**Mostly About Concepts…**

**Creator’s Note**

* These aren’t notes. These are just concepts which I had problems to grasp.
* Just grab a snack and go through them once as revision.
* These are out of order as they were added when I encountered such doubts.
* This file is under construction and will update whenever I get another doubt.
* **Giriraj**

**Tighten your seatbelts, here we go…**

1. Every browser has it’s **JS Engine** which runs JS code, JS File/Code -> Parser [Throws Error] -> Interpreted Language Which is Interpreted in Parser -> Converts it into AST -> Machine Code and All this done by JS Engine.
2. A JS Engine just deals with JS code and JS is a **single threaded language** then **How it performs task asynchronously?**

**JavaScript Runtime**

JavaScript runtime consist of JS Engine + Browser Tools.

Browsers Tool contain various Web APIs [Window Object], Call Stack, Callback Queues & Event Loops which take care of all asynchronous part of code JS Engine transfer that async part to Web API and other Browser tools which take care of that and all this combine constitutes a JavaScript Runtime.

In Node.JS (Node.JS is also JS Runtime) there is V8 Engine for JS code and Libuv Library which handles the other functionality. Thus, we can say is Node.JS establish connection between V8 Engine & Libuv Library.

1. **Is JS is actually interpreted Language?**

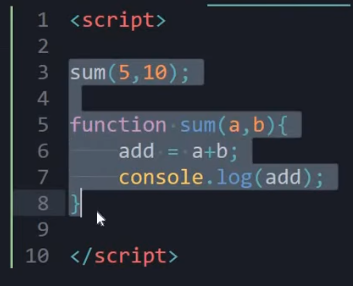
Initially, it was but not today actually what happens is when AST passed to **Interpreter** then It generates byte code and start executing but also it passed whole code to **Profilers** which passes whole file to **compilers** which scans whole file and do the necessary optimization and replaces unoptimized code from byte code to this optimized code and hence, its performance increases gradually. **V8 Engine based on above working. Thus, we can say It is JIT Compiler Language [Just in time Compiler].**

1. Every function creates its **Execution Context** and with a separate memory and space, every function will be stacked according to their calls/invocations, this stack is known as **Call Stack**.

Only functions scope is created in JS, only functions that’s it.

1. Execution Context has 2 parts: -
   1. **Creation Phase** – Hoisting is done here.
   2. **Execution** **Phase-** It is simple execution of code via JS Engine.
2. In creation phase: -
   1. **Hoisting** is done
      1. For each function declaration a property is created in variable object pointing to that function.
      2. For each variable a property is created and value set undefined.
      3. Argument objects are created that were passed into functions.

This is Function Hoisting as its definition will go on top automatically, It’s a feature of JS

****

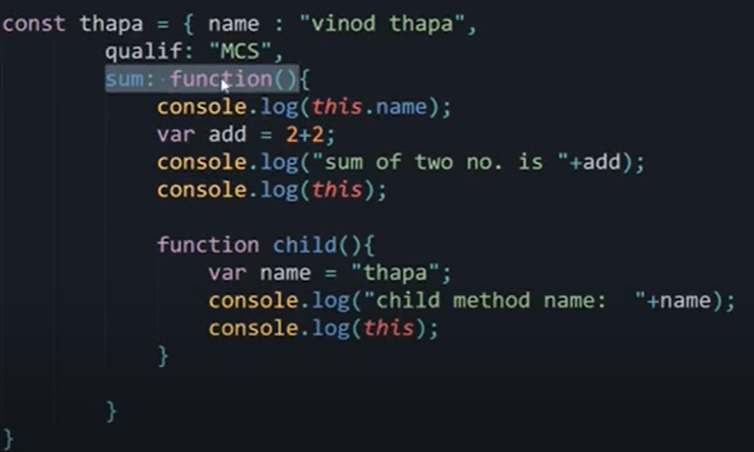
This is variable Hoisting as declaration will move up and set to undefined see only declaration will move up not initialization.

The Output will be undefined



* 1. **Scope Chain** – Parent functions will not be able to access children’s values, but child function can access parent’s function variables, also known as **Lexical scoping**. Thus, scope move upwards from itself to parent to grandparent and so on till global scope is reached.
  2. **This** Keyword – refers to the **object** it belongs to. Some points to remember: -
     1. Refers to object it belongs.
     2. If used in regular function refers to the global object.
     3. In a method, this refers to the owner object, but if in object regular function is created then it will again refer to the global object.

As in below example, child function is defined inside sum method so this keyword used in method function will refer to the object Thapa but the child function isn’t a method, it is simple regular function that’s why it will refer to the global object.



1. **Everything in JS is Object. – Important.**
2. For Primitive variable types like number are pass by value, means if we say var a = 10;

var b = a; then both a & b are independent will have different memory space.

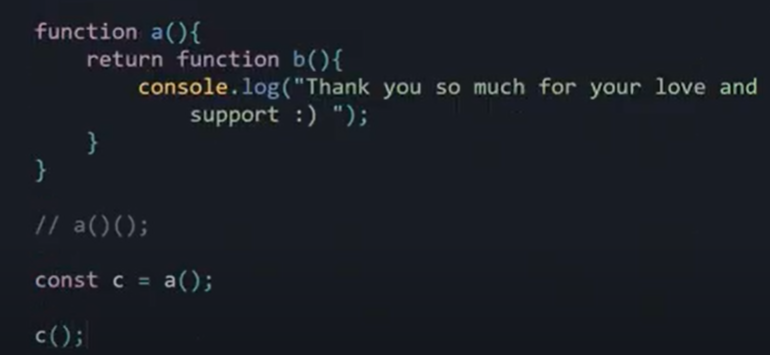
But in case of objects memory reference is created & both variables will refer to the that same memory location and change in one variable automatically changes the other variable as they are pointing to same reference, a big problem in objects & solved by **deep copying**.

Here likes value will remain 100 but subscribers value will change.



