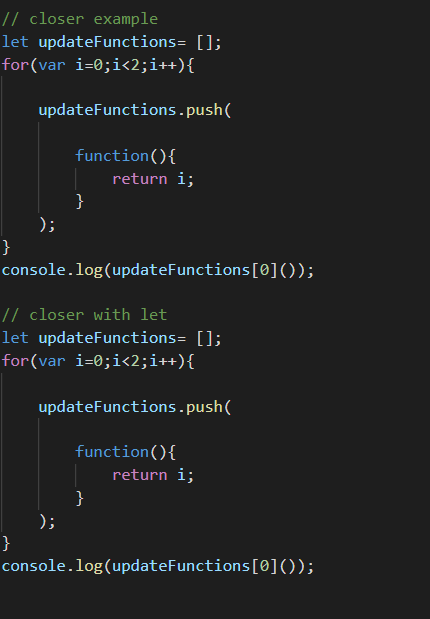
Example of Let

Console.log(product);// undefined

Var product = 10;

Console.log(product);// error

Let product = 20;

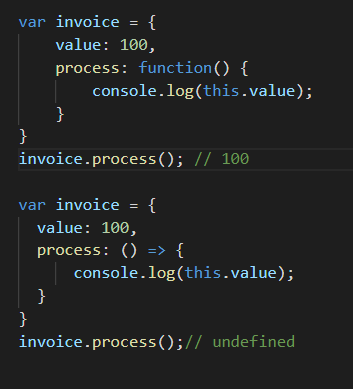


## Arrow Function

An **arrow function expression** is a syntactically compact alternative to a regular [function expression](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/function), although without its own bindings to the [this](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this), [arguments](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/arguments), [super](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/super), or [new.target](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/new.target) keywords. Arrow function expressions are ill suited as methods, and they cannot be used as constructors.

Main benefit of Arrow function is context of this. With call and bind you can’t change the context of this.

For Arrow function it gets this of executing context.



## Default Function Parameters

Var getProduct = function(a=10,b=10){

Return a+b;

}

getProduct();

Default parameter can reference to outside variable and previously defined variable, and if you pass undefined and there is default value mentioned in function definition then default will be used.

## Rest and Spred…

The **rest parameter** syntax allows us to represent an indefinite number of arguments as an array.Object Literal Extension.

function f(*a*, *b*, ...*theArgs*) {

// ...

}

**Spread syntax** allows an iterable such as an array expression or string to be expanded in places where zero or more arguments (for function calls) or elements (for array literals) are expected, or an object expression to be expanded in places where zero or more key-value pairs (for object literals) are expected.

function myFunction(x, y, z) { }

const args = [0, 1, 2];

myFunction(...args);

Spread as array

const parts = ['shoulders', 'knees'];

const lyrics = ['head', ...parts, 'and', 'toes'];

// ["head", "shoulders", "knees", "and", "toes"]

## Object literal Extension

let a = 'foo',

b = 42,

c = {};

// Shorthand property names (ES2015)

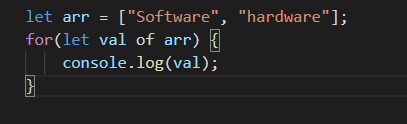
let o = {a, b, c}

// In other words,

console.log((o.a === {a}.a)) // true

## For.. of Loop

New loop to iterate and read element



## Octal and Binary literals

## 

## Template Literals

Template literals are string literals allowing embedded expressions. You can use multi-line strings and string interpolation features with them.

`string text`

`string text line 1

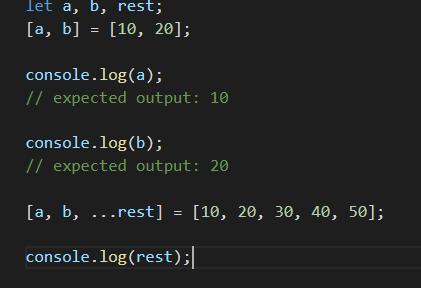
string text line 2`

`string text ${*expression*} string text`

*tag*`string text ${*expression*} string text`

## Destructuring

The **destructuring assignment** syntax is a JavaScript expression that makes it possible to unpack values from arrays, or properties from objects, into distinct variables.



