ES6 Features

Harshita Maheshwari

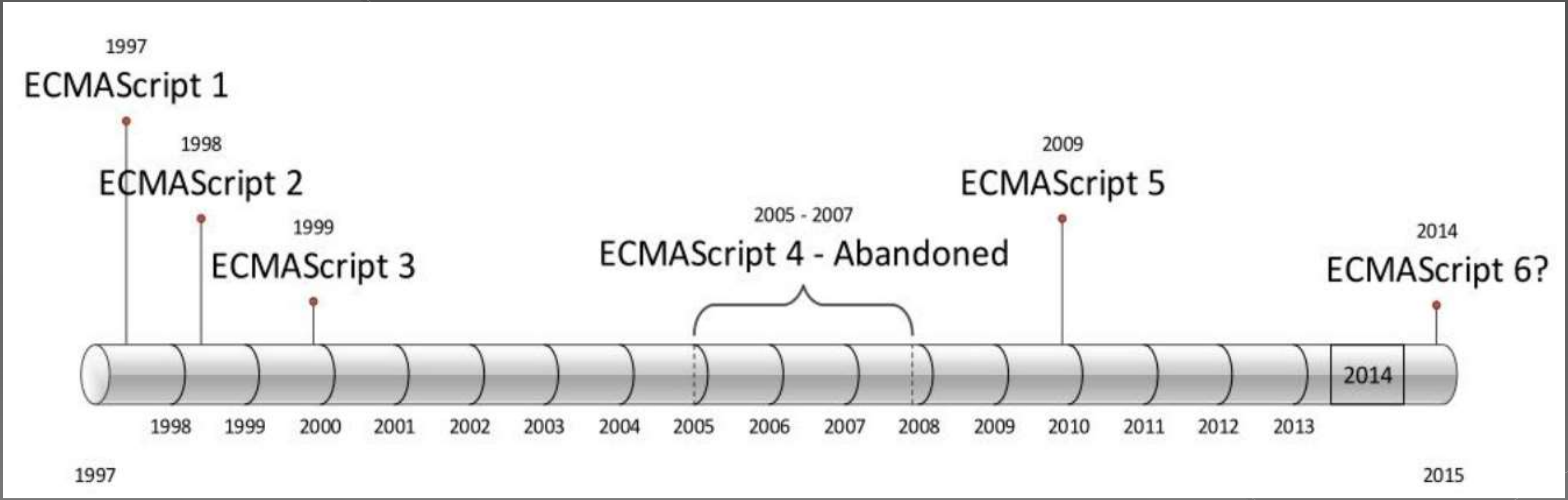
ECMA Script6

JavaScript ES6 (also known as ECMAScript 2015 or ECMAScript 6) is the newer version of JavaScript that was introduced in 2015.

ECMAScript is the standard that JavaScript programming language uses. ECMAScript provides the specification on how JavaScript programming language should work.

JavaScript ES6 brings new syntax and new awesome features to make your code more modern and more readable. It allows you to write less code and do more.

ECMA Script History

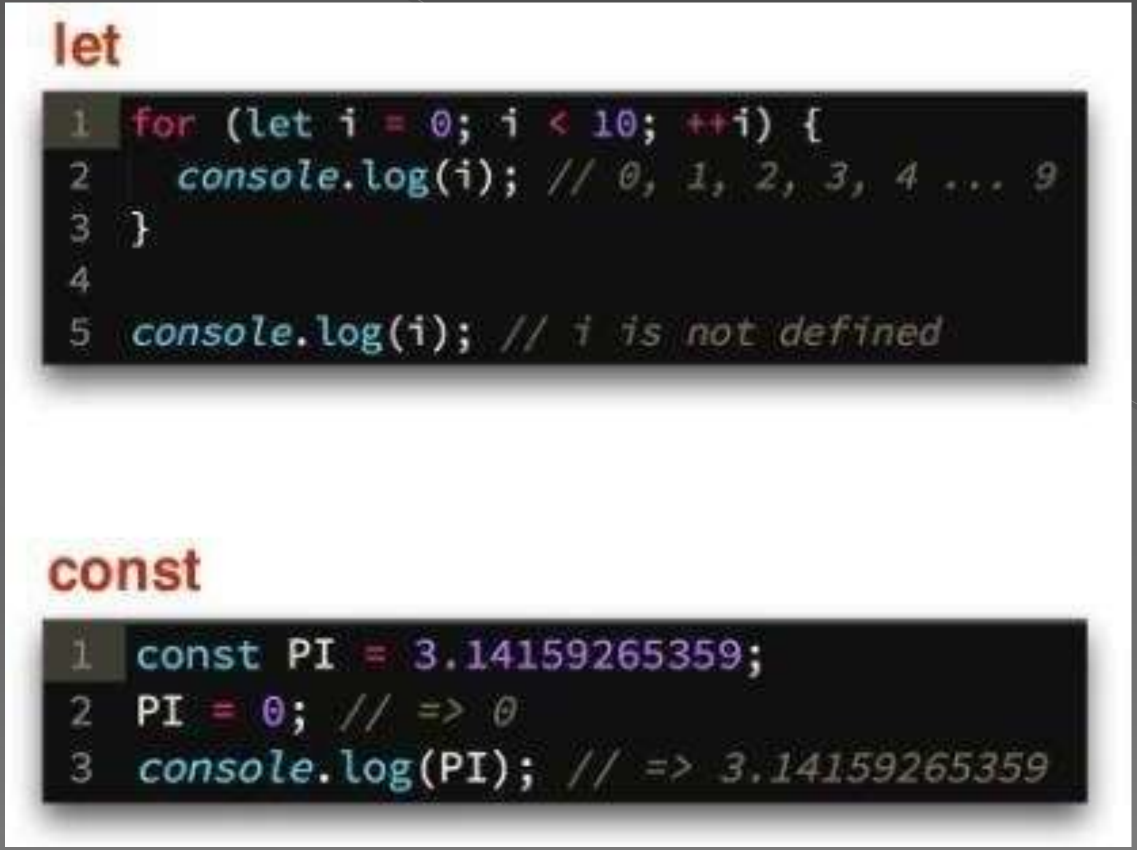


Features

1. let and const keywords
2. Arrow Functions
3. Multi-line Strings
4. Template Literals
5. Default Parameters
6. Rest Parameter and Spread Operators
7. Destructuring Assignment
8. for of loop
9. Classes
10. Inheritance
11. Modules

let and const

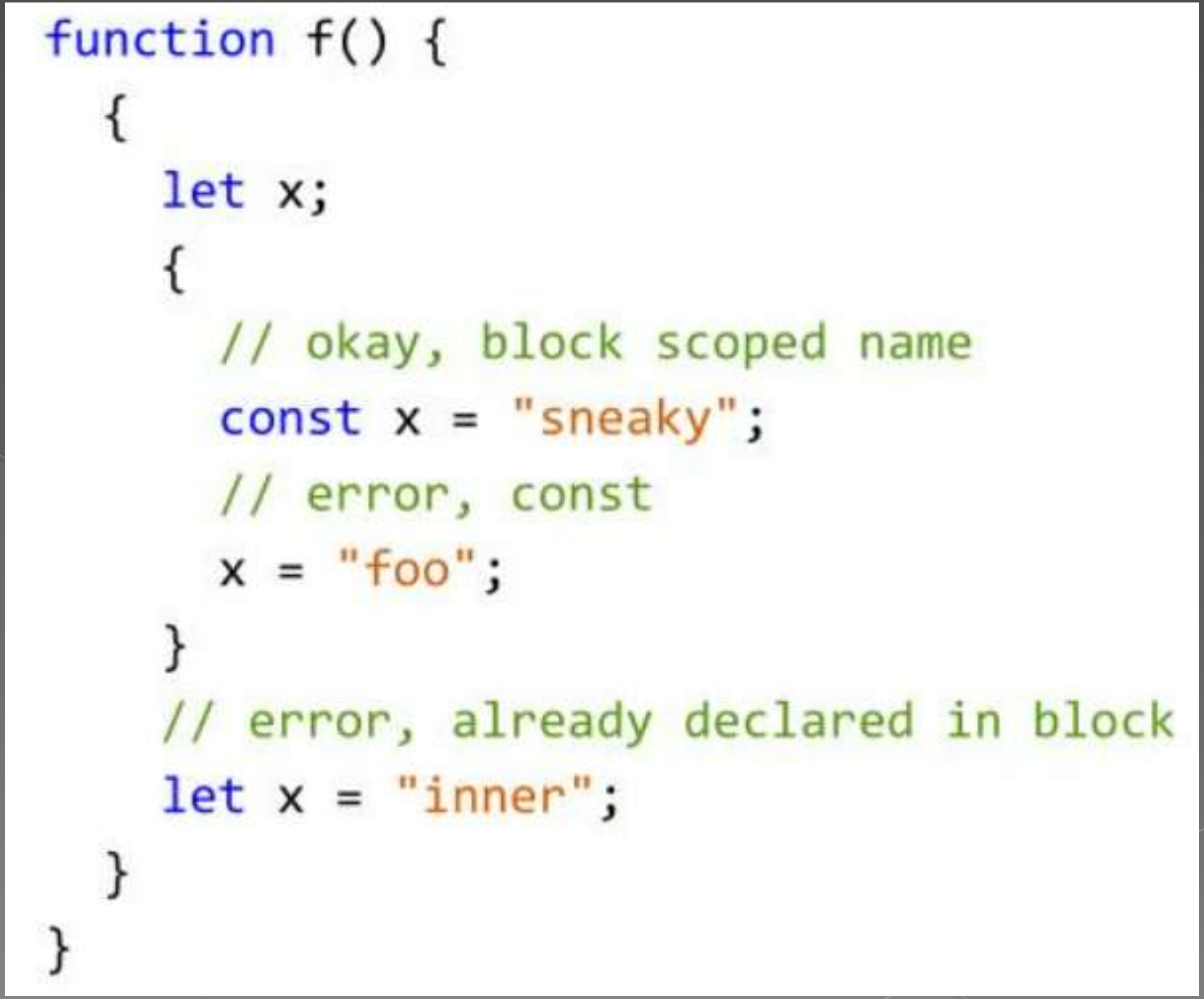
var	let	const
Function scope	Block scope	Block scope
Hoisted	Not Hoisted	Not Hoisted
Can declare and initialization can be done later	Can declare and initialization can be done later	Must be initialized at the time of declaration
Can change	Can change	Read-only
Use for top level variables	Use for localized variable in smaller scope	Use for read-only/fixed values


let

```
for (let i = 0; i < 10; ++i) {
  console.log(i); // 0, 1, 2, 3, 4 ... 9
}

console.log(i); // i is not defined
```

const

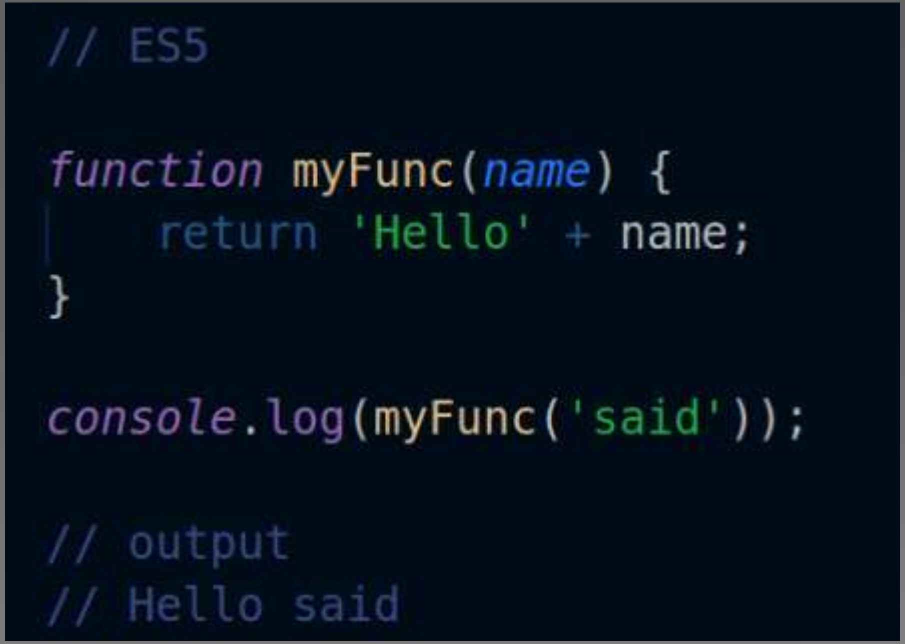
```
const PI = 3.14159265359;
PI = 0; // => 0
console.log(PI); // => 3.14159265359
```



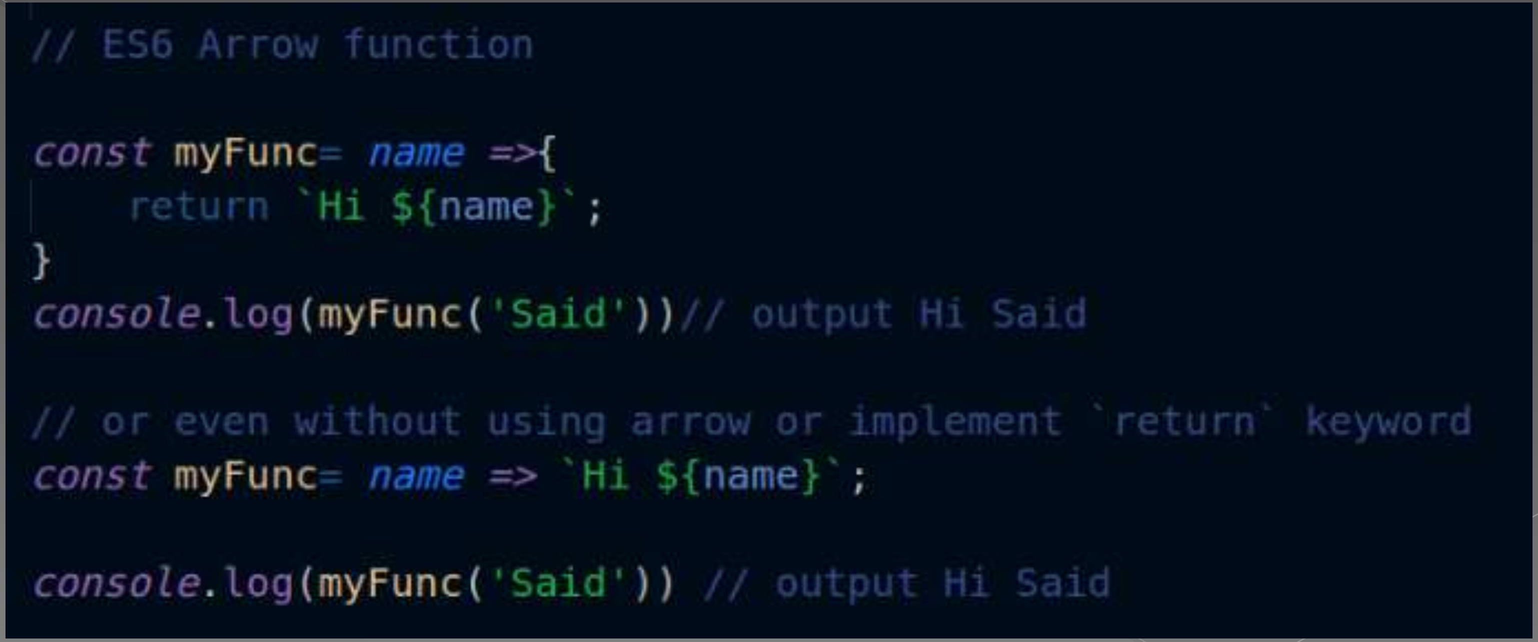
```
function f() {
  {
    let x;
    {
      // okay, block scoped name
      const x = "sneaky";
      // error, const
      x = "foo";
    }
    // error, already declared in block
    let x = "inner";
  }
}
```

Arrow Functions

Shorthand for Function Expression.
Makes your code more readable, more structured, and look like modern code.

