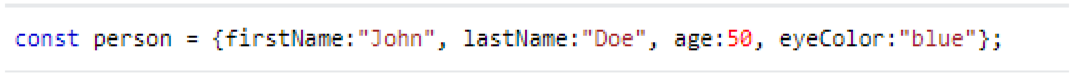
JavaScript OOPS

- JavaScript supports Object Oriented Programming but not in the same way as other OOP languages(C++, php, Java, etc.) do.
- The main difference between JavaScript and the other languages is that, there are no Classes in JavaScript(upto ES-5) whereas Classes are very important for creating objects.
- However there are ways through which we can simulate the Class concept in JavaScript(ES-5).
- Another important difference is Data Hiding. There is no access specifier like (public, private and protected) in JavaScript but we can simulate the concept using variable scope in functions.

Objects

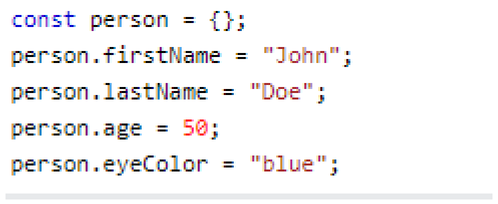
Any real time entity is considered as an Object. Every Object will have some properties and functions.
More ways to create objects in JavaScript like:

1)Creating Object through literal



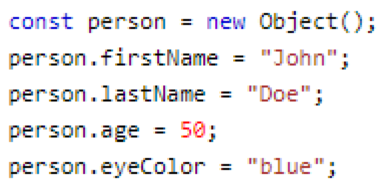
```
const person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};
```

2)Creating with Object.create
var obj= Object.create(null);



```
const person = {};
person.firstName = "John";
person.lastName = "Doe";
person.age = 50;
person.eyeColor = "blue";
```

3)Creating using new keyword



```
const person = new Object();
person.firstName = "John";
person.lastName = "Doe";
person.age = 50;
person.eyeColor = "blue";
```

CREATE YOUR OWN OBJECTS

- We can create our own objects to encapsulate the data and logic. i.e.,

```
 var Person = new Object();
Person.name = “Manvendra SK”;
Person.age = 23;
Person.speakName = function() {
alert(“Hello, I’m ” + this.name);
 };

Person.speakName();
```

ALTERNTAE SYNTAX

- Alternate objects syntax: i.e.,

```
 var Person = {
name: “Manvendra SK”,
age: 23,
speakName: function() {
        alert(“Hello, I’m ” + this.name);
}
 };

 Person.speakName();
```

It’s JSON (JavaScript Object Notation) syntax

CREATING REAL OBJECTS

- We can create objects those can be instantiated using the **new** keyword.

```
var Person = function(name, age) {
  this.name = name;
  this.age = age;
};

Person.prototype.speakName = function() {
  alert("Hello, I'm " + this.name);
};
```

USING REAL OBJECTS

var manvendra = new Person(“Manvendra”, 23);

manvendra.speakName();

- This method of creating objects is called Prototype pattern.

JS Object Accessors(Get & Set)

JavaScript Getter (The get Keyword)
This example uses a lang property to get the value of the language property.

JavaScript Setter (The set Keyword)
This example uses a lang property to set the value of the language property.

Object.defineProperty()
The Object.defineProperty() method can also be used to add Getters and Setters:

Classes

There are no classes in JavaScript(ES-5) as it is Prototype based language. But we can simulate the class concept using JavaScript functions.

```
function Person(){
 //Properties
 this.name="Forzia";
 this.startYear="2015
 ";
 //functions
 this.CompInfo=function(){
 return this.name +" Says
 Hi";
 }
}

//Creating person
instance  var p=new
Person();
alert(p.CompInfo());
```

Constructor (blueprints)

Sometimes we need a "blueprint" for creating many objects of the same "type". The way to create an "object type", is to use an object constructor function.