1. On adding “**use strict**”; at top JS enters in strict mode and behave normally like other programming languages like hoisting disabled you must declare variable before use, you must use **var**, **let** or **const** keyword to define variable etc.
2. Functions are **First Class Citizens**
   1. We can assign functions to variable & then accessed as normal functions.
   2. We can pass functions as arguments – use case: **Callbacks ()**
   3. A function can return a function– also called **Higher Order Functions.**

Two ways of accessing them are:

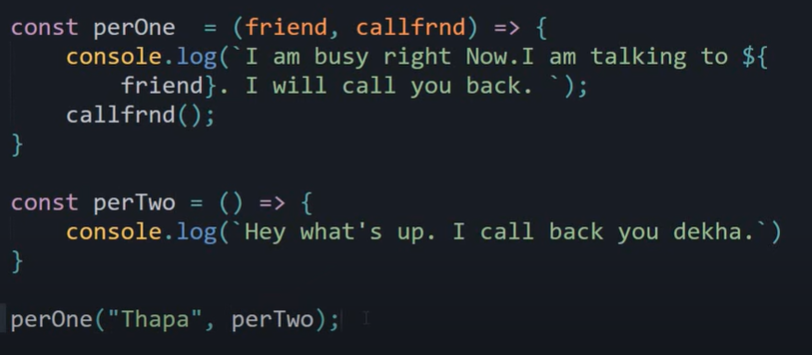


1. A note on **Callback** **()**

JS is event driven language i.e., instead of waiting for response before moving on it will keep executing other code.

Callbacks solve this problem as they make ensure that particular code wait for other code to run first and this is achieved by passing functions in arguments.

While passing function as argument don’t use () just pass the name of function.

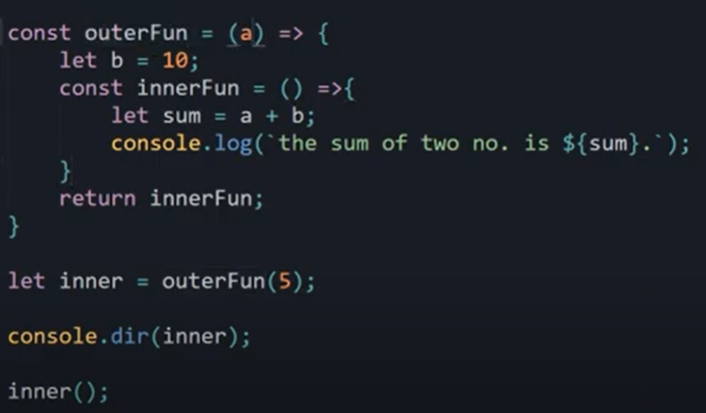


1. **IIFE (Immediately Invoked Function Expression)** as soon as function defined it is called.
   1. Immediately invoked after definition.
   2. Grouping operator (): Particular Lexical scope and it has private scope means the variables defined here will not be available outside

An IIFE Example – as aName variable not available outside & it is invoked as soon as defined and defined inside grouping operator thus it is binded to this particular scope.  


1. What and Why is **Closure**?

Suppose an example



In this example, when outerFun is invoked then it will be removed from callstack and loses its execution context hence, all its memory and variable. Then How innerFun will be able to use its parent variables after it’s invoked. This problem solved by **closure.**

As a memory space created where all parent’s variable their value and other important information is saved.

1. **Call ()** Method in JS

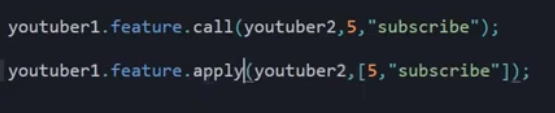
Using call () method an object is able to call function belongs to other object and can evaluate its value into that object’s function.



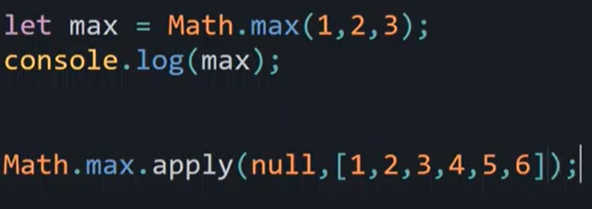
Calling youtuber1’s function with values of youtuber2.

1. **Apply ()** Method in JS

Only difference between call () and apply () is that it apply () take argument of functions in form of array rather than series of argument passed in order.

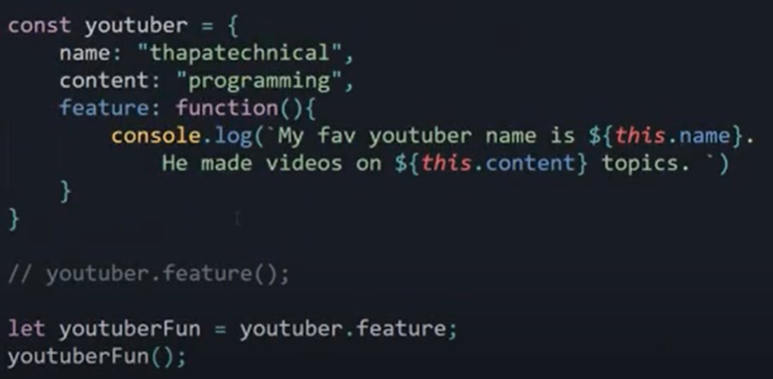


Use case of apply ():



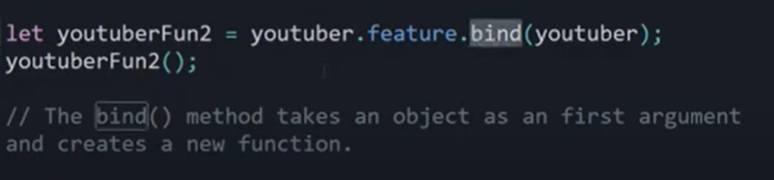
1. **Bind ()** Method in JS

Consider this example



In this youtuberFun will not be able to use name and content property

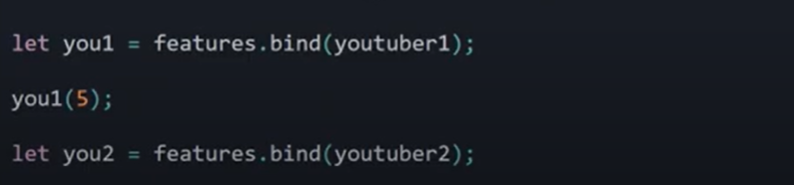
Solution for this is to bind the object to a particular function using bind method