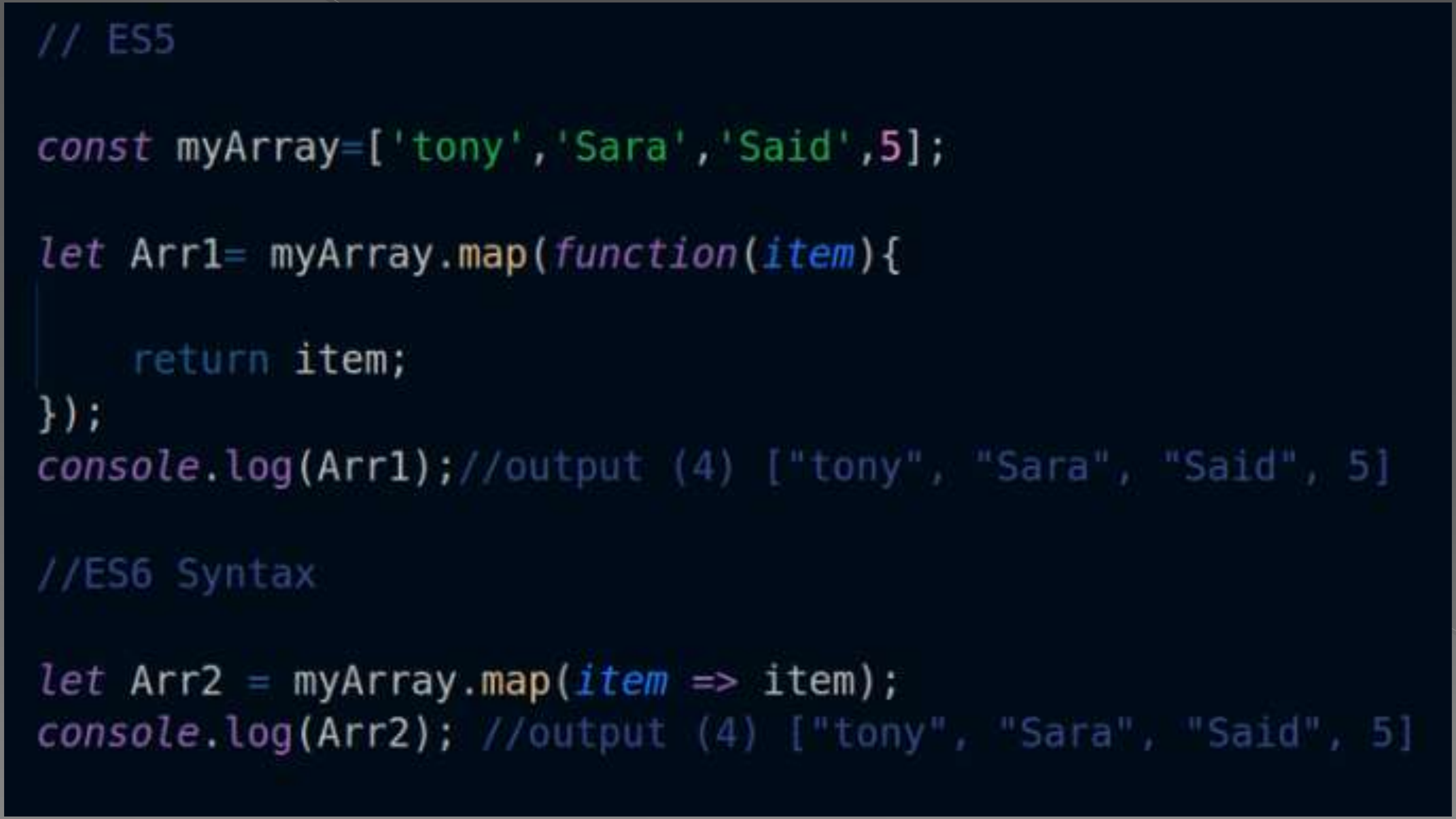


```javascript
// ES5

function myFunc(name) {
    return 'Hello' + name;
}

console.log(myFunc('said'));

// output
// Hello said
```



```javascript
// ES6 Arrow function

const myFunc= name =>{
    return `Hi ${name}`;
}
console.log(myFunc('Said'))// output Hi Said

// or even without using arrow or implement `return` keyword
const myFunc= name => `Hi ${name}`;

console.log(myFunc('Said')) // output Hi Said
```

We can use Arrow function with map, filter, and reduce built-in functions.



```
// ES5

const myArray=['tony','Sara','Said',5];

let Arr1= myArray.map(function(item){

    return item;
});
console.log(Arr1);//output (4) ["tony", "Sara", "Said", 5]

//ES6 Syntax

let Arr2 = myArray.map(item => item);
console.log(Arr2); //output (4) ["tony", "Sara", "Said", 5]
```

Multi-line Strings

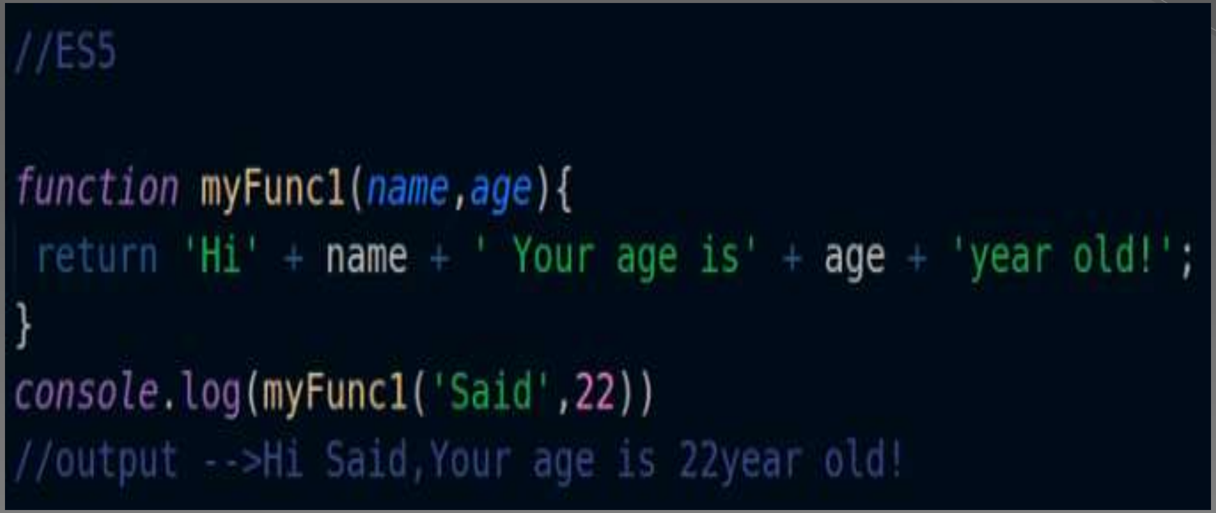
We can create multi-line strings by using back-ticks(`).

It can be done as shown below :

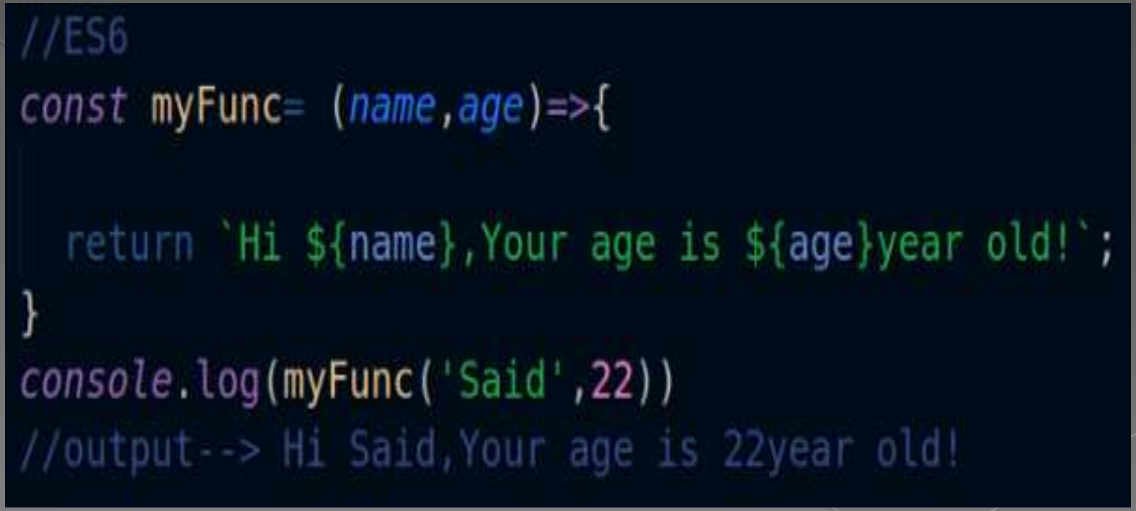
```
let greeting = `Hello World,
                Greetings to all,
                Keep Learning and
                Practicing!`
```

Template Literals

Syntactic sugar for string construction.
We don't have to use the plus (+) operator to concatenate strings, or when we want to use a variable inside a string.



```
//ES5

function myFunc1(name,age){
 return 'Hi' + name + ' Your age is' + age + 'year old!';
}
console.log(myFunc1('Said',22))
//output -->Hi Said,Your age is 22year old!
```



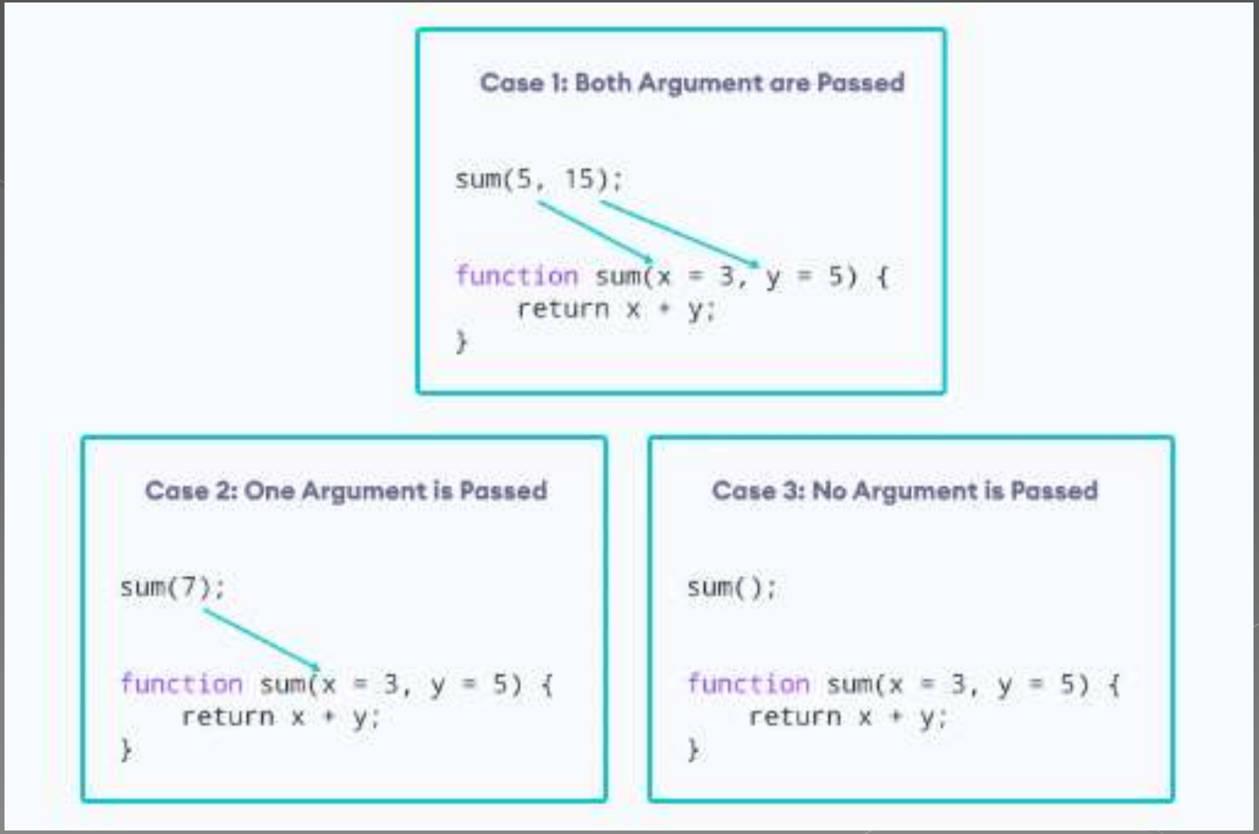
```
//ES6
const myFunc= (name,age)=>{

  return `Hi ${name},Your age is ${age}year old!`;
}
console.log(myFunc('Said',22))
//output--> Hi Said,Your age is 22year old!
```

Default Parameter

Allows us to give default values to function parameters.