Actually Constructor is a concept that comes under Classes. Constructor is used to assign values to the properties of the Class while creating object using new operator.

```
function Person(name,age){
//Assigning values through constructor  this.name=name;
this.age=age;
//functions
this.sayHi=function(){
return this.name +" Says Hi";
}}

//Creating person instance
var p=new Person("aravind",23);
alert(p.sayHi());
//Creating Second person instance
var p=new Person("jon",23);
alert(p.sayHi());
```

Inheritance (Prototype)

Inheritance is a process of getting the properties and function of one class to other class.
For example let’s consider "Student" Class, here the Student also has the properties of
name and age which has been used in Person class.
So it's much better to acquiring the properties of the Person instead of re-creating the properties.
Now let’s see how we can do the inheritance concept.

```
var Person = function (){
            this.sayHi=function(){
            return " Says Hi"; }
            }
```

1)**Prototype based Inheritance**

```
            Student.prototype= new Person();
```

2)**Inhertance throught Object.create**

```
            Student.prototype=Object.create(Person);
            var stobj=new Student();  alert(stobj.sayHi());
```

We can do inheritance in above two ways.

Abstraction

Abstraction means hiding the inner implementation details and showing only outer details. To understand Abstraction we need to understand Abstract and Interface concepts from Java. But we don't have any direct Abstract or Interface in JS.
Ex. In JQuery we will use

```
$("#ele")
```

to select select an element with id ele on a web page. Actually this code calls JavaScript code

```
document.getElementById("ele");
```

But we don't need to know that we can happy use the $("#ele") without knowing the inner details of the implementation.

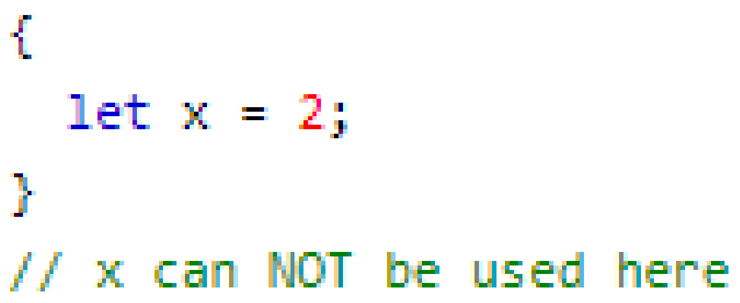
Polymorphism

The word Polymorphism in OOPs means having more than one form.

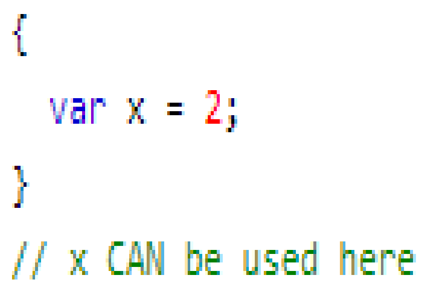
```
function Person(age,weight){
 this.age = age;
 this.weight = weight;
}

Person.prototype.getInfo = function(){  return "I am " + this.age + "
 years old " +
   "and weighs " + this.weight +" kilo.";
 };

function Employee(age,weight,salary){  this.age = age;
 this.weight = weight;  this.salary = salary;
}

Employee.prototype = new Person();

 Employee.prototype.getInfo = function(){  return "I am " + this.age +
  " years old " +  "and weighs " + this.weight +" kilo " +  "and earns "
                                          + this.salary + " dollar.";
                                    };

var person = new Person(50,90);
var employee = new Employee(43,80,50000);
console.log(person.getInfo());  console.log(employee.getInfo());
```

ECMA Script(ES-6) Block Scope

- Before ES6 (2015), JavaScript had only Global Scope and Function Scope.
- ES6 introduced two important new JavaScript keywords: let and const.
- These two keywords provide Block Scope in JavaScript.
- Variables declared inside a { } block cannot be accessed from outside the block:



```
{
  let x = 2;
}
// x can NOT be used here
```



```
{
  var x = 2;
}
// x CAN be used here
```

let keyword

- Variables defined with let cannot be Redeclared.
- Variables defined with let must be Declared before use.
- Variables defined with let have Block Scope.

```
let x = "John Doe";

let x = 0;

// SyntaxError: 'x' has already been declared
```

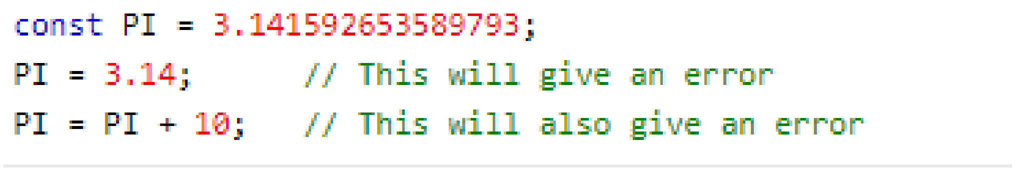
- Redeclaring a variable inside a block will not redeclare the variable outside the block:

```
var x = 10;
// Here x is 10

{
var x = 2;
// Here x is 2
}

// Here x is 2
```

```
let x = 10;
// Here x is 10

{
let x = 2;
// Here x is 2
}

// Here x is 10
```