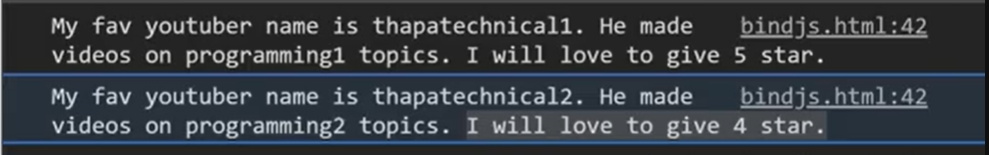


We initialize the feature method to youtuberFun2 but we also binded youtuber object to this function too.

For better use case follow below example.



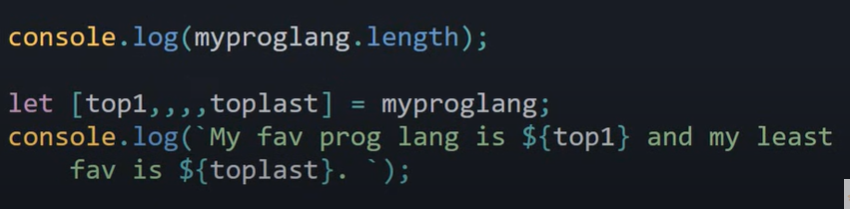




1. What is **Array Destructuring**?

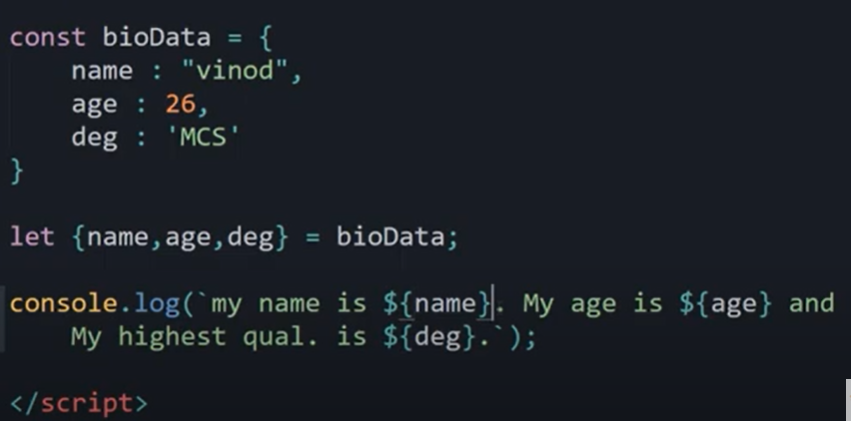
Follow the below example, it is way to initialize the variable to array values



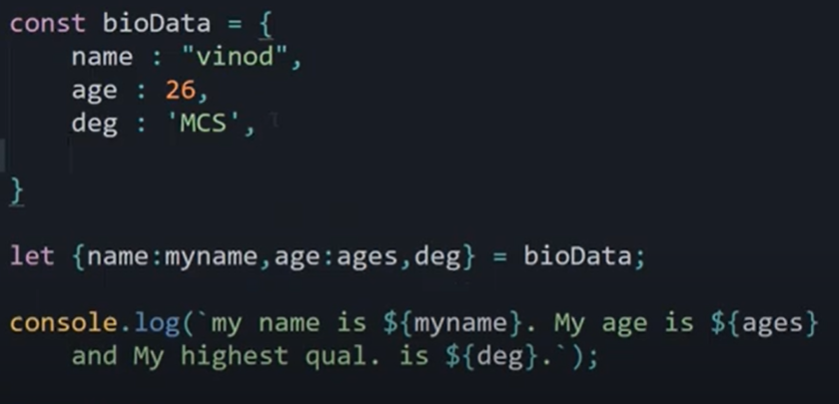


1. Similar to Array Destructuring there is **Object Destructuring**

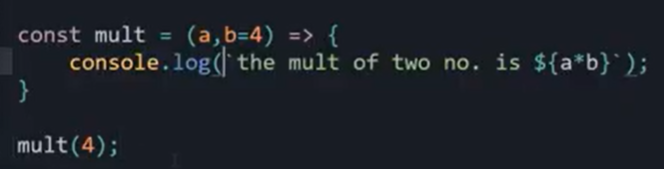
It is similar to array destructuring as it also unpacks the values of object but here the please note that **name of variable must be same as the property names** otherwise it will not work.



**Although we can rename the variables, follow the below example**



1. Default Parameters Example

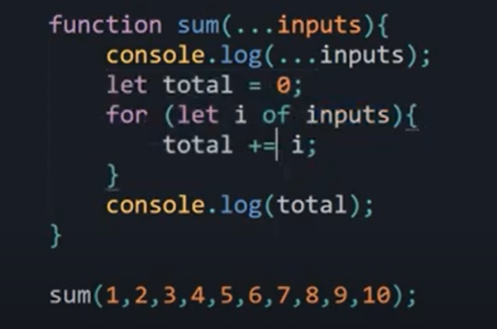


1. REST and SPREAD Operators in JS

Both have same operator i.e., (…)

**REST Operator:** …variable\_name

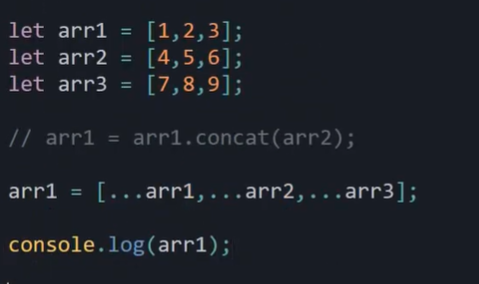
It is used to convert indefinite no. of passed parameters/arguments into array.