We can provide the default values right in the signature of the functions.

```
function sum(x = 3, y = 5) {

    // return sum
    return x + y;
}

console.log(sum(5, 15));  // 20
console.log(sum(7));        // 12
console.log(sum());          // 8
```



Case 1: Both Argument are Passed

```
sum(5, 15);

function sum(x = 3, y = 5) {
    return x + y;
}
```

Case 2: One Argument is Passed

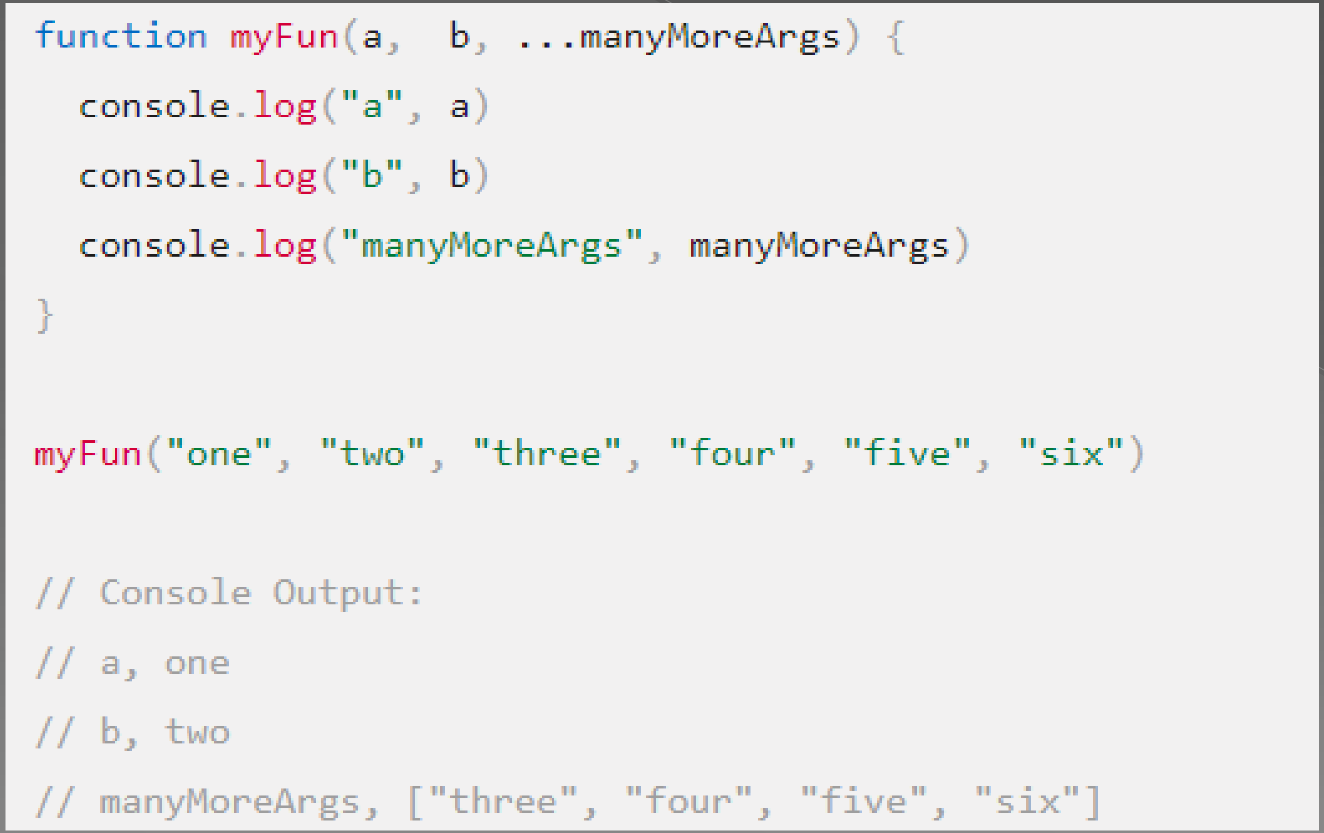
```
sum(7);

function sum(x = 3, y = 5) {
    return x + y;
}
```

Case 3: No Argument is Passed

```
sum();

function sum(x = 3, y = 5) {
    return x + y;
}
```

Rest parameter

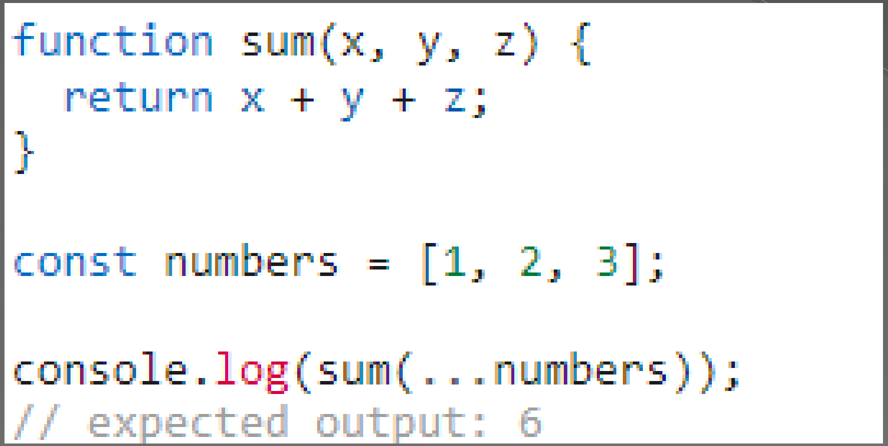
The rest parameters allows a function to accept an indefinite number of arguments as an array.



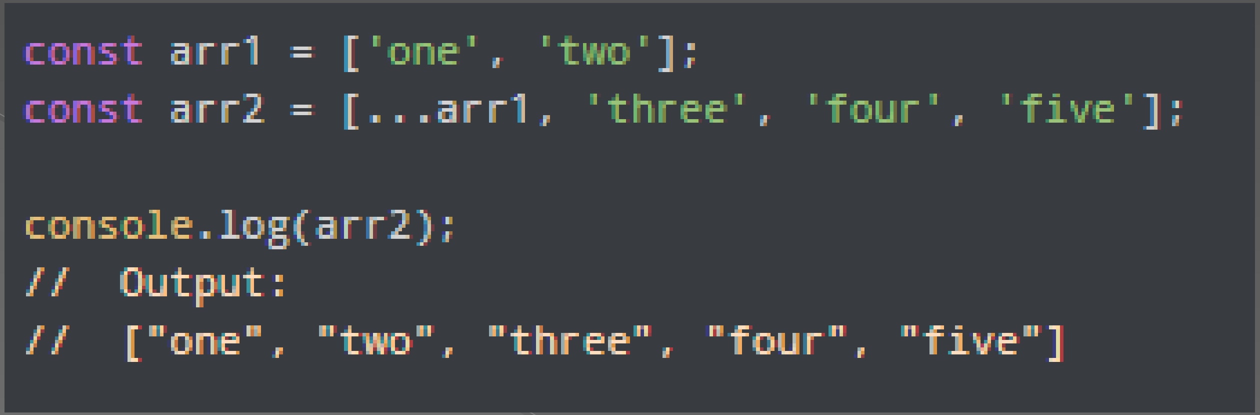
```
function myFun(a,  b, ...manyMoreArgs) {
  console.log("a", a)
  console.log("b", b)
  console.log("manyMoreArgs", manyMoreArgs)
}

myFun("one", "two", "three", "four", "five", "six")

// Console Output:
// a, one
// b, two
// manyMoreArgs, ["three", "four", "five", "six"]
```

Spread Operator

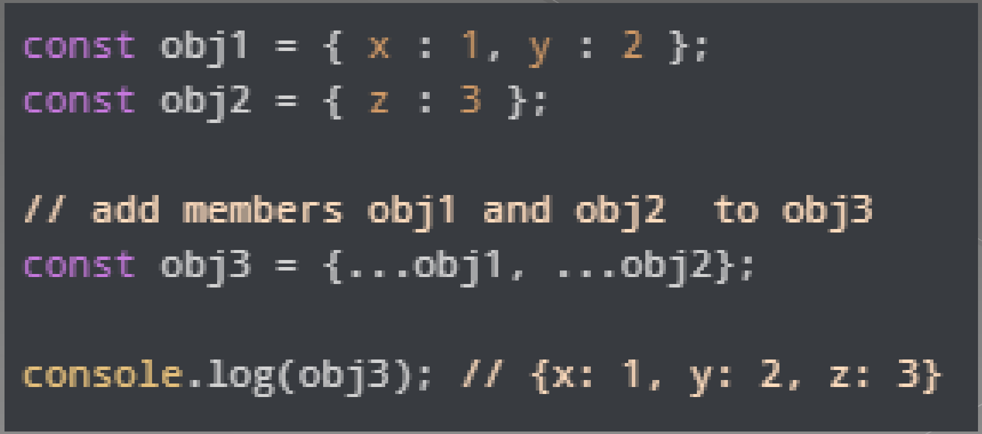
The spread operator … is used to expand or spread an iterable or an array.



```
function sum(x, y, z) {
  return x + y + z;
}

const numbers = [1, 2, 3];

console.log(sum(...numbers));
// expected output: 6
```



```
const arr1 = ['one', 'two'];
const arr2 = [...arr1, 'three', 'four', 'five'];

console.log(arr2);
//  Output:
//  ["one", "two", "three", "four", "five"]
```



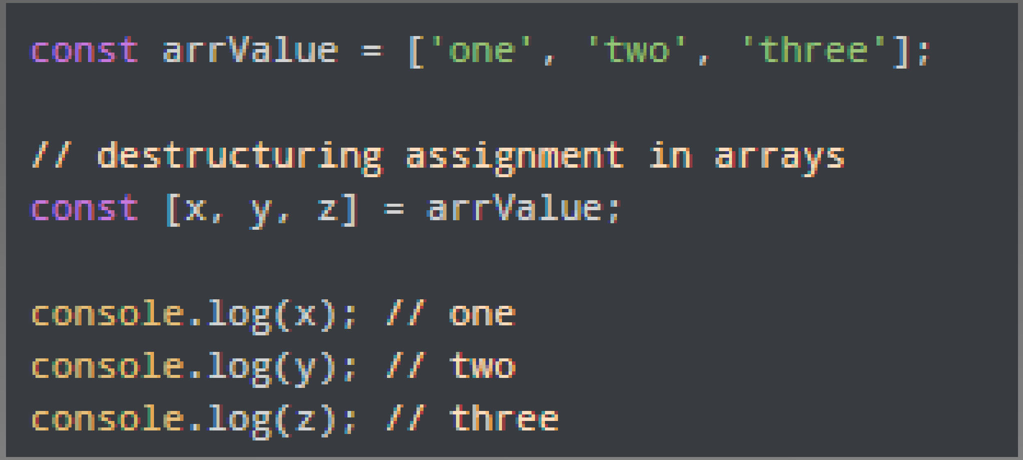
```
const obj1 = { x : 1, y : 2 };
const obj2 = { z : 3 };

// add members obj1 and obj2  to obj3
const obj3 = {...obj1, ...obj2};

console.log(obj3); // {x: 1, y: 2, z: 3}
```

Destructuring Assignment

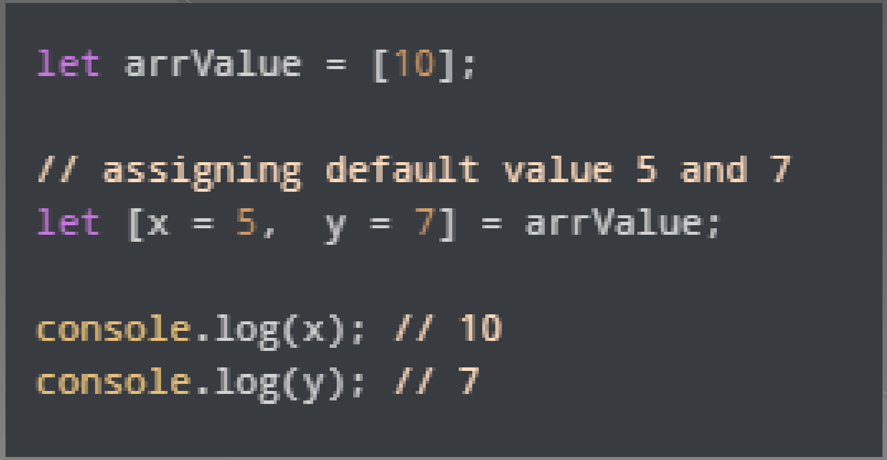
infoway TECHNOLOGIES

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

Array Destructuring :-

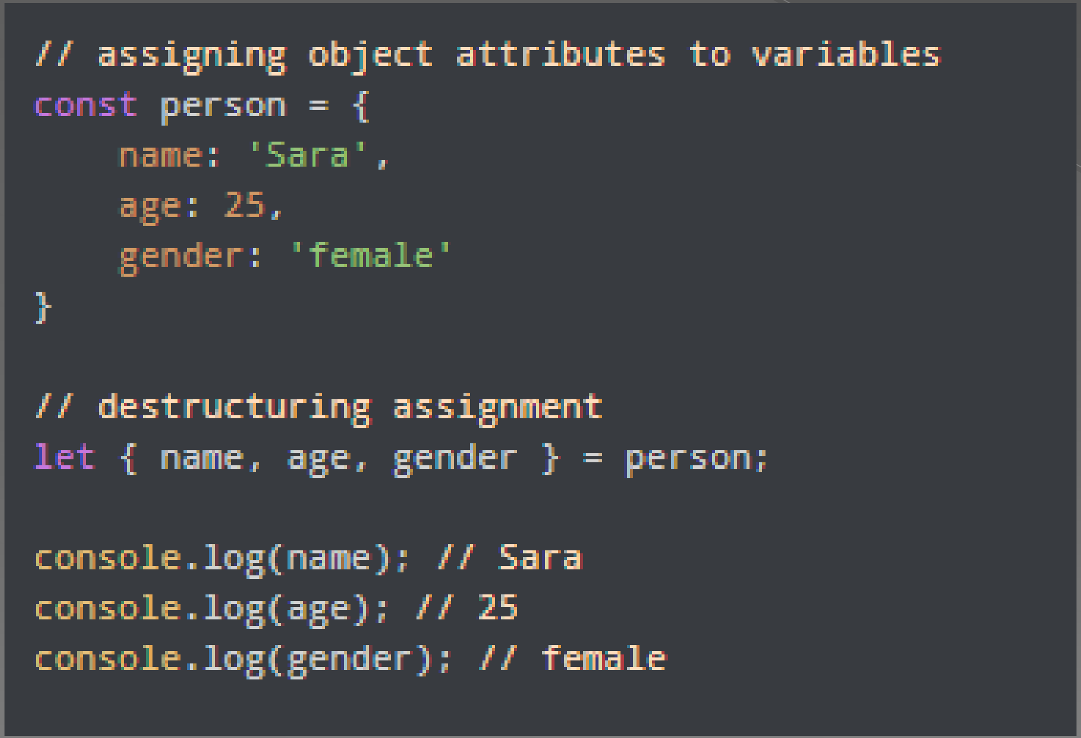


```javascript
const arrValue = ['one', 'two', 'three'];

// destructuring assignment in arrays
const [x, y, z] = arrValue;

console.log(x); // one
console.log(y); // two
console.log(z); // three
```