## Module and Classes

### Module Basic

**Installation**

Use of native JavaScript modules is dependent on the [import](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import) and [export](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/export) statements, the browser compatibility

Module supports only on server( as cors denied accces to local files) so create a local server

Go to the project directory run below command in CMD/npm cli

Step 1- $npm init;

Step2- $npm install http-server -g; // install local server locally

Step 3- $http-start // to Run the server

Change in index.html from type= “text/javascript” => type=”module”

Create a directory for Modules

IF type=”module” is not supported by your browsers then you have to use some module loader plugin to work with ex- <https://blog.revillweb.com/using-es2015-es6-modules-with-babel-6-3ffc0870095b>

### Classes

Classes are not hoisted as normal variable or function.

JavaScript classes, introduced in ECMAScript 2015, are primarily syntactical sugar over JavaScript's existing prototype-based inheritance. The class syntax does not introduce a new object-oriented inheritance model to JavaScript.

Defining classes

Classes are in fact "special functions", and just as you can define function expressions and function declarations, the class syntax has two components: class expressions and class declarations.

One way to define a class is using a **class declaration**. To declare a class, you use the class keyword with the name of the class ("Rectangle" here).

class Rectangle {

constructor(height, width) {

this.height = height;

this.width = width;

}

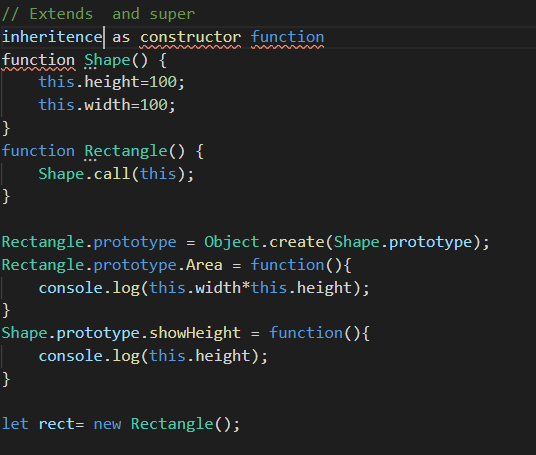
}

#### Hoisting

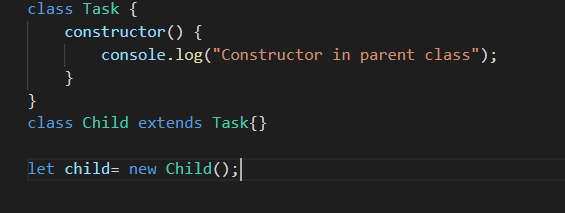
An important difference between **function declarations** and **class declarations** is that function declarations are [hoisted](https://developer.mozilla.org/en-US/docs/Glossary/Hoisting) and class declarations are not. You first need to declare your class and then access it, otherwise code will throw a [ReferenceError](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/ReferenceError):

### Extends and Super

Inheritance in Constructor pattern (Prototypical inheritance)