Here, …inputs a rest operator.

**SPREAD Operator:** …variable\_name

It allows iterables to expand at place where 0+ values are needed.

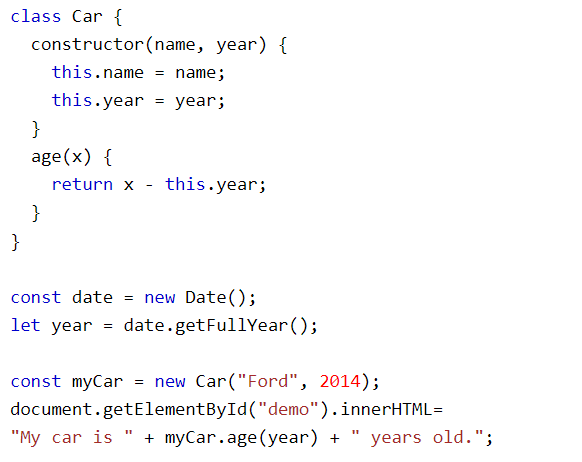


Thus, though having same operator both REST and SPREAD works differently.

REST take indefinite number of parameters passed used when we don’t know exact amount of varialbes and SPREAD just spreads the values of iterables where it is used mostly used in deep copying.

1. Some Information about **Classes** in JS

Classes are somewhat like functions but a more a generalized representation.



Some important points to remember

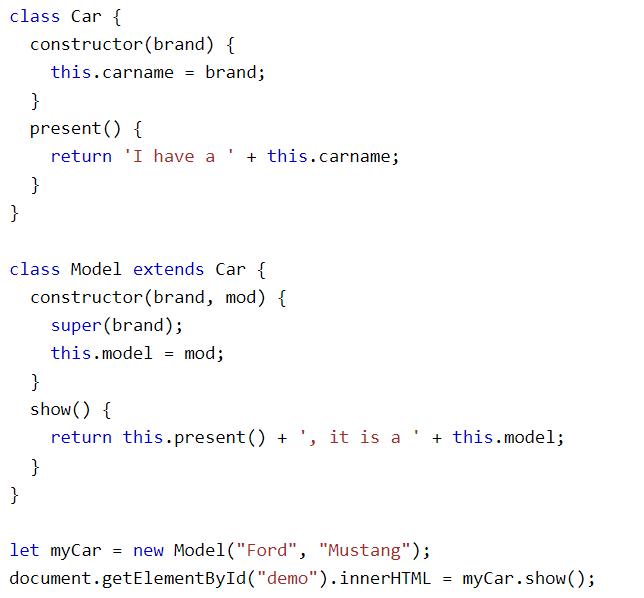
1. No need to save methods in some variables.
2. Constructors are automatically called and used to initialize the properties.
3. Properties can only be initialized inside functions only.
4. **3 types of functions**.

**Constructor** – called automatically and used to initialize properties of classes.

**Prototype** – these are normal functions/methods we create for class.

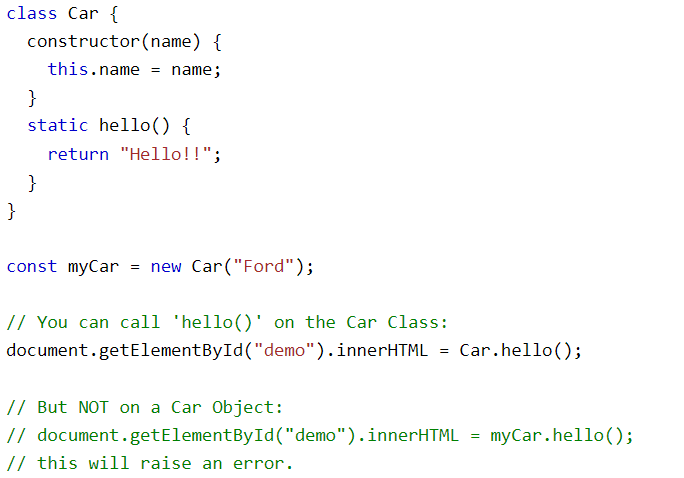
**Static** – not available to any object just belong to class only and only class can call it also it can’t be inherited to other classes.

1. Classes can show **Inheritance**



Here in this example, Model inherit Car class using **extends** keyword. Here are also some points to remember. To initialize properties of parent class we use **super ()** method. Also same for same name method in parent class otherwise it will call child’s method.

1. Also, to initialize the properties of parent class you have to pass in **same way** as mentioned. If the name of properties and methods are different child object can access parent’s methods & properties.
2. **Static Methods**

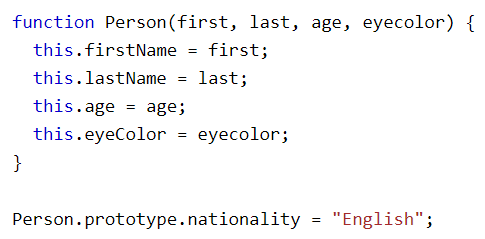


* **Prototype and \_\_proto\_\_**

This will allow you to add new property to object constructor and thus will get you extra functionality. They can be chained means A prototype can access B prototype property can access C prototype property. Although it is deprecated after release of class concept but it still exists and can be useful.

If you ask for property then JS first will check current Object’s property if it isn’t there then it will check proto-object.

You can see many objects like string and arrays have some properties defined in proto-object that we use a lot and also, we can add our custom properties in objects also.