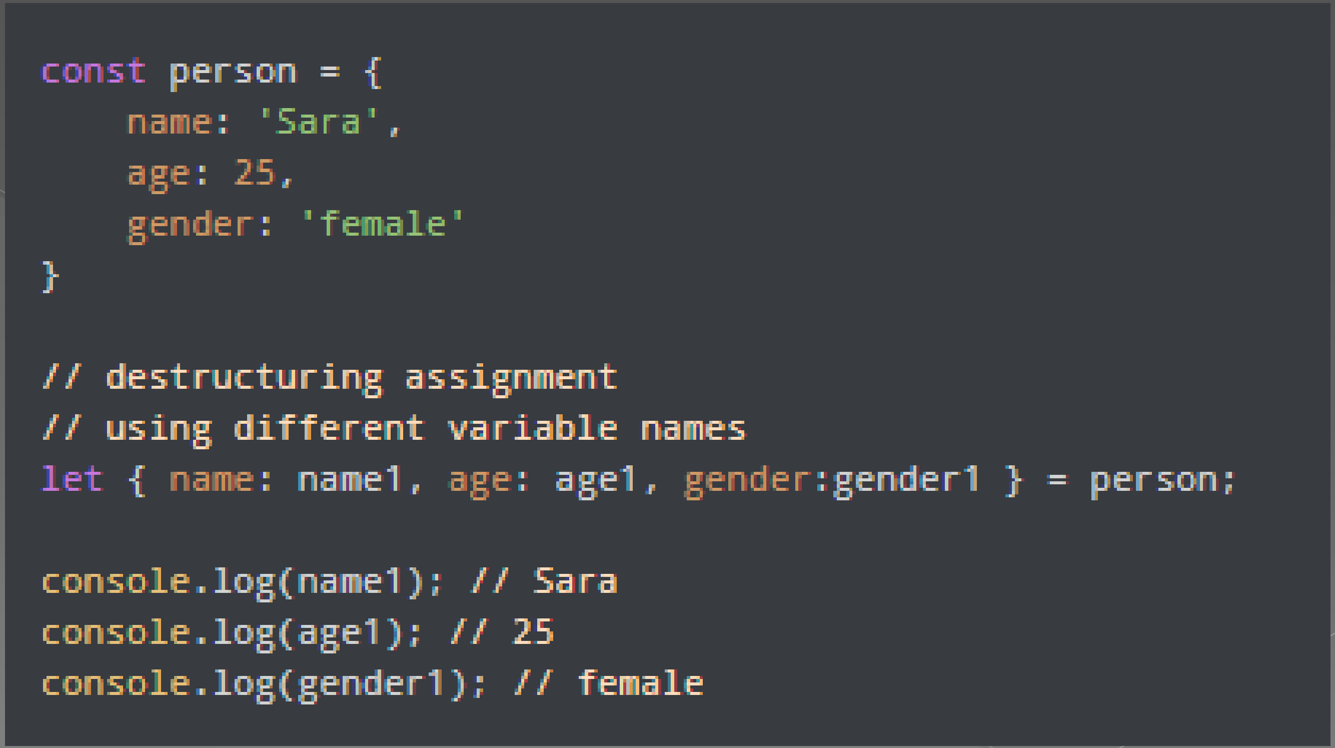


```javascript
let arrValue = [10];

// assigning default value 5 and 7
let [x = 5,  y = 7] = arrValue;

console.log(x); // 10
console.log(y); // 7
```

Object Destructuring -



```javascript
// assigning object attributes to variables
const person = {
    name: 'Sara',
    age: 25,
    gender: 'female'
}

// destructuring assignment
let { name, age, gender } = person;

console.log(name); // Sara
console.log(age); // 25
console.log(gender); // female
```

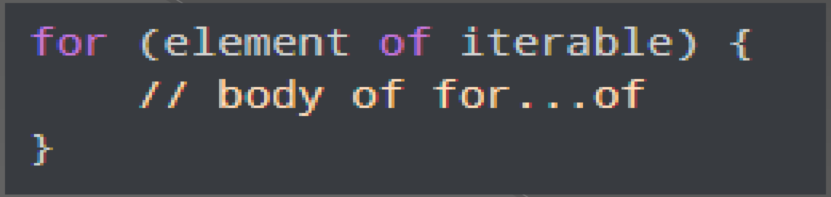


```javascript
const person = {
    name: 'Sara',
    age: 25,
    gender: 'female'
}

// destructuring assignment
// using different variable names
let { name: name1, age: age1, gender:gender1 } = person;

console.log(name1); // Sara
console.log(age1); // 25
console.log(gender1); // female
```

for of loop

Allows to iterate over iterable objects (arrays, strings etc).

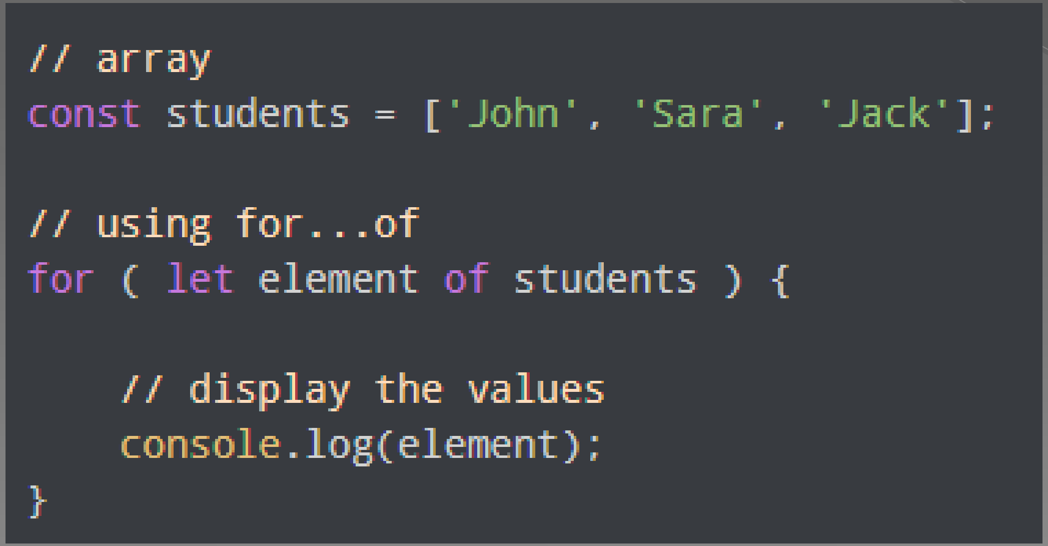
Syntax -



```
for (element of iterable) {
    // body of for...of
}
```

iterable - an iterable object (array, set, strings, etc).

element - items in the iterable



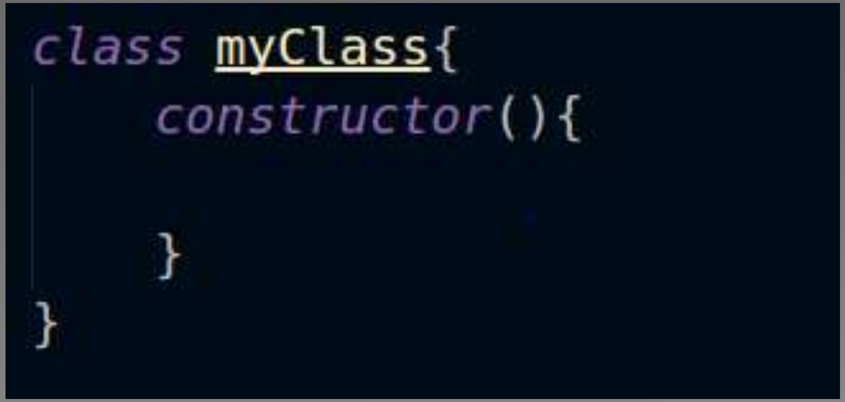
```
// array
const students = ['John', 'Sara', 'Jack'];

// using for...of
for ( let element of students ) {

    // display the values
    console.log(element);
}
```

class

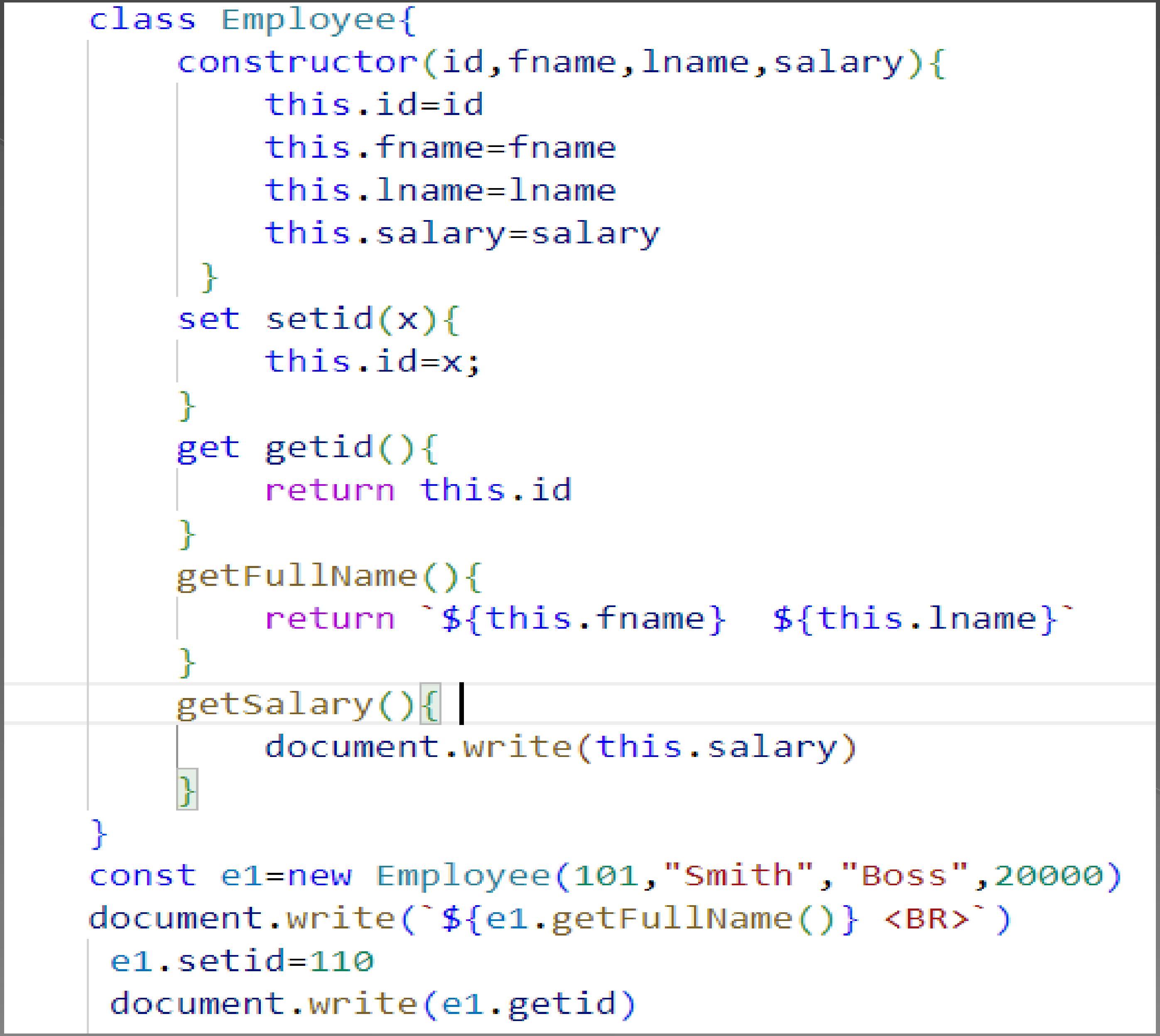
A Class is template/blueprint for creating objects.

Ex- A sketch (prototype) of a house is a class . It contains all the details about the floors, doors, windows, etc. Based on these descriptions, you build the house. House is the object.

Syntax -



```
class myClass{
    constructor(){

    }
}
```

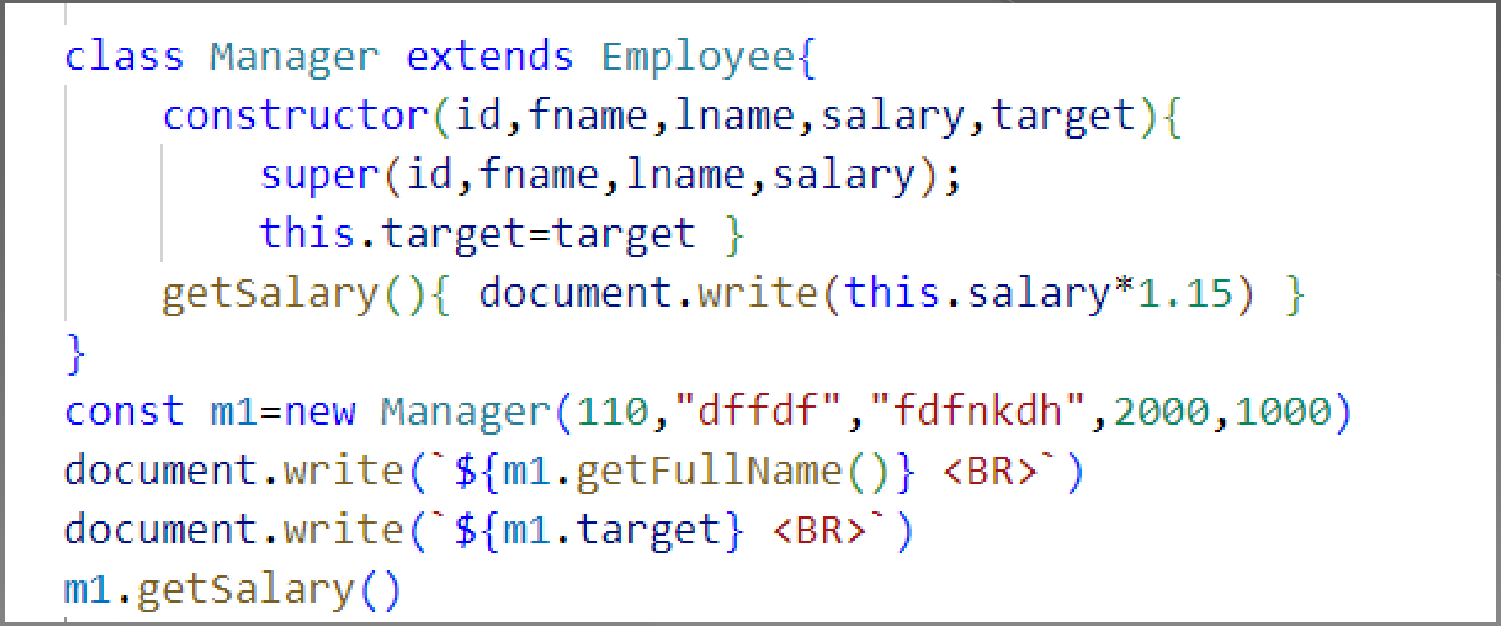


```
class Employee{
    constructor(id,fname,lname,salary){
        this.id=id
        this.fname=fname
        this.lname=lname
        this.salary=salary
     }
    set setid(x){
        this.id=x;
    }
    get getid(){
        return this.id
    }
    getFullName(){
        return `${this.fname}  ${this.lname}`
    }
    getSalary(){
        document.write(this.salary)
    }
}
const e1=new Employee(101,"Smith","Boss",20000)
document.write(`${e1.getFullName()} <BR>`)
 e1.setid=110
 document.write(e1.getid)
```

Inheritance

Inheritance enables you to define a class that takes all the functionality from a parent class and allows you to add more.

Reusability + Extensibility



```
class Manager extends Employee{
    constructor(id,fname,lname,salary,target){
        super(id,fname,lname,salary);
        this.target=target }
    getSalary(){ document.write(this.salary*1.15) }
}
const m1=new Manager(110,"dffdf","fdfnkdh",2000,1000)
document.write(`${m1.getFullName()} <BR>`)
document.write(`${m1.target} <BR>`)
m1.getSalary()
```

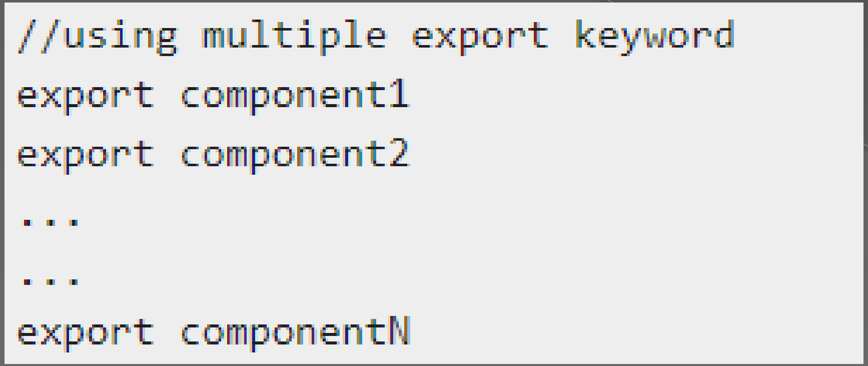
modules

A module is nothing more than a chunk of JavaScript code written in a file. By default, variables and functions of a module are not available for use. Variables and functions within a module should be exported so that they can be accessed from within other files. Modules in ES6 work only in strict mode.