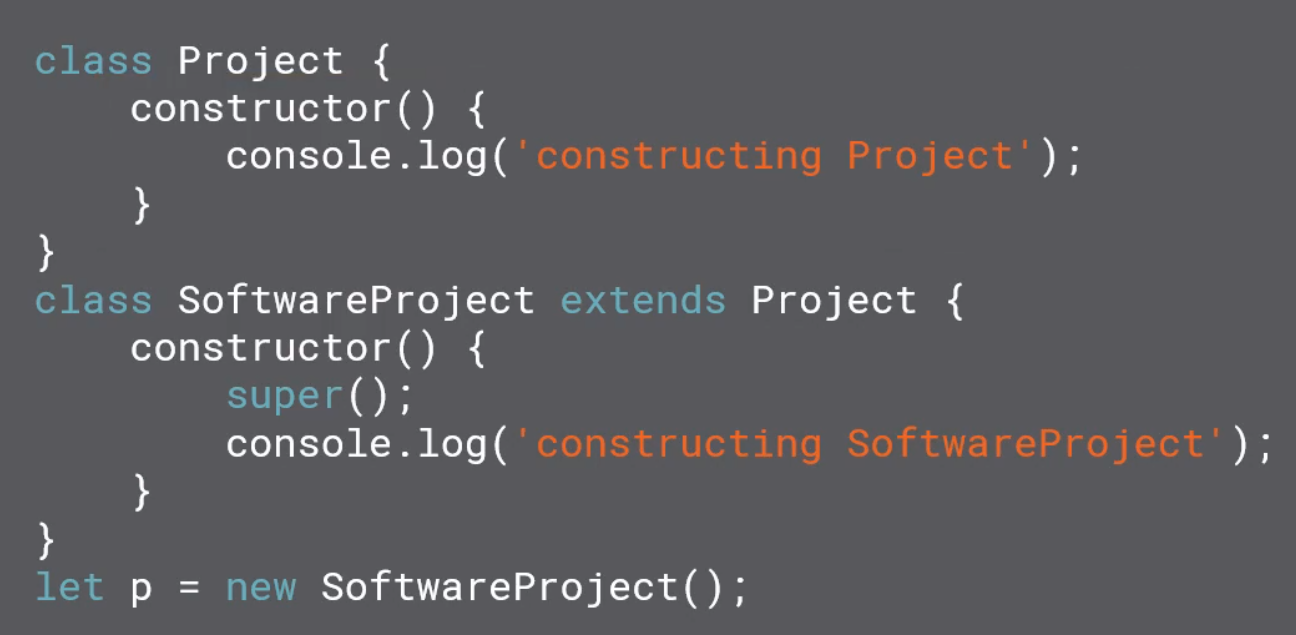


Inheritance in ES6 is with the keywords Extends .

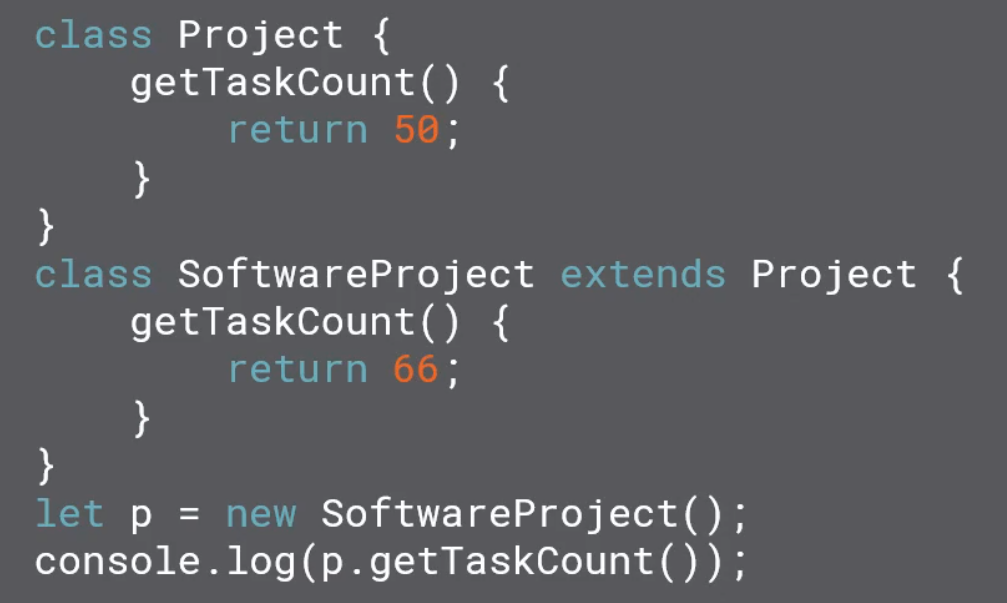


Super keyword

It works similar to other languages (Java), If you define a constructor in child class you have to call super first otherwise it will throw error on instantiation (thi)