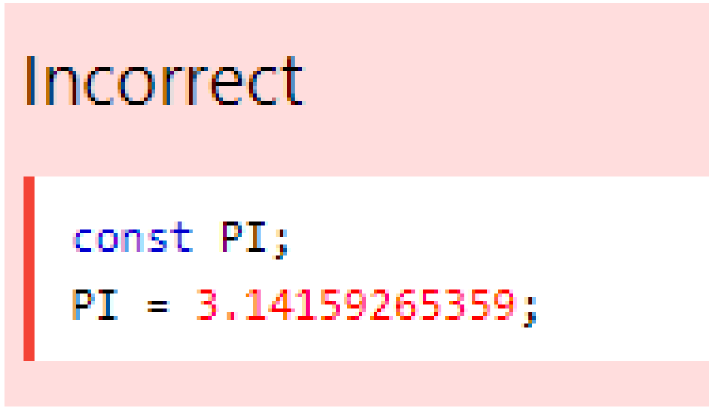
const keyword

- Variables defined with const cannot be Redeclared.
- Variables defined with const cannot be Reassigned.
- Variables defined with const have Block Scope..



```
const PI = 3.141592653589793;
PI = 3.14;      // This will give an error
PI = PI + 10;   // This will also give an error
```

- JavaScript const variables must be assigned a value when they are declared



Incorrect

```
const PI;
PI = 3.14159265359;
```

Arrow function

- Arrow functions allow us to write shorter function syntax

```
hello = function() {
  return "Hello World!";
}
```

```
hello = () => {
  return "Hello World!";
}
```

- If the function has only one statement, and the statement returns a value, you can remove the brackets and the return keyword:

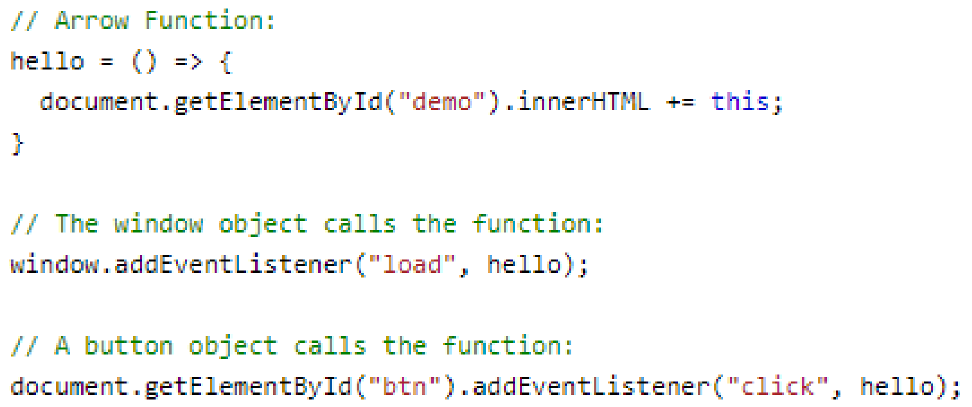
```
hello = () => "Hello World!";
```

- If function with parameters

```
hello = (val) => "Hello " + val;
```

“this” in Arrow function

- The handling of this is also different in arrow functions compared to regular functions.
- In short, with arrow functions there are no binding of this.
- In regular functions the this keyword represented the object that called the function, which could be the window, the document, a button or whatever.
- With arrow functions the this keyword always represents the object that defined the arrow function.



```
// Arrow Function:
hello = () => {
  document.getElementById("demo").innerHTML += this;
}

// The window object calls the function:
window.addEventListener("load", hello);

// A button object calls the function:
document.getElementById("btn").addEventListener("click", hello);
```

JS Classes

- JavaScript Classes are templates for JavaScript Objects.
- Use the keyword class to create a class.
- A JavaScript class is not an object.
- It is a template for JavaScript objects.