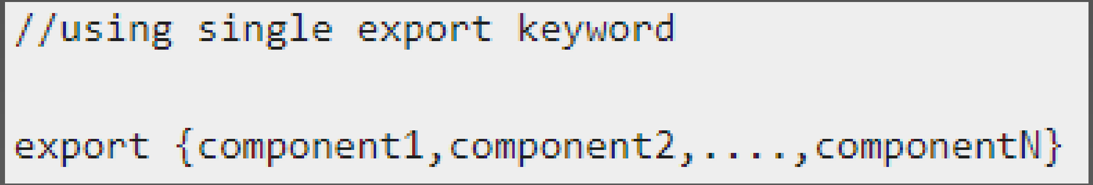
Exporting and importing a Module :-
Named Exports and imports
Default Exports and imports

Named export and import

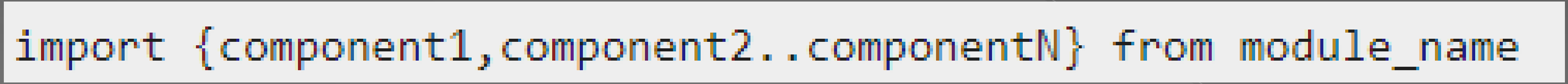
Named exports are distinguished by their names. There can be several named exports in a module.



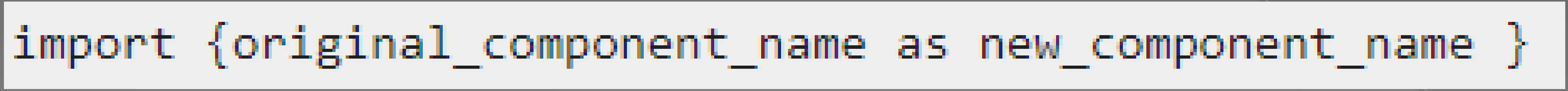
```
//using multiple export keyword
export component1
export component2
...
...
export componentN
```



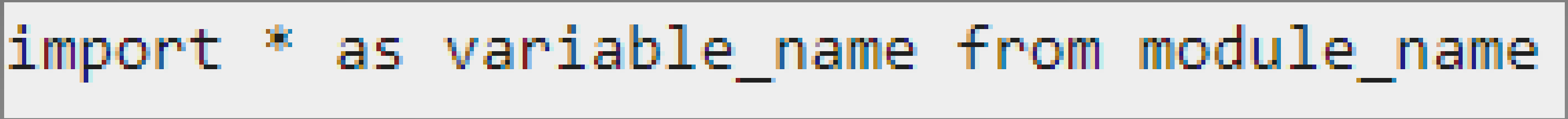
```
//using single export keyword

export {component1,component2,....,componentN}
```



```
import {component1,component2..componentN} from module_name
```



```
import {original_component_name as new_component_name }
```



```
import * as variable_name from module_name
```

Default export and import

Modules that need to export only a single value can use default exports. There can be only one default export per module.

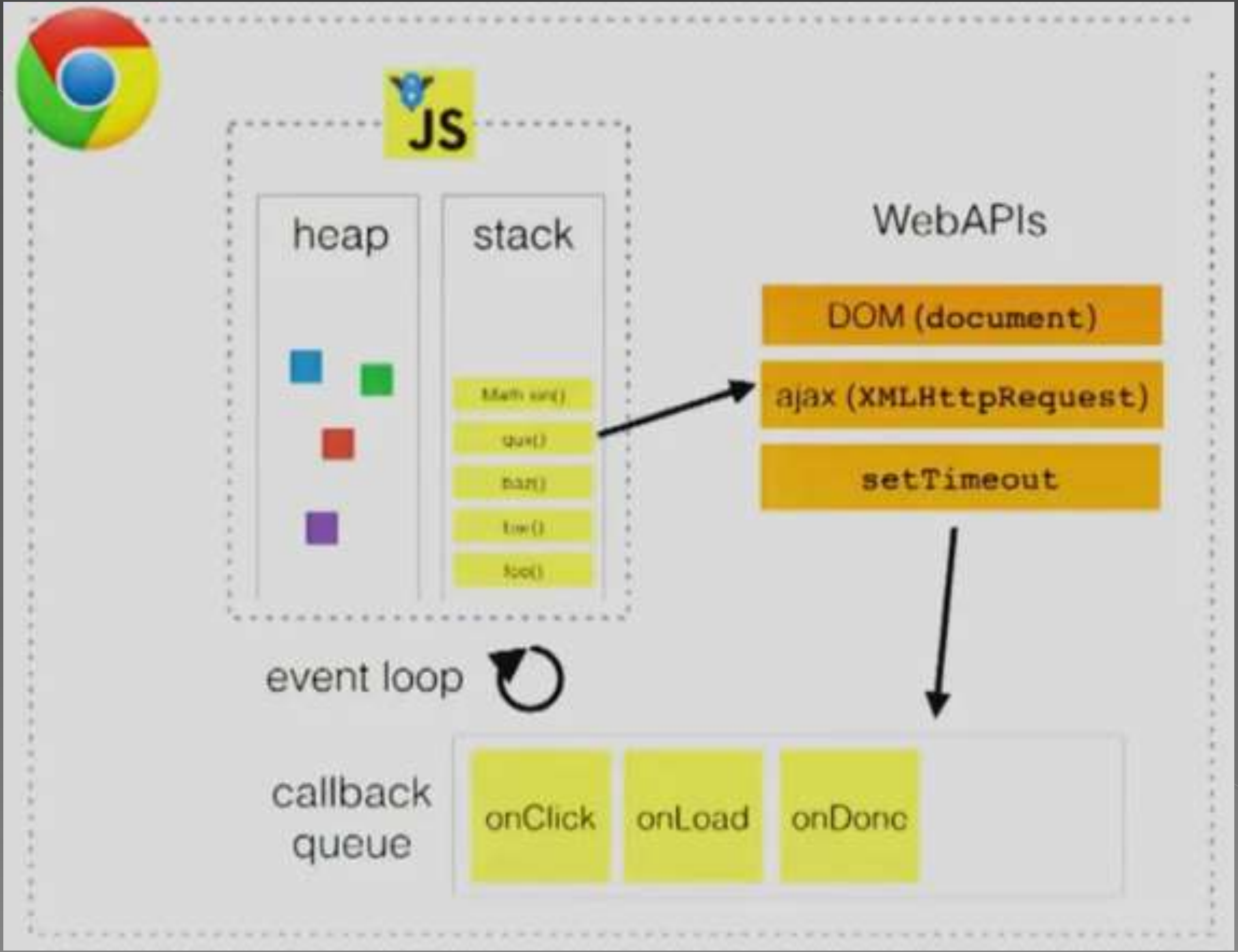
```
export default component_name
```

Unlike named exports, a default export can be imported with any name.

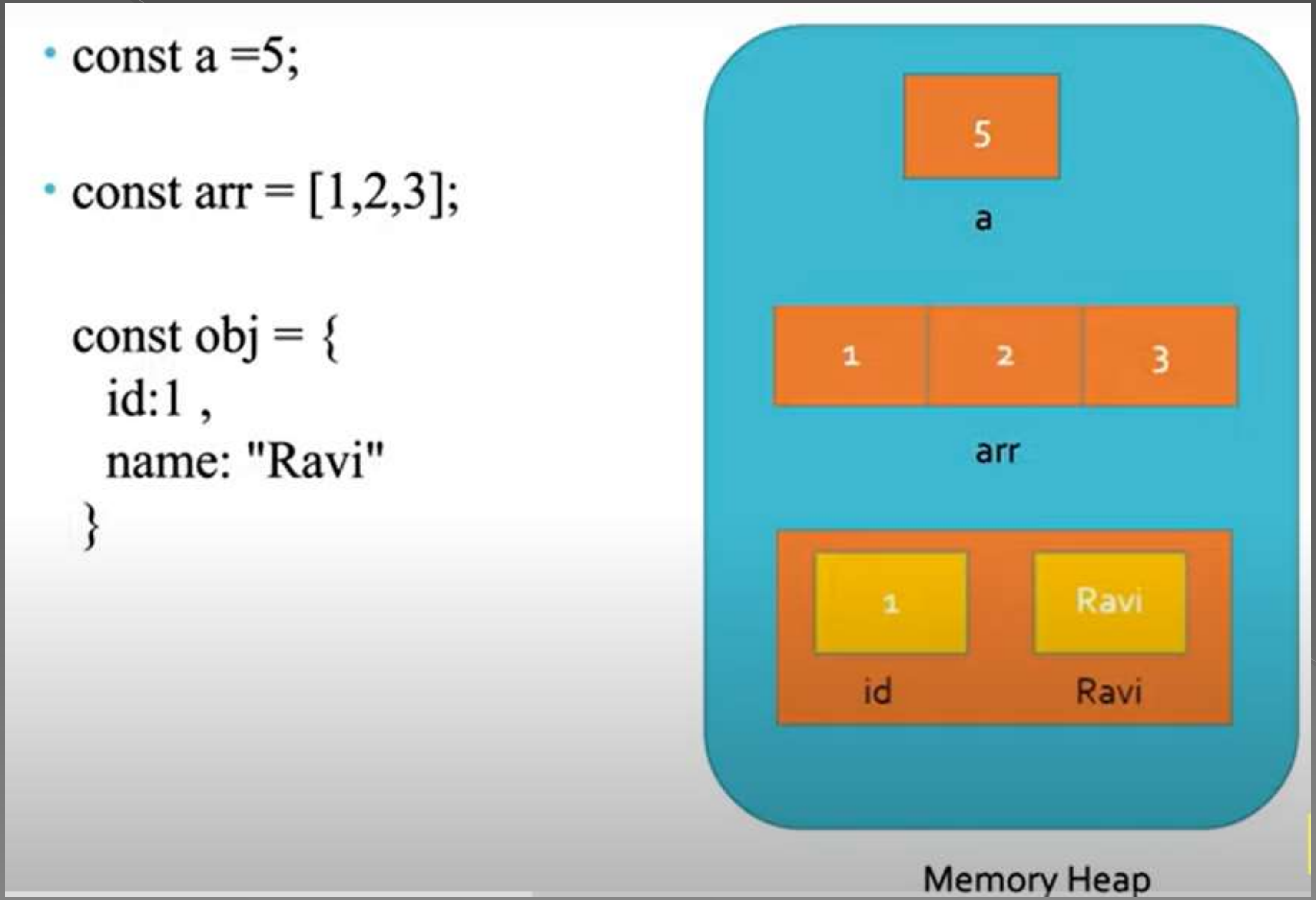
```
import any_variable_name from module_name
```

To execute both the modules we need to make a html file as shown below and run this in live server. Note that we should use the attribute type="module" in the script tag.

How JavaScript Works??


JS
heap
stack
Math.sin()
qux()
baz()
bar()
foo()
WebAPIs
DOM (document)
ajax (XMLHttpRequest)
setTimeout
event loop
callback queue
onClick
onLoad
onDone

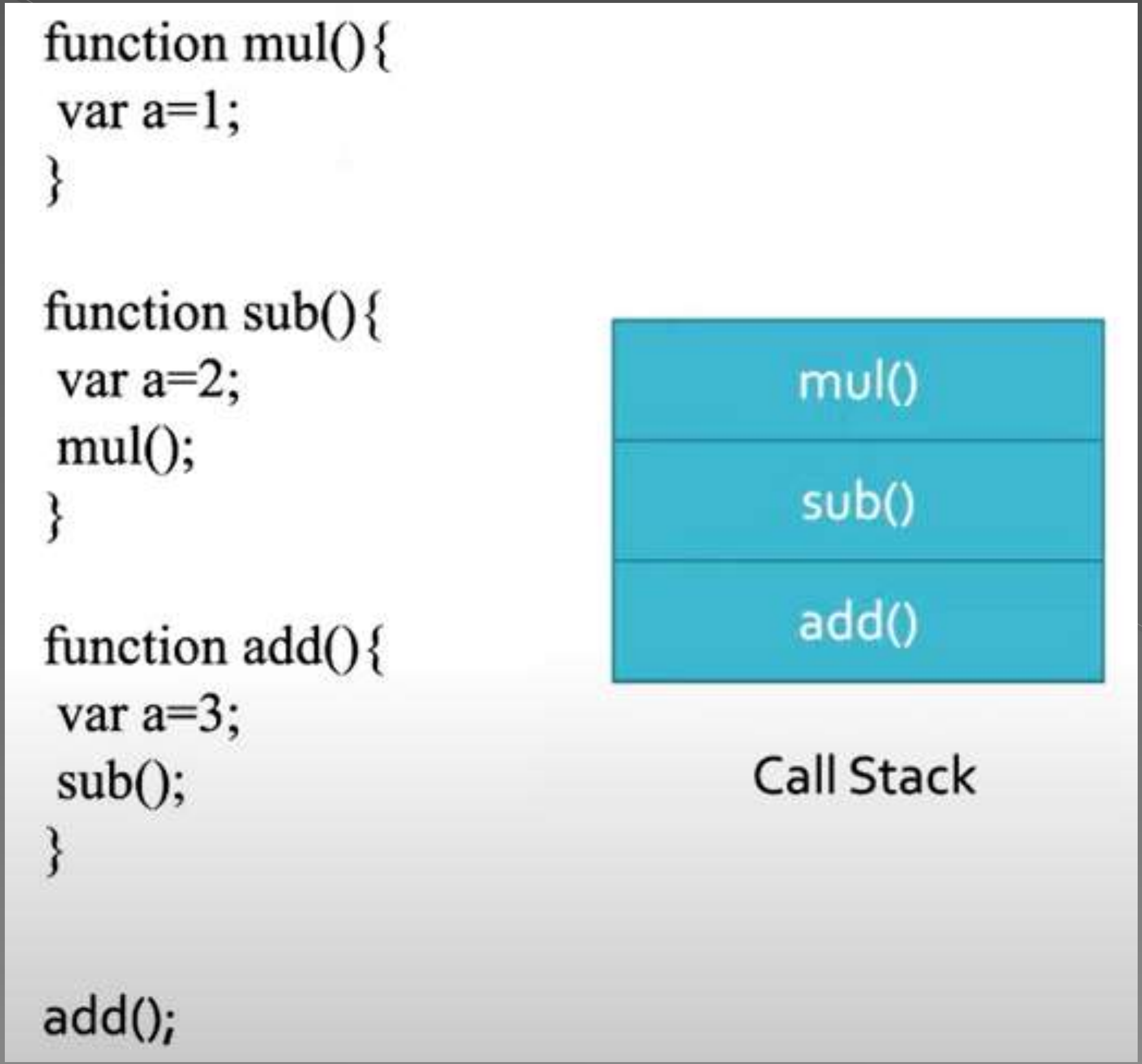
Memory Heap



- const a =5;
- const arr = [1,2,3];