1. **Synchronous and Asynchronous**

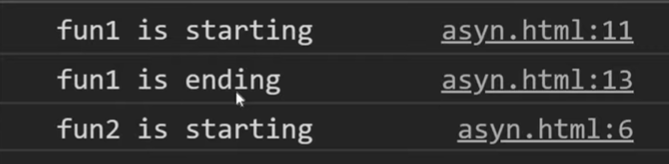
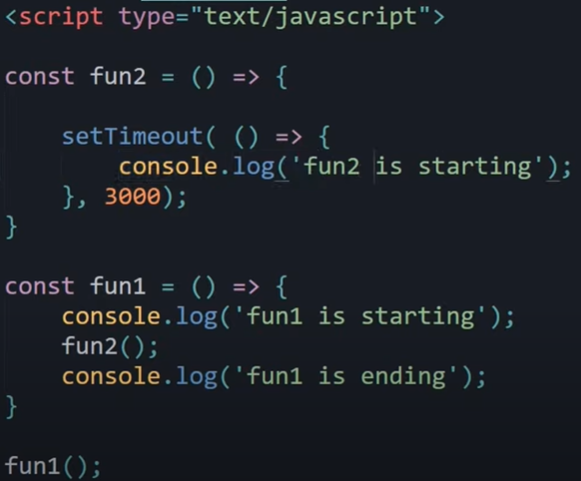
In Synchronous JS, request blocks client until operation completes. Means it waits for the particular function to complete execution before moving to next operations.

Page become unresponsive and browser become unactive.

In Asynchronous JS, request doesn’t block the client operation and doesn’t wait for the task to complete its execution instead it moves to the next operations.

Page remains responsive and browser remains active.

JS follows **Asynchronous approach**.

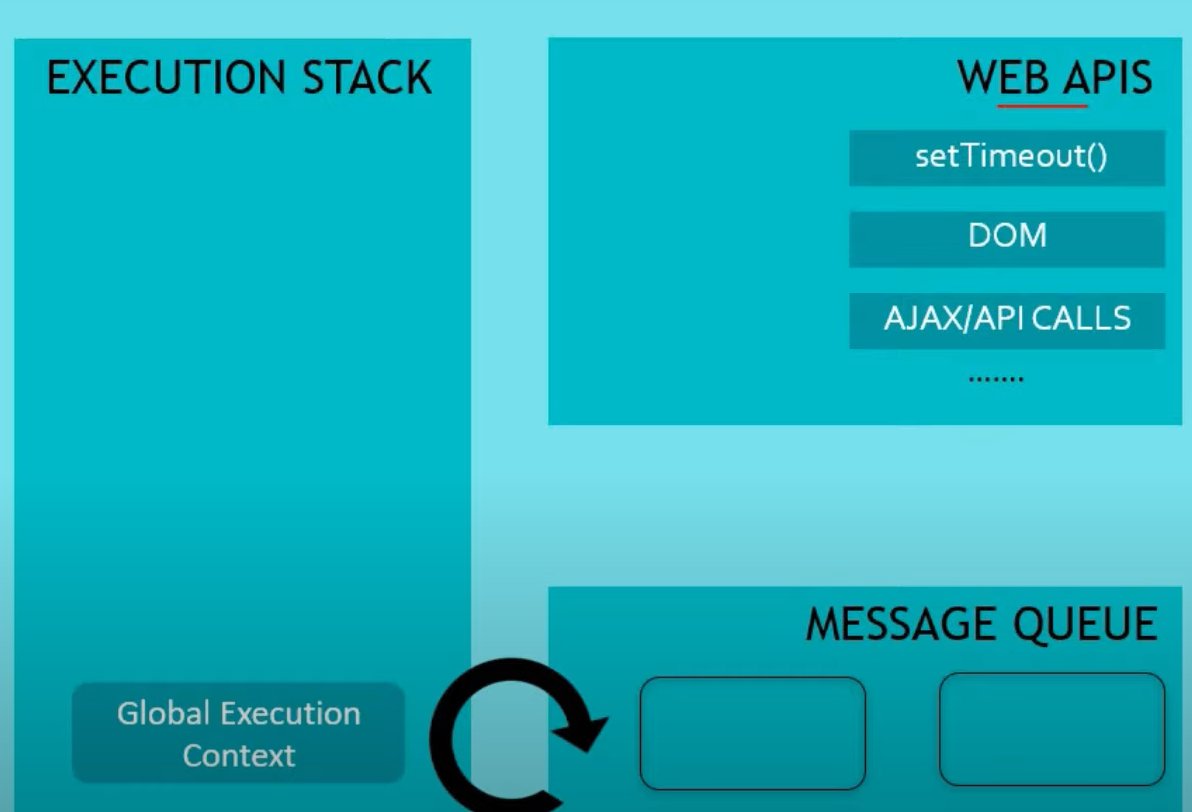
4

Here you can see that it didn’t wait for setTimeout () to complete 3 seconds and continue executing next operations. This is Asynchronous approach.

Hence, it is used to make responsive websites which will not unactive the website for some functions to complete its execution.

1. What is **Event Loop**?

It’s the way through which JS becomes an Asynchronous Language, as when setTimeout () or any other function which is waiting/running some other work in background related to web API or any other tasks completes its time/work, will pass to messages queue from where a continuous event loop is running which check the message queues and transfer ready to execute functions back to Execution Stack and then they work normally.