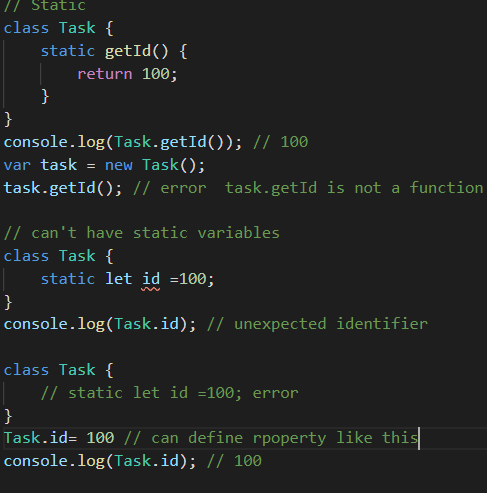
Overriding



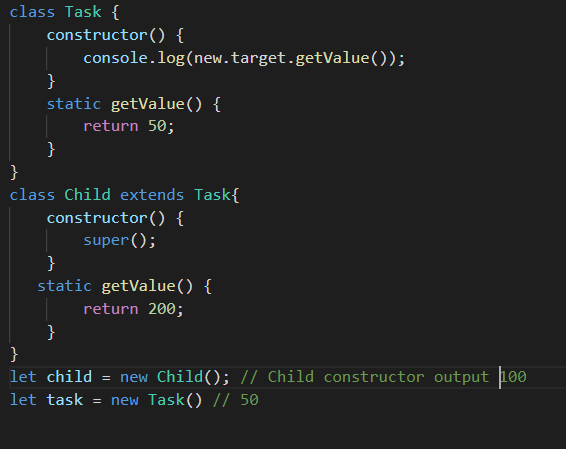
### Static Member

We can have Static method in es6 but can’t define static variables

Static method attached to class and no instance have access to it.

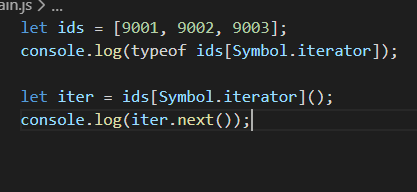


### New.target

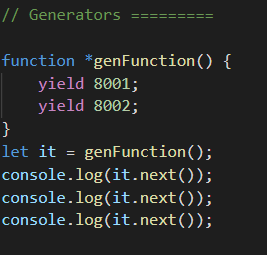


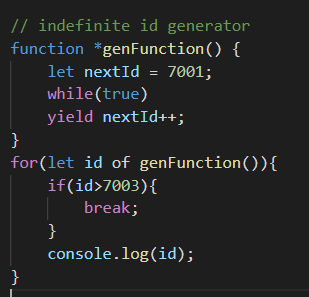
## Iterators and Generators

Iterators ->



Generators -> function \* genFunction

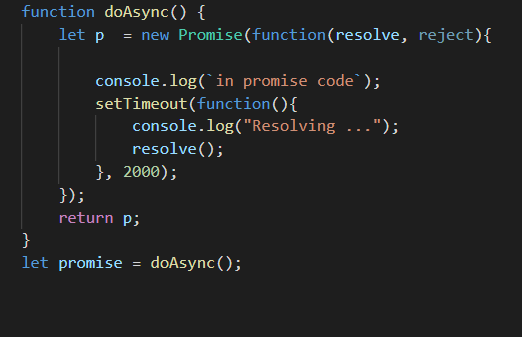


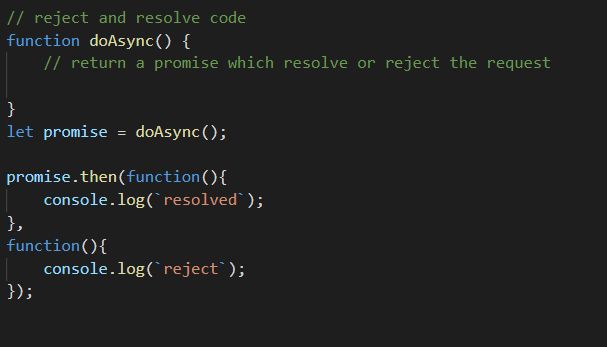


## Promise

It gives ability to do

asynchronous programming natively.





Promise chaining can also be done.