JavaScript Tutorials

Javascript 🡪 It is a lightweight, interpreted (just-in-time), object-oriented programming language best known for developing dynamic web pages.

It is multi-paradigm scripting language that supports oops, functional programming, imperative styles.

Whenever we run JS code an execution context is created.

Consider the execution context as a container. It has two parts memory component( variable environment) which stores variables and functions as a key:value pairs. The other part is Code Execution in which code is executed sequentially.

\*Note: Along with the variables and functions there is a reference to the lexical environment of the parent.

**Javascript is single threaded synchronous programming language.**

Code Execution (Thread Of Execution)

Memory (variable environment)

Sequential Execution

Key:value

a:10

fn: {}

Execution context is created in two phases first is Creation Phase/ Memory creation phase and second is code execution phase.

**In the first phase** JS will goes through whole program and allocates memory to all the variables (name of the variable will be the key) and stores special value “undefined” and for the case of function it stores the function name as the key and value will be whole code of the function.

Lets understand by an example.

var n=2;

function square(num)

{

    var ans=num\*num;

    return ans;

}

var square2 = square(2);

var square4 = square(4);

Code Execution

Memory

n: undefined

square: {….code of square function….}

square2: undefined

square4: undefined

**In the second phase (Code Execution)**, JS starts executing the whole code, it scans through the code sequentially and starts assigning the value for the variables.

Like in this example when JS start executing it changes value of variable n to 2 from “undefined”.

Code Execution

Memory

var n=2;

function square(num)

{

    var ans=num\*num;

    return ans;

}

var square2 = square(2);

var square4 = square(4);

n: ~~undefined~~ 2

square: {….code of square function….}

square2: undefined

square4: undefined

**Functions** are like mini program in JS.

Var, let and const are function scoped means that if any var, let and const variable is declared in the function then it cannot be access outside that function. So those variables have function scope. (OR) The variables which can be accessed only within the function and not outside it are said to have function scope.

Whenever a new function is invoked a new Execution context is created inside the parent execution context and this execution context also has two parts variable environment and code execution

So in **phase 1 (memory)** memory allocation will take place and memory will be allocated for function parameters and variables in function with the special value “undefined”.

Code Execution

Memory

code

memory

n: ~~undefined~~ 2

square: {….code of square function….}

square2: undefined

square4: undefined

num: undefined

ans: undefined

**Global Context in JavaScript**

**Phase 2 (Code Execution)** of the function will start where argument’s value will be assigned to the function’s parameter value and other execution of function will take place.

When square(2) function is invoked.

Here in example, (1) variable num value will be changed from undefined to 2 and (2) value of the variable ans will change from “undefined” to 4.

(3) In return ans statement JS will then find the value of that ans variable in local variable environment and (4) finally value of variable ans will return to the variable square2 of parent execution context.

Code Execution

Memory

Find value of variable ans in local memory

code

memory

n: ~~undefined~~ 2

square: {….code of square function….}

square2: undefined

square4: undefined

function square(num)

{

    var ans=num\*num;

    return ans;

}

num: ~~undefined~~ 2

ans: ~~undefined~~ ~~4~~

**Global Context in JavaScript**

**Phase 2 (code execution) when square(2) is invoked**

After the execution of the function the execution context of the function is deleted. Again the same procedure will be followed for the square(4) function.

**Call Stack/ Runtime Stack/ Program Stack/ Execution Context Stack**

JS has a call stack(FILO) to manage the execution context.

GEC 🡪 global execution context.

EC 🡪 execution context

Square2 func invoked so new EC is pushed on the stack.

Square2 func finished so remove

EC of the square2 funciton

GEC

GEC

EC1(square2)

GEC

**Hoisting in JavaScript**

var function1 = function(){}

var function2 = () => {}

Both the variables function1 and function2 will behave like “variables” and not function and that’s why in execution context in Phase 1 (memory Allocation) function1 and function2 will have value **“undefined”**.

function function3(){}

Here function3 is not treated as a “variable” but as a function.

**LET and CONST**

Let and const are hoisted but the memory is assigned to them in a “temporal dead zone”.

\*You cannot access the let and const until and unless a value is assigned to them.