Here is an example,

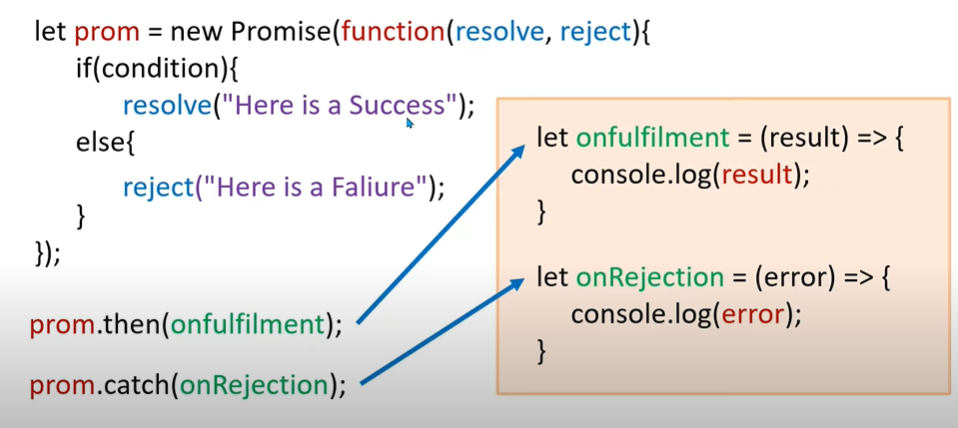
1. You called a function F1 and in F1 there is setTimeout () of 10 seconds and inside setTimeout () callback function there is console log statement.
2. So, F1 is pushed into Execution Stack then setTimeout is transferred to Web API Environment then from there It will wait for 10 seconds.
3. F1 is poped out from execution stack as it’s work is complete although setTimeout is running in background.
4. After 10 Seconds setTimeout is transferred to message queue where event loop checks it and then on seeing it ready it transfers it back into execution stack where it runs its console log statement and then finally it poped out.
5. **Promises – Upgrade to callbacks**

To maintain asynchronous power of JS and to avoid callback hells. We need promises.

Basically, promises have 3 stages.

* Pending: Till the ongoing work doesn’t finish
* Fulfilled: When there is a success – here resolve function is called.
* Rejected: When there is a failure – here reject function is called.

Below is basic idea of promises.



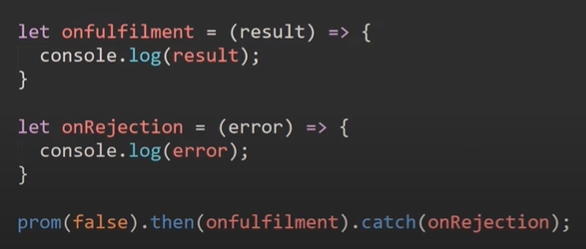
We can either create promise and assign it to a variable or we can return the promise via function.

|  |  |
| --- | --- |
| Method 1: Creating Promise and assign it to a variable. | Method 2: Returning Promise via function. |
|  |  |

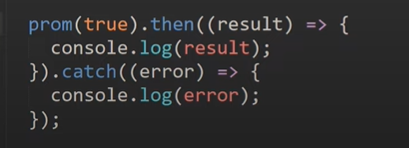
Now, suppose if promise is fulfilled then resolve/reject function is called depending upon the condition.

Which we call in below mentioned way. Also, the parameters passed into resolve function is accessible to then () and same for reject function. We can pass a function into then () which will perform the certain task.

Follow the example based on above code.



It’s not necessary to pass the function we can use anonymous function too.

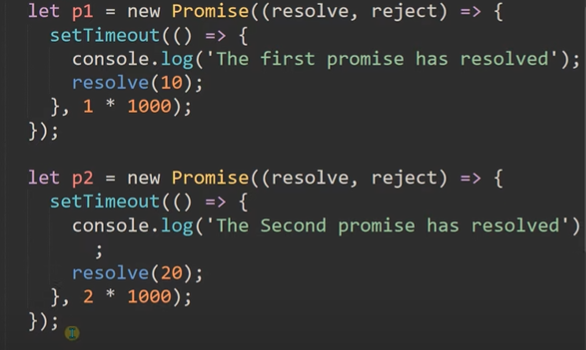
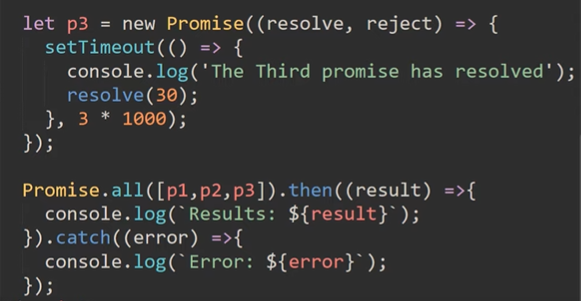


What is **Promise.all ()?**

Well, if we call each and every function then It will increase the code therefore to call all or certain promise at once we use Promise.all () function.



Follow the below example

1. **Async/Await – Upgradation to Promises**

Async function returns a Promise but it makes the task much easier. To use async/await we first add keyword async to a function definition. Await can only be used inside async function only.

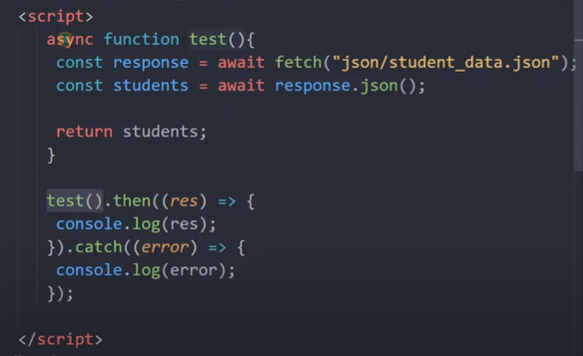
Just like in promises we wait for the result and by checking result we decide what to do next but in case of async/await here we don’t need to write messy if – else conditions we just use await keyword.

Follow the example.



Here you can see await is waiting for the code/task to complete and execution didn’t stop for it, Execution goes on and type message 5 then after completes the remaining task. By default, it assumes that task resolves.

Use case example of async/await.



Here we didn’t use the if-else conditions just write await and store it in result if success it will return result if error it will return error.

Thus, in case of success it then () is called and in failure catch () is called but here we don’t need to write resolve and reject they will be called internally and handled by await.

Thus, always use **Async/Await**, they will make code easier and understandable.

1. **XMLHttpRequest () vs fetch ()**

* Basically, 3 types of **data format** used: text, XML, JSON. JSON most favourable.
* 5 **readyStates**: -

0 – request not initialized

1 – server connection established

2 – request received

3 – processing request

4 – request finished & response is ready

* **Status codes** – like 404, 200, 403 etc.
* **Response text** in XML or JSON.