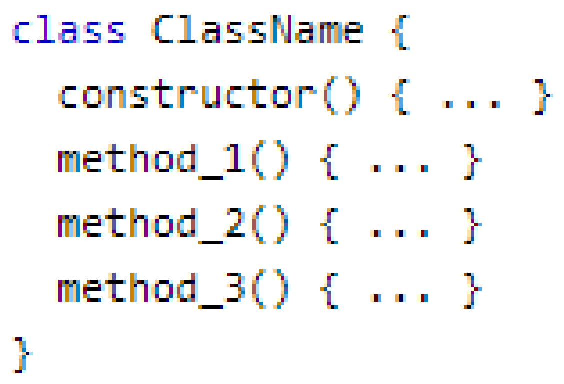
The Constructor Method

- The constructor method is a special method:
- It has to have the exact name "constructor"
- It is executed automatically when a new object is created
- It is used to initialize object properties
- If you do not define a constructor method, JavaScript will add an empty constructor method.

JS Classes

Class Methods

- Class methods are created with the same syntax as object methods.
- Use the keyword class to create a class.
- Always add a constructor() method.
- Then add any number of methods.



```
class ClassName {
  constructor() { ... }
  method_1() { ... }
  method_2() { ... }
  method_3() { ... }
}
```

JS Classes - Inheritance

Inheritance is useful for code reusability: reuse properties and methods of an existing class when you create a new class.

- To create a class inheritance, use the extends keyword.
- A class created with a class inheritance inherits all the methods from another class

Super() method

- The super() method refers to the parent class.
- By calling the super() method in the constructor method, we call the parent's constructor method and gets access to the parent's properties and methods.

JS Classes – Getter & Setters

To add getters and setters in the class, use the get and set keywords.

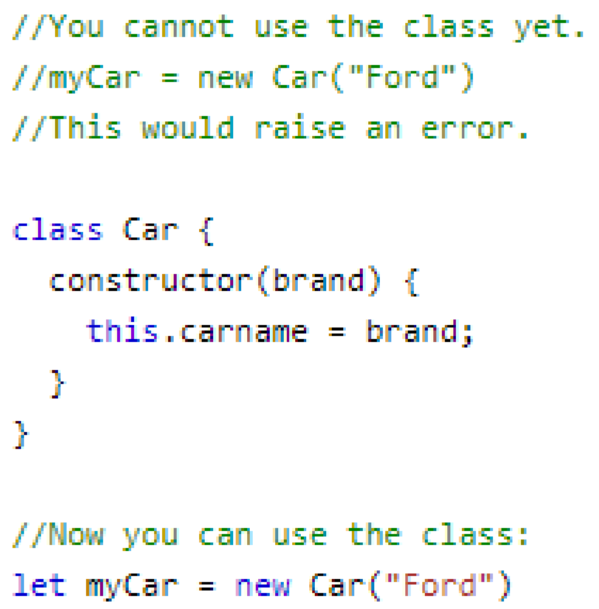
Note:

- Even if the getter is a method, you do not use parentheses when you want to get the property value.
- The name of the getter/setter method cannot be the same as the name of the property
- Many programmers use an underscore character _ before the property name to separate the getter/setter from the actual property

JS Classes – Hoisting

Unlike functions, and other JavaScript declarations, class declarations are not hoisted.

That means that you must declare a class before you can use it



```
//You cannot use the class yet.
//myCar = new Car("Ford")
//This would raise an error.

class Car {
  constructor(brand) {
    this.carname = brand;
  }
}

//Now you can use the class:
let myCar = new Car("Ford")
```