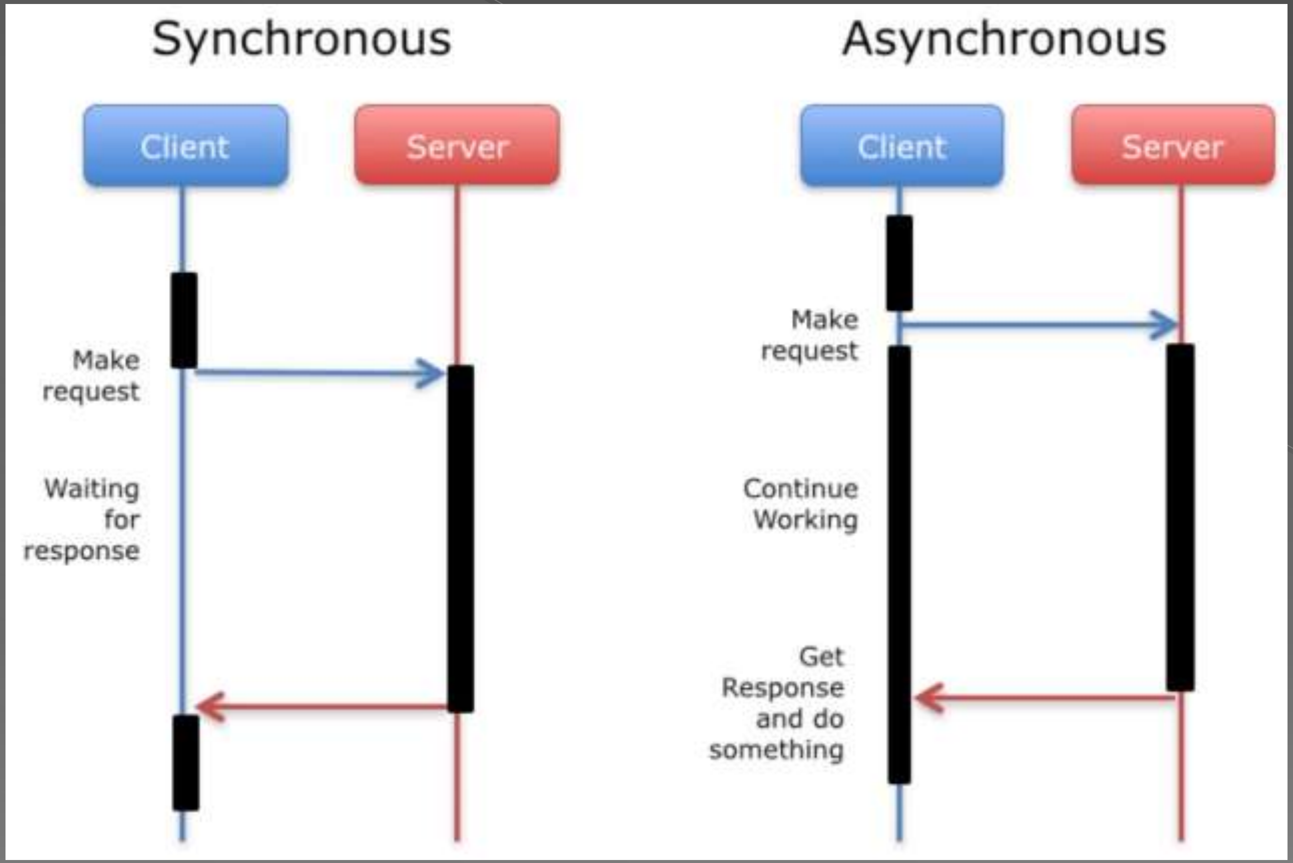
```
const obj = {
  id:1 ,
  name: "Ravi"
}
```

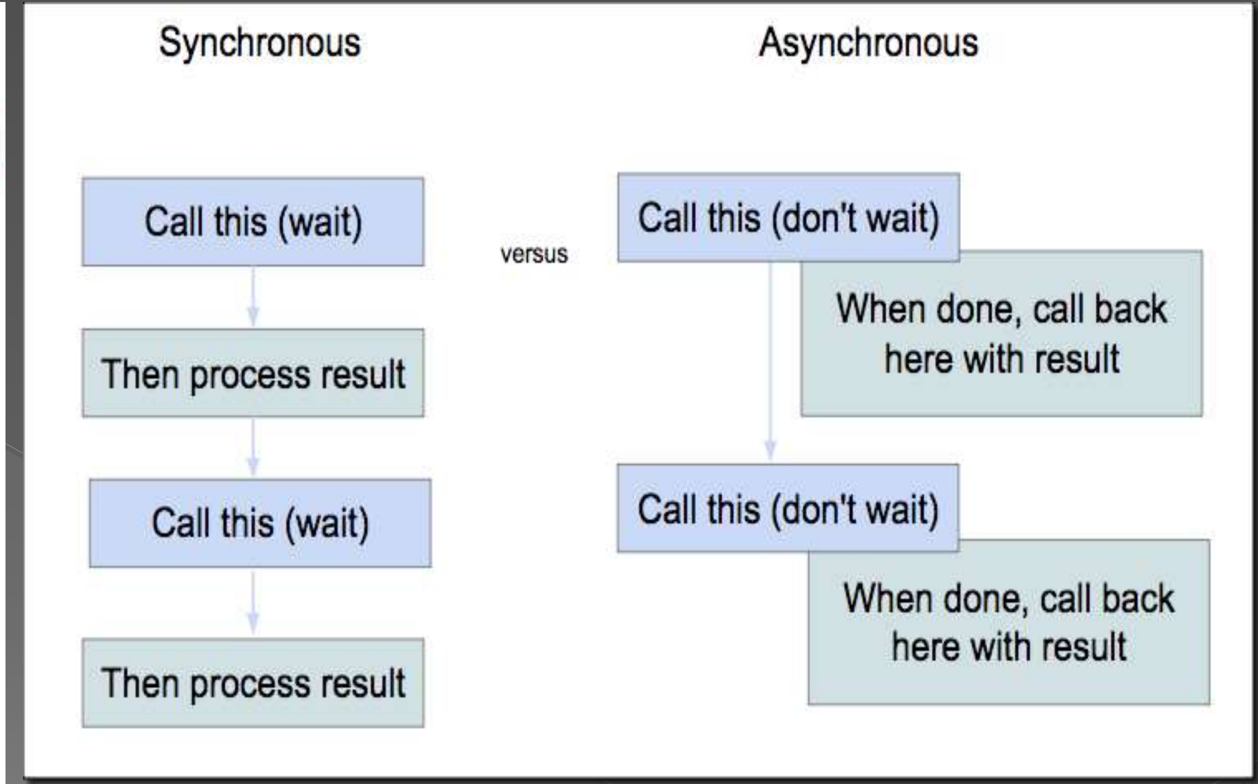
Call stack



```
function mul(){
 var a=1;
}

function sub(){
 var a=2;
 mul();
}

function add(){
 var a=3;
 sub();
}


add();
```

synchronous and asynchronous





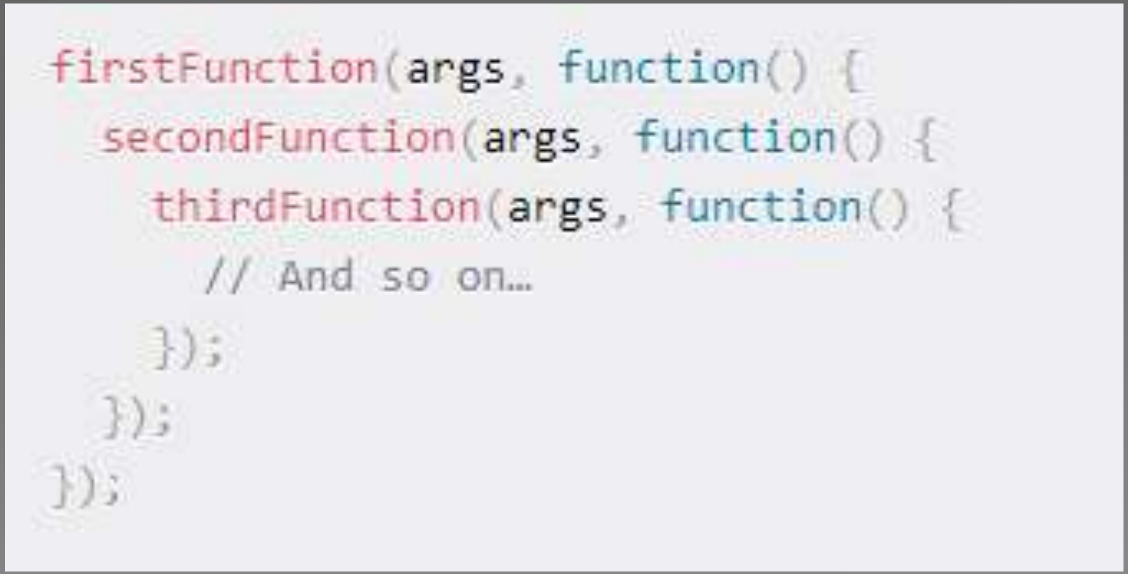
Callback

A callback is a function that is passed as an argument to another function that executes the callback based on the result. They are basically functions that are executed only after a result is produced.

Ex –
Event Handlers are sort of callbacks.
Setitmeout and setInterval takes a callback argument

Callback hell

Callback Hell is essentially nested callbacks stacked below one another forming a pyramid structure. Every callback depends/waits for the previous callback, thereby making a pyramid structure that affects the readability and maintainability of the code.



```
firstFunction(args, function() {
  secondFunction(args, function() {
    thirdFunction(args, function() {
      // And so on…
    });
  });
});
```

Solutions to callback hell

There are four solutions to callback hell:

1. Write comments
2. Split functions into smaller functions
3. Using Promises
4. Using async/await

Constructing a callback hell

Let’s imagine we’re trying to make a aloo tikki burger. To make a burger, we need to go through the following steps:

- Get ingredients
- Boil potato's
- Make aloo patties
- Get burger buns
- Put the cooked patties between the buns
- Serve the burger

Promise

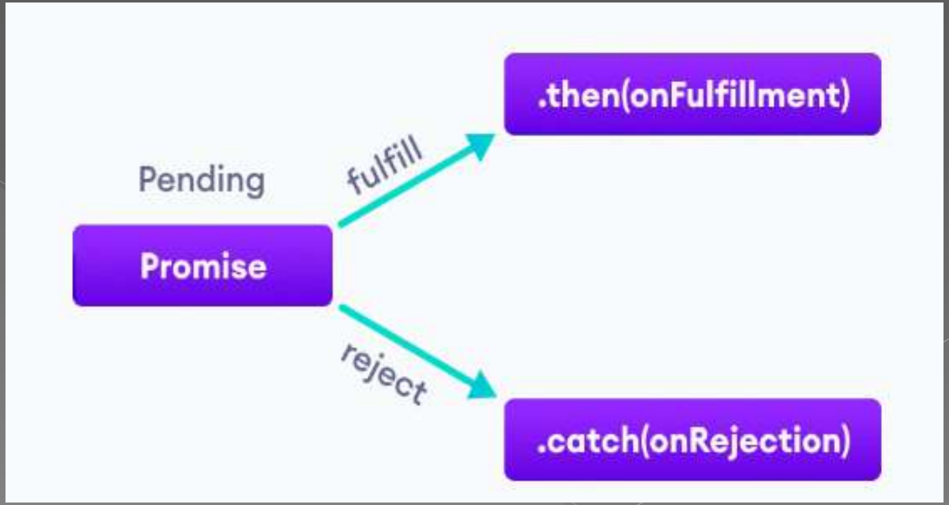
Promises can make callback hell much easier to manage.

It is an object that might return a value in the future.

It accomplishes the same basic goal as a callback function, but with many additional features and a more readable syntax.

A promise may have one of three states.

- Pending (unfulfilled yet, still being computed)
- Fulfilled (resolved , succeeded)
- Rejected (an error happened)



Settled or Pending

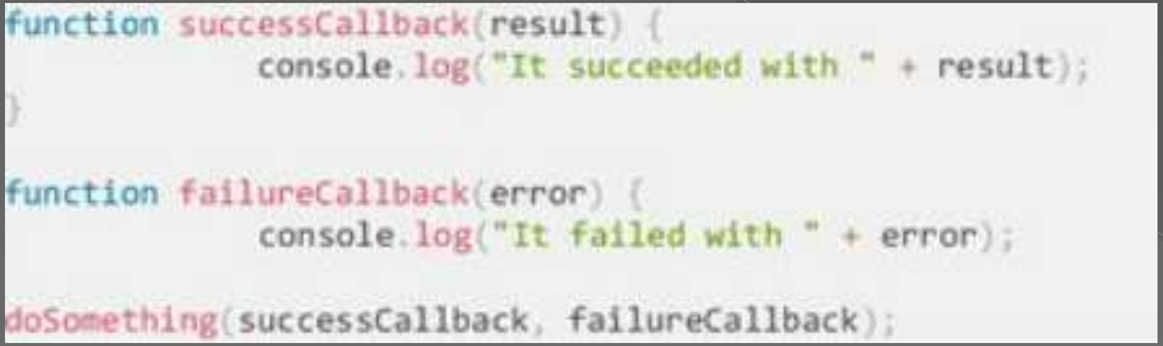
A promise is settled if it is not pending (it has been resolved or rejected).). Once a Promise has settled, it is settled for good. It cannot transition to any other state.

Promise users can attach callbacks to handle the fulfilled value or the reason for rejection.

Promise with callback

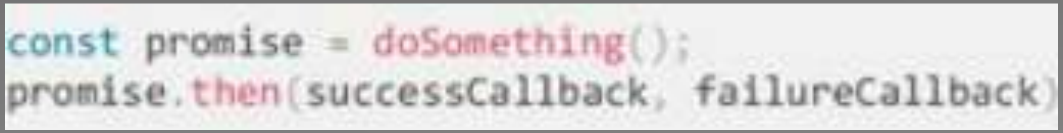
Essentially, a promise is a returned object to which you attach callbacks, instead of passing callbacks into a function.