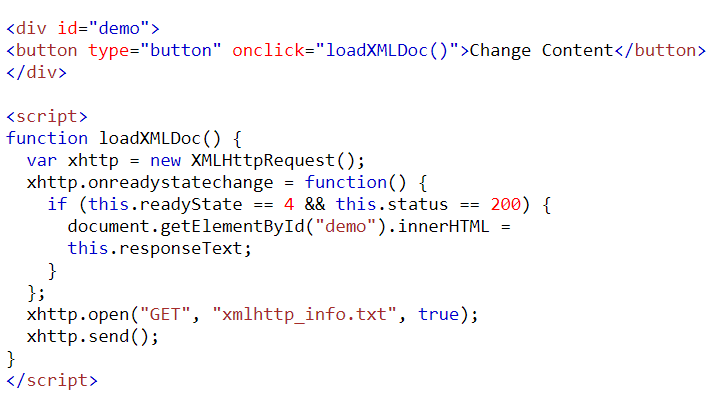
Now, how to do it using **XMLHttpRequest ()**?

First, we made a XMLHttpRequest object and save it in a variable.

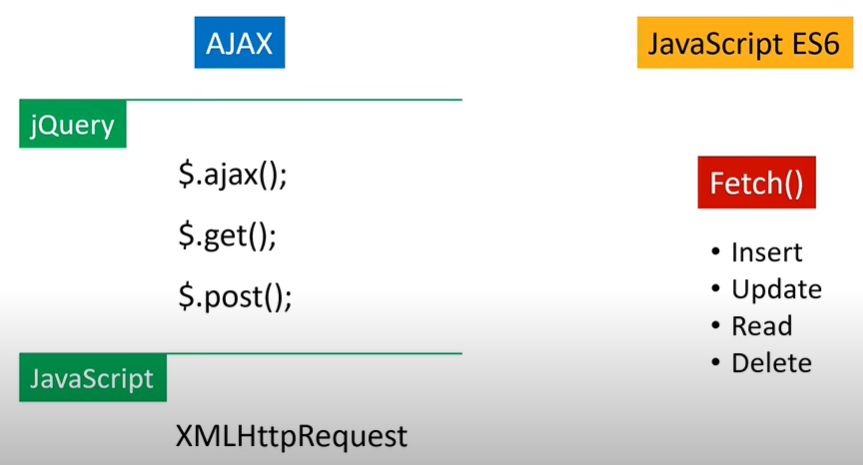
Then, we set onreadystatechange property of object which constantly monitors the change in state and if the given condition met the requirements, it do as coded.

Obj.open() and Obj.send() used with each other.

Follow the Example



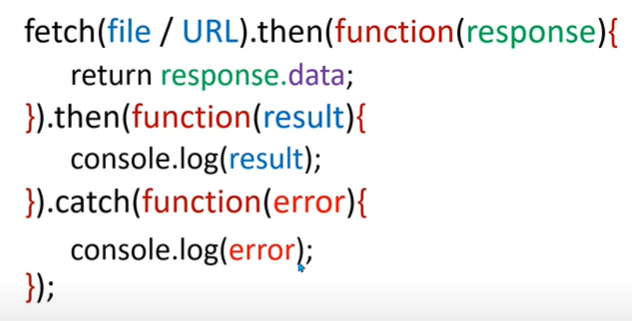
Overview of different techniques used for background data collection from server



How to do it using **fetch ()?**

Fetch () is introduced to tackle complex task easily & it will return **Promise.**

**Basic syntax of fetch ()**



Example of using **fetch ()**



Fetch () function has other functionality also like: **Update, Delete, Insert**

See on Internet how they work as it is easy & step-wise procedure.

1. **Iterators & Generators in JS**

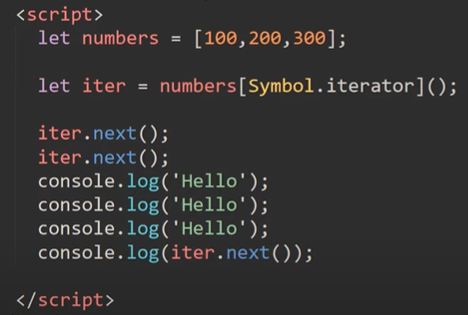
**Iterators** give full control on the looping and we can decide when, what and how will the next value behave and produced.

There is a special way to define iterators and we use next () function to iterate to next value which have 2 values: **value** and **done**

**Value –** what is the value given/ if ended then value becomes undefined.

**Done –** whether the iterable entitiy ended or not / true means ended false means not.

Follow the example



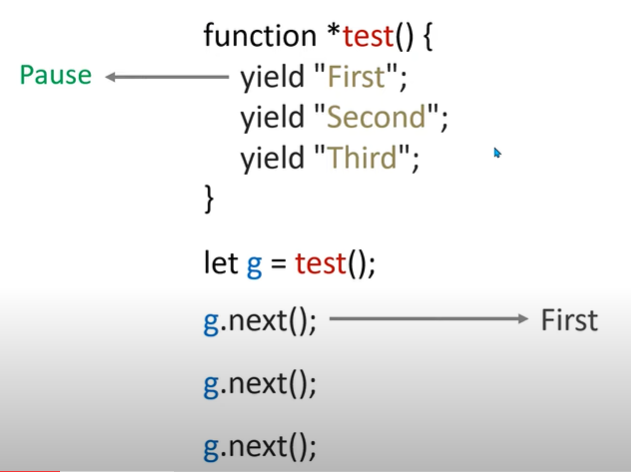
Traverse to 100, 100 and then we console values then again Traverse 300 console this value.

If we used loops then we have to use if else conditions. Hence, Iterators will give more power in our hands.