JS Classes – Static method

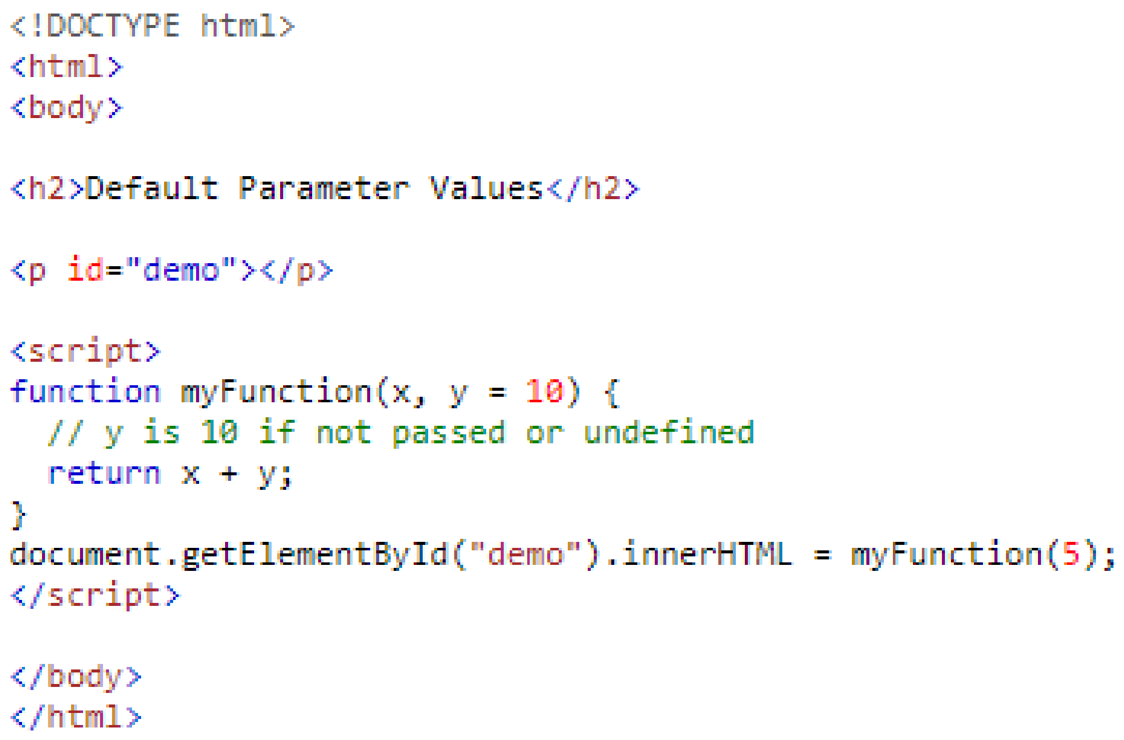
Static class methods are defined on the class itself.

You cannot call a static method on an object, only on an object class.

```
class Car {
  constructor(name) {
    this.name = name;
  }
  static hello() {
    return "Hello!!";
  }
}

let myCar = new Car("Ford");

// You can calll 'hello()' on the Car Class:
document.getElementById("demo").innerHTML = Car.hello();

// But NOT on a Car Object:
// document.getElementById("demo").innerHTML = myCar.hello();
// this will raise an error.
```

Default parameter

ES6 allows function parameters to have default values



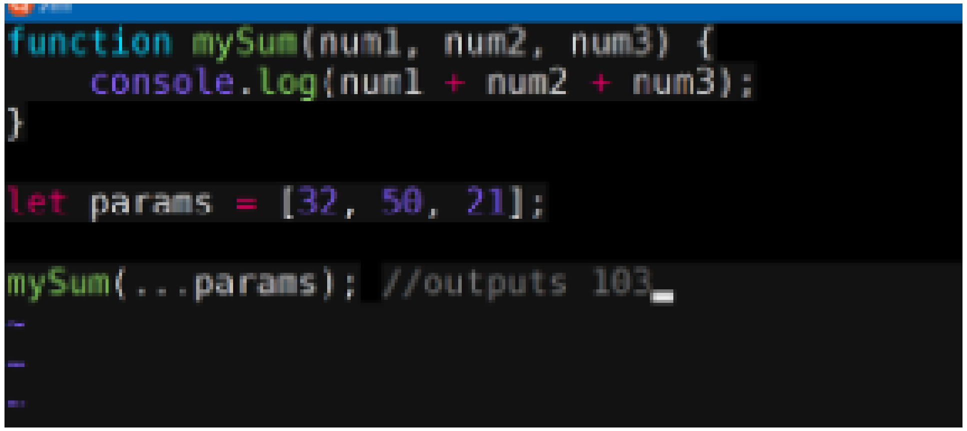
```
<!DOCTYPE html>
<html>
<body>

<h2>Default Parameter Values</h2>

<p id="demo"></p>

<script>
function myFunction(x, y = 10) {
  // y is 10 if not passed or undefined
  return x + y;
}
document.getElementById("demo").innerHTML = myFunction(5);
</script>

</body>
</html>
```