E.g., instead of an old-style function that expects two callbacks, and calls one of them on eventual completion or failure:



```
function successCallback(result) {
            console.log("It succeeded with " + result);
}

function failureCallback(error) {
            console.log("It failed with " + error);

doSomething(successCallback, failureCallback);
```

modern functions return a promise you can attach your callbacks to instead:



```
const promise = doSomething();
promise.then(successCallback, failureCallback)
```

Create a Promise

```
const count = true;
let countValue = new Promise(function (resolve, reject) {
     if (count) {
          resolve("There is a count value.");
         }
    else {
          reject("There is no count value");
          }
});
```

Consume Promise

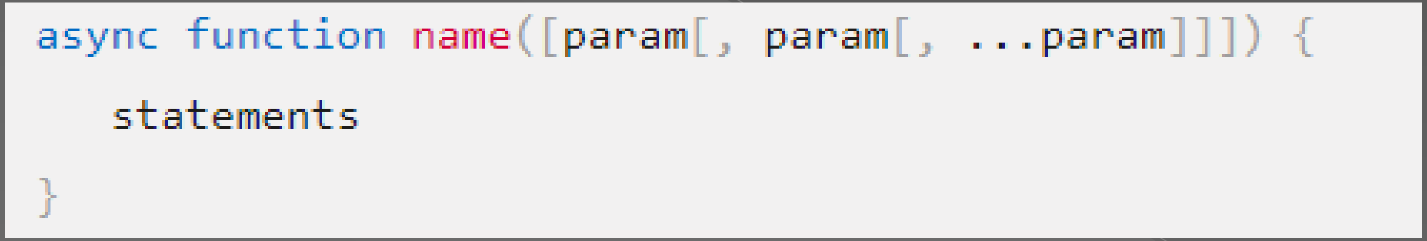
```
countValue.then((result)=> {// executes when promise is resolved successfully
    console.log(result);
})
.catch( (error)=> {      // executes if there is an error
    console.log(error);
});
```

Method	Description
then()	Handles a resolve. Returns a promise, and calls onFulfilled function asynchronously
catch()	Handles a reject. Returns a promise, and calls onRejected function asynchronously
finally()	Called when a promise is settled. Returns a promise, and calls onFinally function asynchronously

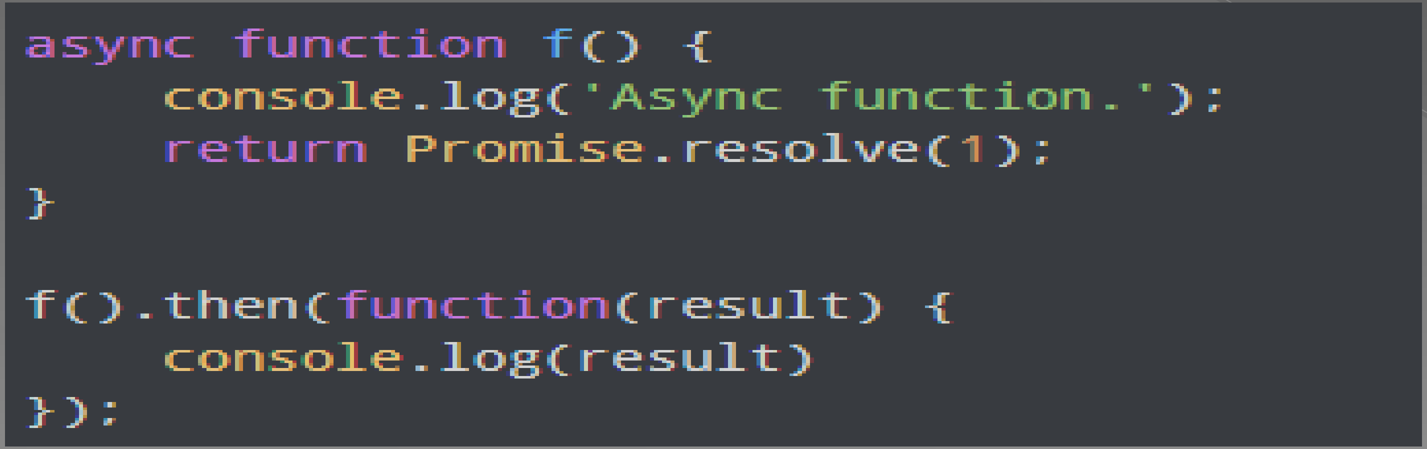
async function

An async function is a function declared with the async keyword, and the await keyword is permitted within it.
The async and await keywords enable asynchronous, promise-based behavior to be written in a cleaner style, avoiding the need to explicitly configure promise chains.

Syntax -



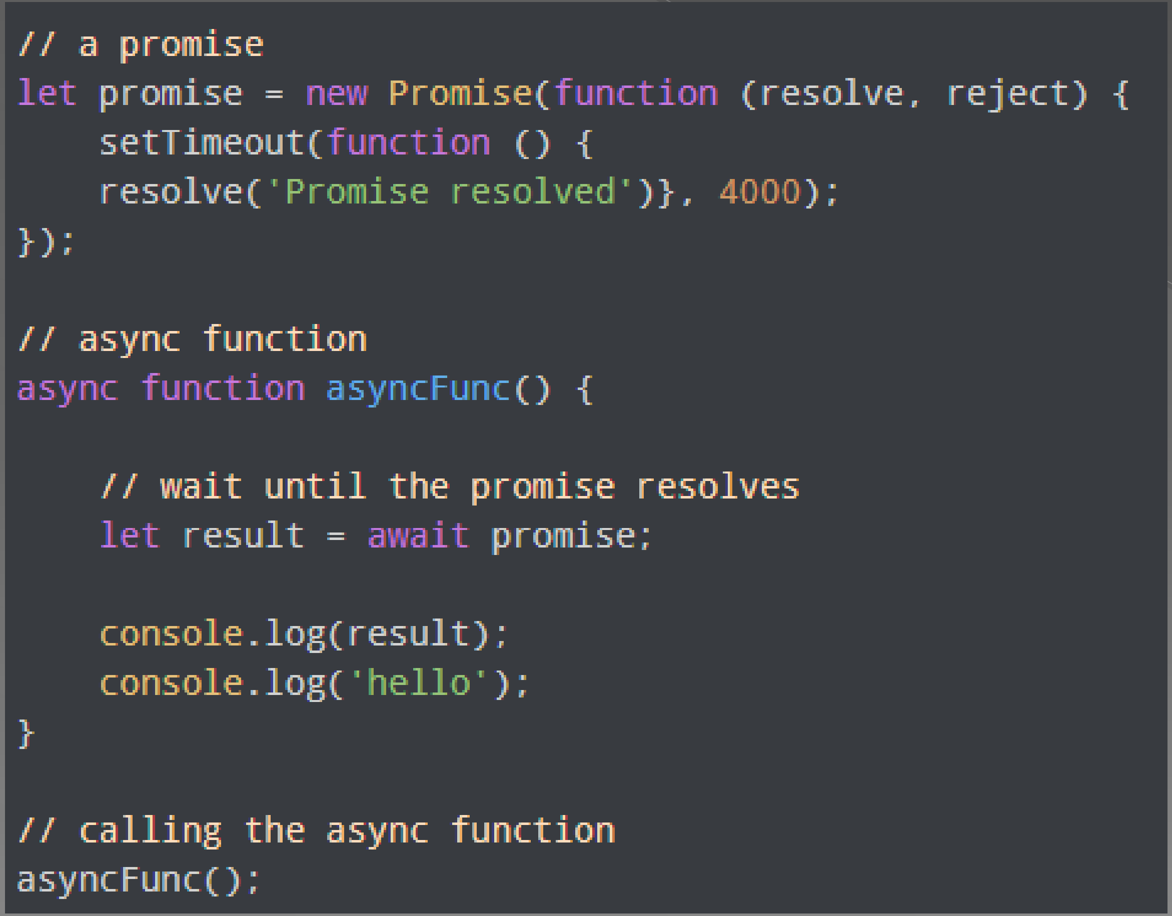
```
async function name([param[, param[, ...param]]]) {
   statements
}
```



```
async function f() {
     console.log('Async function.');
     return Promise.resolve(1);
}

f().then(function(result) {
     console.log(result)
});
```

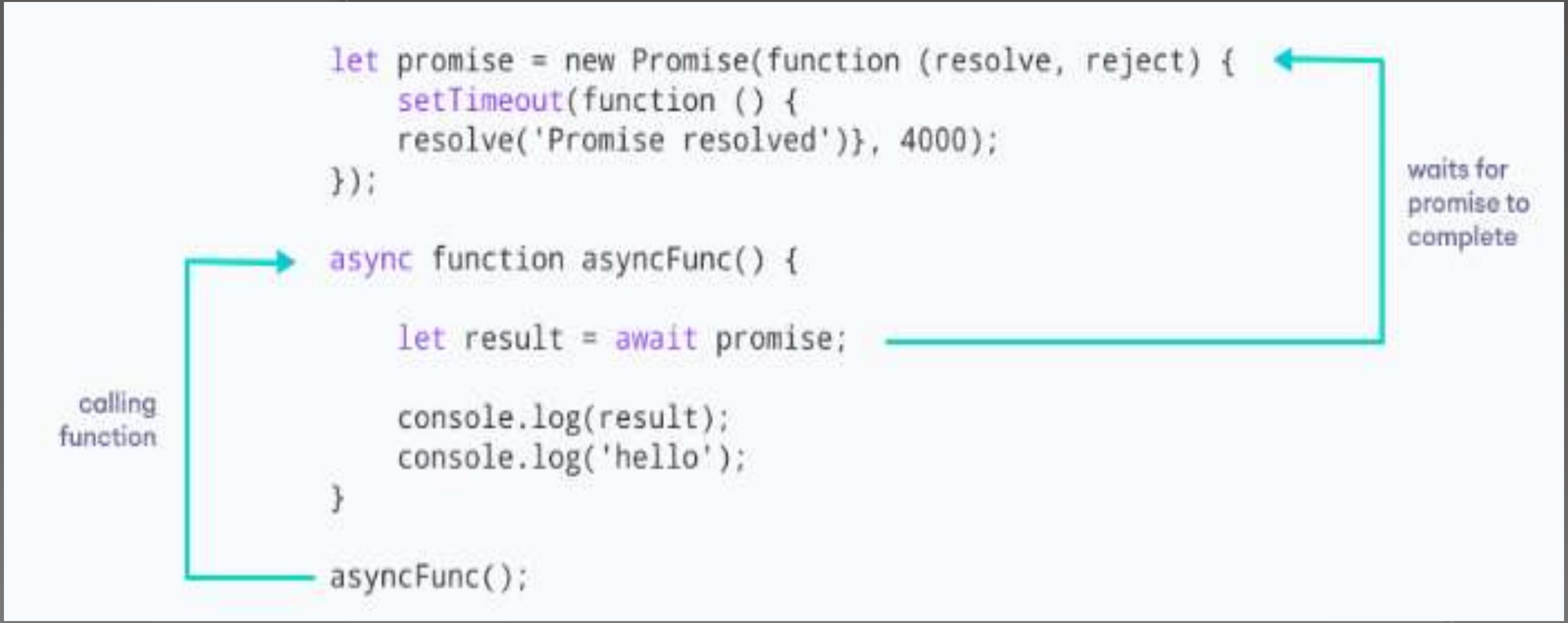
await keyword

The await keyword is used inside the async function to wait for the asynchronous operation.



```
// a promise
let promise = new Promise(function (resolve, reject) {
    setTimeout(function () {
    resolve('Promise resolved')}, 4000);
});

// async function
async function asyncFunc() {

    // wait until the promise resolves
    let result = await promise;

    console.log(result);
    console.log('hello');
}

// calling the async function
asyncFunc();
```

Working of async/await



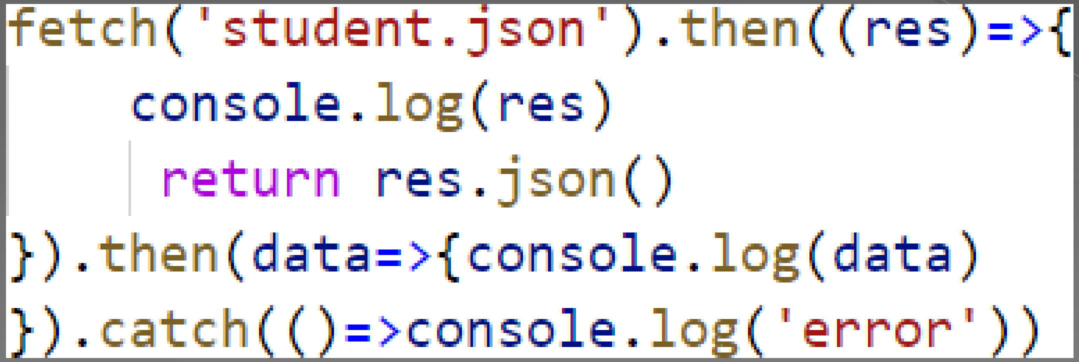
```
let promise = new Promise(function (resolve, reject) {
    setTimeout(function () {
    resolve('Promise resolved')}, 4000);
});

async function asyncFunc() {

    let result = await promise;

    console.log(result);
    console.log('hello');
}

asyncFunc();
```

Fetch API

The Fetch API interface allows web browser to make HTTP requests to web servers.

Syntax :

let promise = fetch(url, [options])



```
fetch('student.json').then((res)=>{
    console.log(res)
     return res.json()
}).then(data=>{console.log(data)
}).catch(()=>console.log('error'))
```

JSON

What is JSON

JavaScript Object Notation (JSON):

- Is an open standard light-weight format that is used to store and exchange data.
- Is an easier and faster alternative to XML.
- Is language independent format that uses human readable text to transmit data objects.
- Consists of objects of name/value pairs.
- Files have the extension .json.

Syntactically, JSON is similar to the code for creating JavaScript objects. Due to this similarity, standard JavaScript methods can be used to convert JSON data into JavaScript objects.

Why use JSON

- Straightforward syntax
- Easy to create and manipulate
- Supported by all major JavaScript frameworks
- Supported by most backend technologies

When use JSON

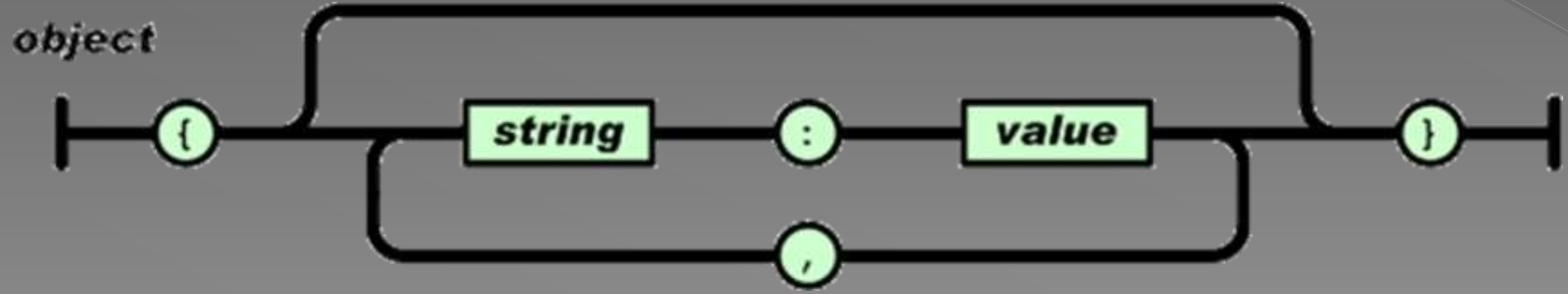
- Transfer data to and from a server
- Perform asynchronous data calls without requiring a page refresh
- Working with data stores
- Compile and save form or user data for local storage

JSON Syntax

- JSON uses name/value pairs to store data.
- Commas are used to separate multiple data values.
- Objects are enclosed within curly braces.
- Square brackets are used to store arrays.
- JSON keys must be enclosed within double quotes.