**Generators** give full control over continuous code and functions and give breakpoint power in functions.

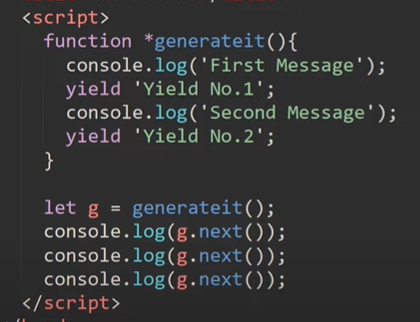
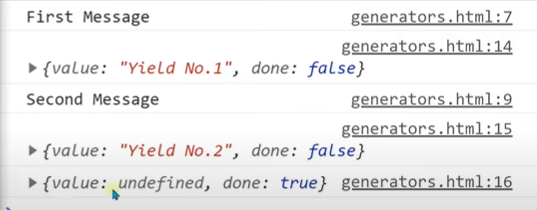
It have different functions like next (), return () etc.

**Syntax**

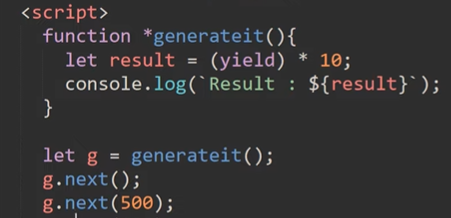


It will not yield keyword until next () function is used. Hence, can be used as breakpoints.

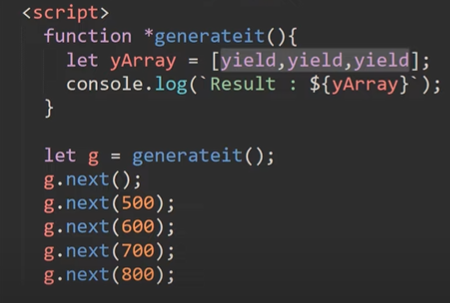
Here some examples demonstrating generators and other use cases.

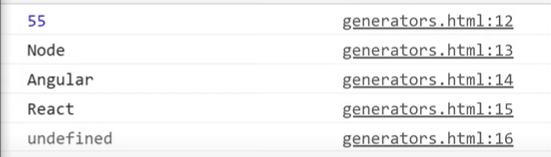
* We can pass values into yields using next () function.



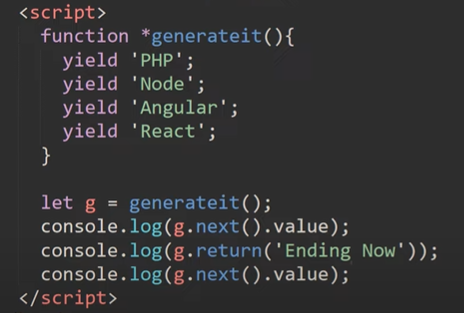
* We can pass multiple values into generators at same point.



* How to use yield array values at same point.

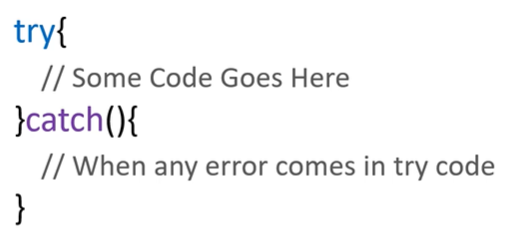
* We can also return, it used as normal return in functions code below return doesn’t run, it will return from the yield it has returned.

1. **Error Handling in JS**

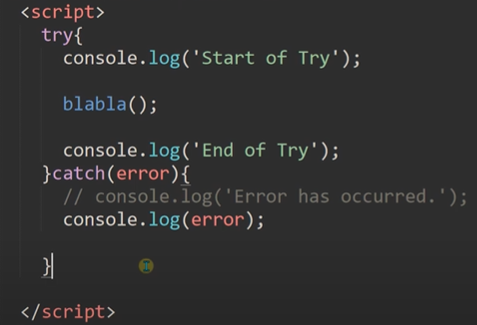
We use try () and catch () for error handing in JS.

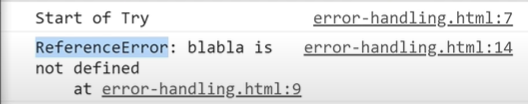
Basic Syntax



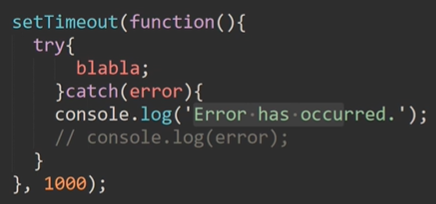
We write all the code in try section and if error occurs then it will go into catch function and there, we will display message or handle the error.

Example

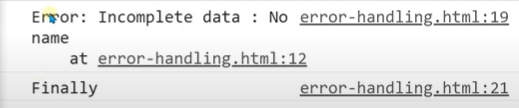
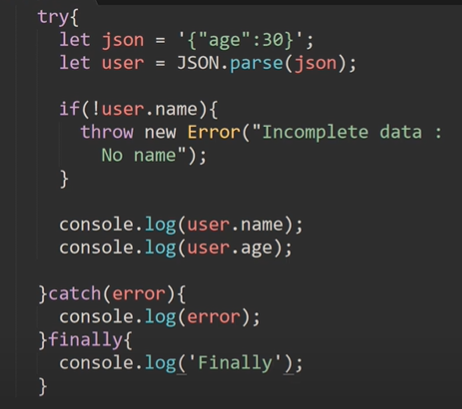




**Note:** try and catch work synchronously and thus function like setTimeout () behave abnormally thus for error handling of setTimeout type functions use try-catch inside setTimeout code block.



**For validation and testing we can throw our custom errors in try code using throw () function.**



As shown in above example, that there is also other method known as **finally ()** which will run whether the try block runs or catch block it will run whatever happens.

1. **JOB Queue**

With new versions of ECMAScript and addition of concept of **Promises** and a new queue other than callback queues are added known as JOB Queue for the High Priority Tasks like Promises and Async and Await Functionality.

So, JS code will run on main Thread and all other Tasks will be handled by other Library/Browser. For JS It will be Browser and For Node.JS it will be LibUV library.

Thus, Processor First will check the Job Queue than will go to Call back queue.

That’s why Promises will run earlier than other functions like setTimeout etc.