REST & SPREAD parameter

The rest parameter (...) allows a function to treat an indefinite number of arguments as an array

```
function sum(...args) {
  let sum = 0;
  for (let arg of args) sum += arg;
  return sum;
}

let x = sum(4, 9, 16, 25, 29, 100, 66, 77);
```

the spread operator (...) takes an array (or any iterable) and spreads it values



```
function mySum(num1, num2, num3) {
    console.log(num1 + num2 + num3);
}

let params = [32, 50, 21];

mySum(...params); //outputs 103
~
~
~
```

Modules

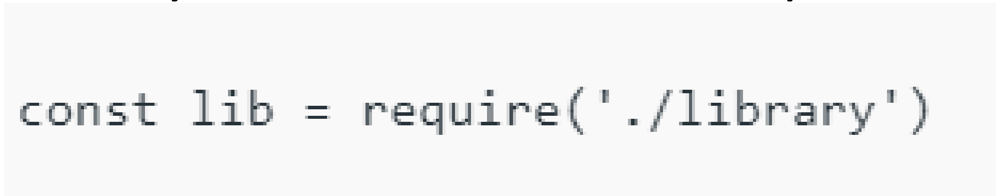
JavaScript Modules are basically libraries which are included in the given program. They are used for connecting two JavaScript programs together to call the functions written in one program without writing the body of the functions itself in another program

- **Exporting a library**

There is a special object in JavaScript called module.exports. When some program include or import this module (program), this object will be exposed.

- **Importing a library**

It means include a library in a program so that use the function is defined in that library. For this, use ‘require’ function in which pass the library name with its relative path



```
const lib = require('./library')
```

Global functions

Everything in JS is bound to containing scope. Therefore, if you define a function directly in file, it will be bound to window object, i.e. it will be global.

To make it "private", you have to create an object, which will contain these functions. You are correct that littering global scope is bad, but you have to put something in global scope to be able to access it, JS libraries do the same and there is no other workaround. But think about what you put in global scope, a single object should be more than enough for your "library".

Global functions

Example:

```
MyObject = {
    abc: function(...) {...},
    pqr: function(...) {...}
    // other functions...
}
```

To call abc for somewhere, be it same file or another file:

```
MyObject.abc(...);
```

Promises

Promises are used to handle asynchronous operations in JavaScript. They are easy to manage when dealing with multiple asynchronous operations where callbacks can create callback hell leading to unmanageable code.

- Prior to promises events and callback functions were used but they had limited functionalities and created unmanageable code.
- Multiple callback functions would create callback hell that leads to unmanageable code.
- Events were not good at handling asynchronous operations.

Promises

Benefits of Promises

- Improves Code Readability
- Better handling of asynchronous operations
- Better flow of control definition in asynchronous logic
- Better Error Handling

Promise has four states:

- **fulfilled**: Action related to the promise succeeded
- **rejected**: Action related to the promise failed
- **pending**: Promise is still pending i.e not fulfilled or rejected yet
- **settled**: Promise has fulfilled or rejected

A promise can be created using Promise constructor.
Syntax

```
var promise = new Promise(function(resolve, reject){
    //do something
});
```

Promises

A promise is an object that may produce a single value some time in the future: either a resolved value, or a reason that it’s not resolved (e.g., a network error occurred).

A promise may be in one of 3 possible states: fulfilled, rejected, or pending.
A promise is an object which can be returned synchronously from an asynchronous function. It will be in one of 3 possible states:

- Fulfilled: onFulfilled() will be called (e.g., resolve() was called)
- Rejected: onRejected() will be called (e.g., reject() was called)
- Pending: not yet fulfilled or rejected

Promises Cosumers

Promise Consumers

Promises can be consumed by registering functions using .then and .catch methods.

- then() is invoked when a promise is either resolved or rejected.

Parameters:

- then() method takes two functions as parameters.

First function is executed if promise is resolved and a result is received.

Second function is executed if promise is rejected and an error is received. (It is optional and there is a better way to hanlde error using .catch() method

Syntax:

```
.then(function(result){
      //handle success
   }, function(error){
      //handle error
   })
```

Async and Await

Javascript is a Synchronous which means that it has an event loop that allows you to queue up an action that won't take place until the loop is available sometime after the code that queued the action has finished executing.

But there's a lot of functionalities in our program which makes our code Asynchronous.

Async/Await is the extension of promises