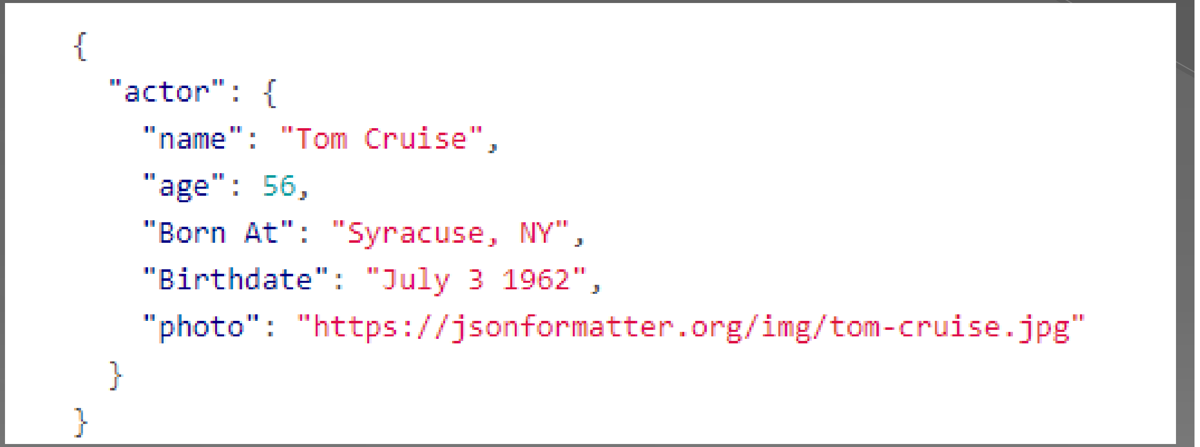
Syntax:

{"fName":"Ronald", "lName":"Smith", "Contact":12112345}

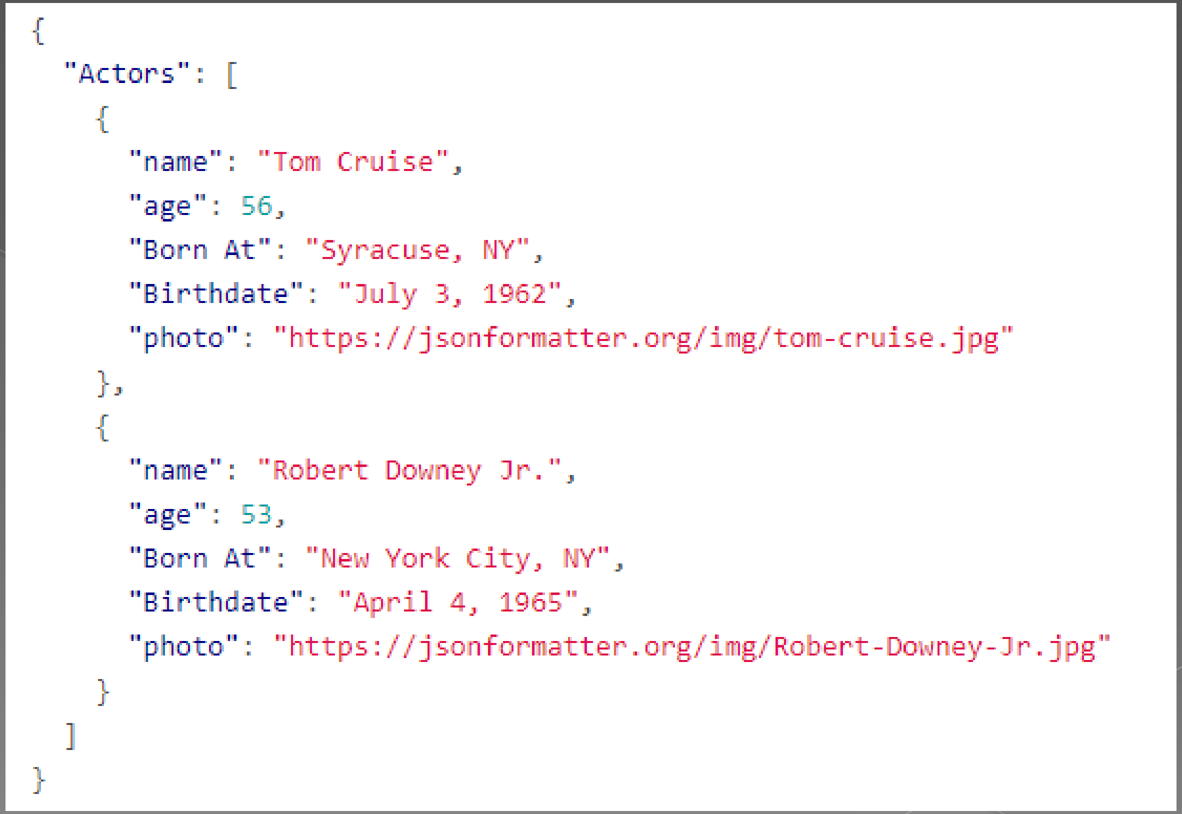


JSON Values

String
Number
Boolean
Null
Object
Array

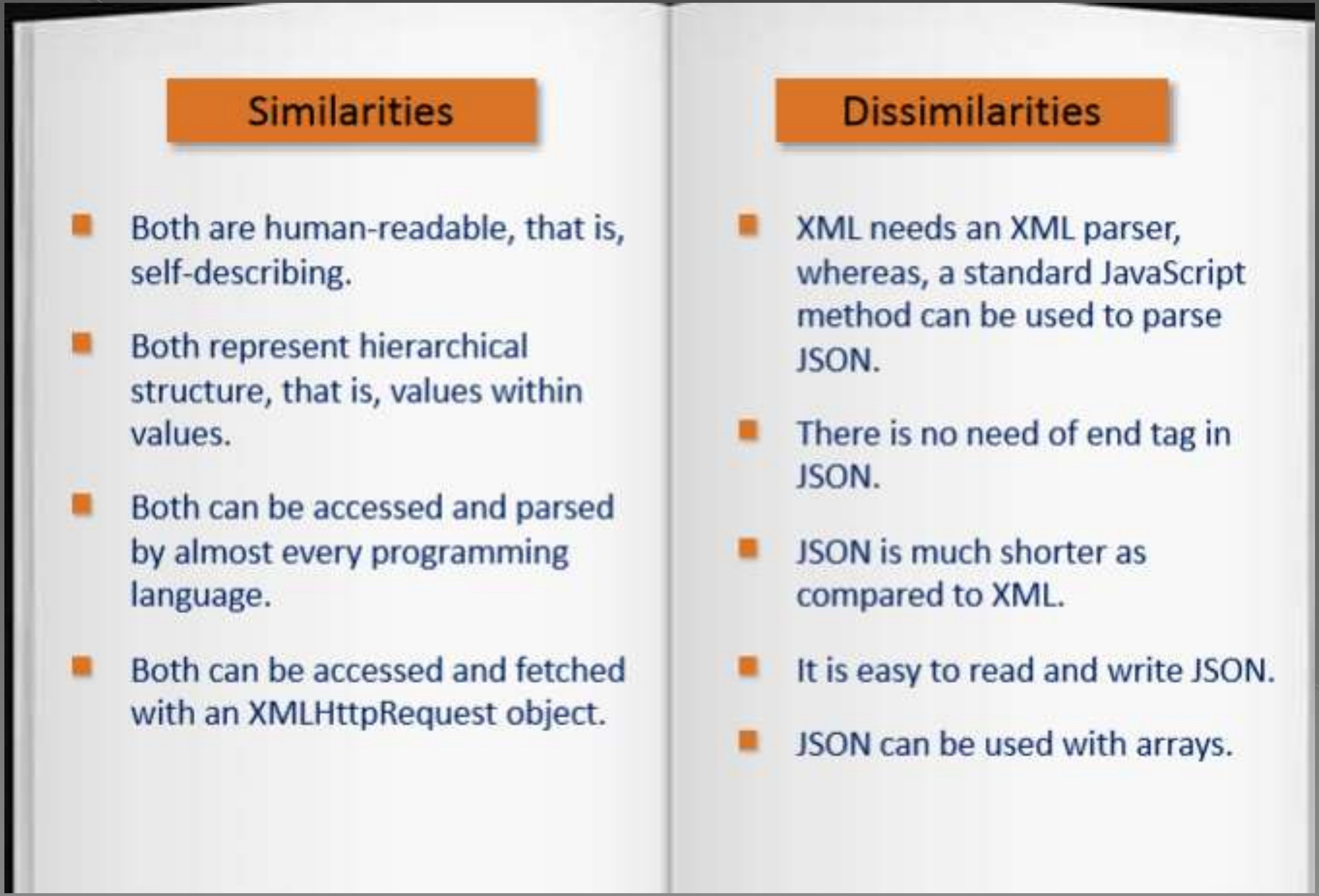


```
{
  "actor": {
    "name": "Tom Cruise",
    "age": 56,
    "Born At": "Syracuse, NY",
    "Birthdate": "July 3 1962",
    "photo": "https://jsonformatter.org/img/tom-cruise.jpg"
  }
}
```



```
{
  "Actors": [
    {
      "name": "Tom Cruise",
      "age": 56,
      "Born At": "Syracuse, NY",
      "Birthdate": "July 3, 1962",
      "photo": "https://jsonformatter.org/img/tom-cruise.jpg"
    },
    {
      "name": "Robert Downey Jr.",
      "age": 53,
      "Born At": "New York City, NY",
      "Birthdate": "April 4, 1965",
      "photo": "https://jsonformatter.org/img/Robert-Downey-Jr.jpg"
    }
  ]
}
```

JSON vs XML



Similarities

- Both are human-readable, that is, self-describing.
- Both represent hierarchical structure, that is, values within values.
- Both can be accessed and parsed by almost every programming language.
- Both can be accessed and fetched with an XMLHttpRequest object.

Dissimilarities

- XML needs an XML parser, whereas, a standard JavaScript method can be used to parse JSON.
- There is no need of end tag in JSON.
- JSON is much shorter as compared to XML.
- It is easy to read and write JSON.
- JSON can be used with arrays.

Infoway Technologies Pvt. Ltd., 3rd Floor Commerce Centre, Rambaug Colony, Paud Road Pune 411038

JSON vs XML

XML	JSON

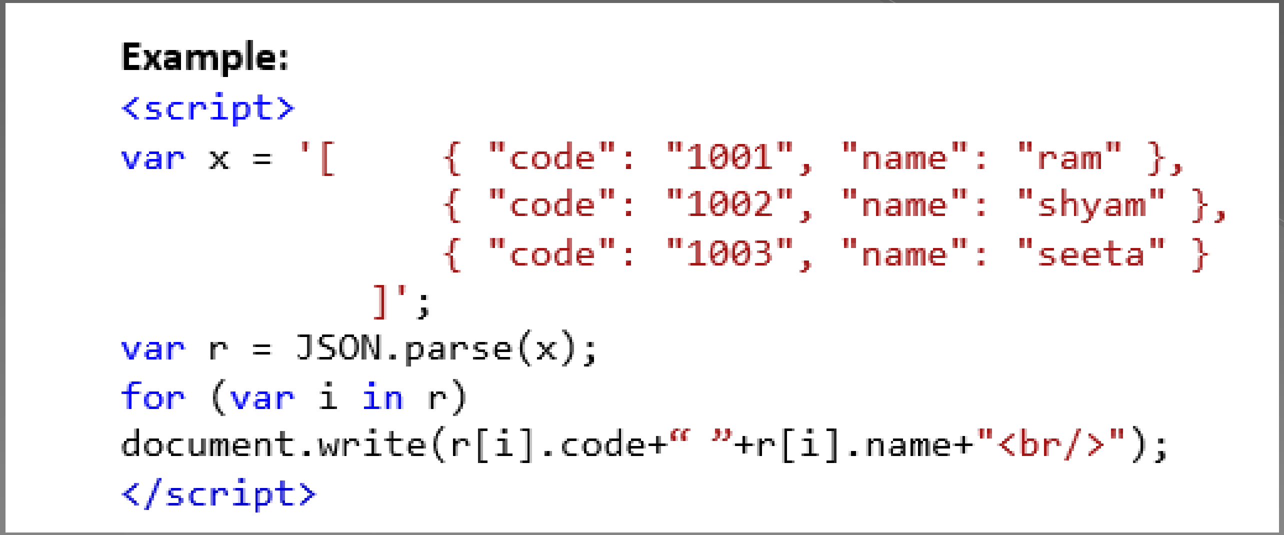
```
<students>
  <student>
    <fName>Jenny</fName>
    <lName>Watson</lName>
  </student>
  <student>
    <fName>Dean</fName>
    <lName>Smith</lName>
  </student>
</students>
```

```
{"students":[
    {"fName":"Jenny", "lName":"Watson"},
    {"fName":"Dean", "lName":"Smith"}
]}
```

Reading Data From JSON

To read data from a JSON object, you can use the JSON.parse()method
provided by JavaScript.

Syntax: var obj = JSON.parse(text);



Example:

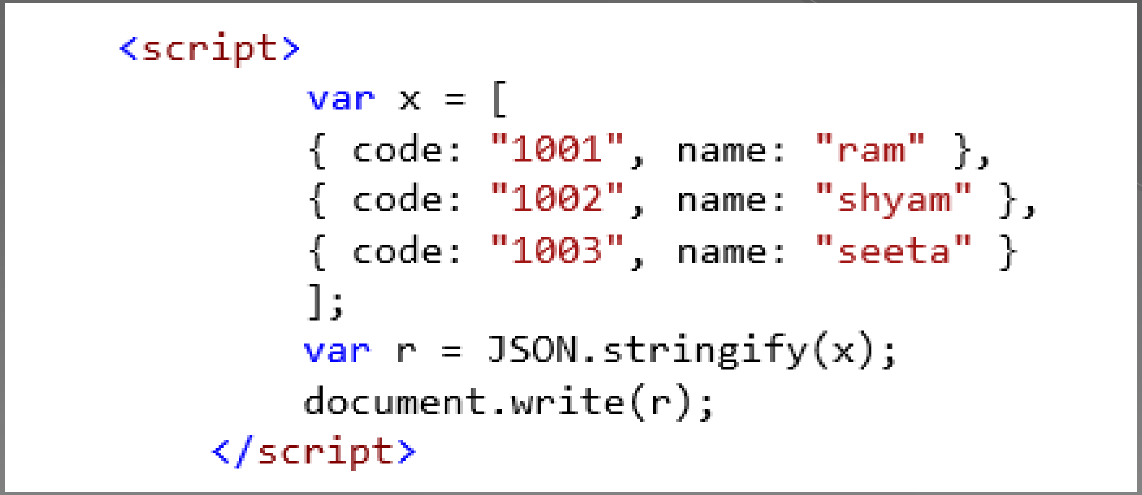
```
<script>
var x = '[     { "code": "1001", "name": "ram" },
               { "code": "1002", "name": "shyam" },
               { "code": "1003", "name": "seeta" }
          ]';
var r = JSON.parse(x);
for (var i in r)
document.write(r[i].code+" "+r[i].name+"<br/>");
</script>
```

Creating JSON Text From JavaScript

JavaScript provides you the JSON.stringify()method that allows you to convert

JavaScript value to a JSON string.

Syntax: var obj = JSON.stringify(value);



```
    <script>
            var x = [
            { code: "1001", name: "ram" },
            { code: "1002", name: "shyam" },
            { code: "1003", name: "seeta" }
            ];
            var r = JSON.stringify(x);
            document.write(r);
        </script>
```

Thank you !!