If we try to access the let and const variable before initialization then it will throw Reference Error.

\*You cannot redefine the let variable again or else it will throw Syntax Error.

**const** variable must be initialized at the time of declaration and the value cannot be changed after that.

**TypeError:** assignment to a const variable.

Const a = 1000;

a=222;

JS expects that variable won’t be assigned to a const variable again after initialization.

**Syntax Error: Identifier ‘a’ has already been declared.**

Let a = 1

Let a = 100

**Reference Error: When JS tries to access the variable which is not in memory or it cannot be access we get reference error.**

Eg1

Console.log(a)

Let a = 1900

Here we cannot access “a” because “a” is hoisted but is in “temporal dead zone”.

E.g2

Var a = 100;

Console.log(b)

Again a reference error that “b” is not defined.

**Scope**

Scope determines the accessibility of the variables/functions.

**Lexical Environment**

Lexical means in sequence/ in hierarchy.

In Lexical Scope, the child scope accesses the variable defined in the parent scope lexically. The lexically means that JavaScript determines the parent scope by looking at **where we created the function** and **not where we invoked it**.

Lexical Scope determines scope of variable based on the physical location of the code.

Lexical scoping means that the accessibility of variables is determined by the position of the variables inside the nested scopes.

the lexical scoping means that inside the inner scope you can access variables of outer scopes.

\*Note: At the global execution context the reference to the parent lexical environment is null.

e.g Function b() is hierarchy /lexically inside function a()

function a()

{

    var car = 10;

    function b()

    {

        console.log(car);

    }

}

a();

Call Stack would be like

Car: 10

A referring to function GEC

B referring to function A

Reference to parent lexical environment

Function b

Function a

Global Execution Context

null

In the console.log(car) statement in function c() the variable “car” doesn’t exist in C’s local memory space to it finds the variable “car” in parent lexical environment. The parent environment for function c is function “a” so it finds the variable car in function a.

Chain of Scope is called as **Scope Chain**.

**Block**

Block {} is used to group the multiple statements.

What is the need of block?

JS will sometimes expect to run a single statement but if we need to run multiple commands we need block.

**Block Scope: Variables accessible only within the block**

Let and const variable defined within the block are only accessible in that block but not outside that block so we say let and const have block scope.

Let and const are hoisted in block scope. Once we are out of the block the block scope gets removed.

A separate space where certain variables (let and const) are accessible.

{

Var a = 10

Let b = 20

Const c = 30

Console.log(a)

Console.log(b)

Console.log(c)

}

Console.log(a)

Console.log(b)

Console.log(c)

The let variable “b” and const variable “c” cannot be accessed since b and c are block scope.

**Shadowing in JS**

var a = 100

{

Var a = 10

Let b = 20

Const c = 30

Console.log(a)

Console.log(b)

Console.log(c)

}

Console.log(a)

Console.log(b)

Console.log(c)

Variable var a is not a block scope so the var a in block scope and var a at the first line both will point to the same memory address so value of var a will become 10.

The following example will throw reference error for last 2 lines.

let a = 100

{

Let a = 10

Let b = 20

Const c = 30

Console.log(a)

Console.log(b)

Console.log(c)

}

Console.log(b)

Output:

10 (let a of block scope)

20 (let b of block scope)

30 (const c of block scope)

100 (let a of outer scope)

It will work correctly.

We cannot shadow a let variable with var because it is illegal shadowing.

Let a = 20

{

Var a = 20

}

We cannot shadow a const variable with var because it is illegal shadowing.

Const a = 20

{

Var a = 20

}

We can shadow let variable using let variable.

Let a =20

{let a = 20}

**Global Scope**

The variables which are declared outside any function are said to have global scope. Global variables can be accessed from anywhere in the program.

**Closure**

A closure is combination of the function bundled together with reference to its lexical environment.

The closure is a function that accesses its lexical scope even executed/invoked outside of its lexical scope

When a function is defined inside another function, the child function has access to the scope of the parent function even if the parent function is already executed and those variables are not accessible from outside the function.

e.g

function x()

{

    var a =7;

    function y()

    {

        console.log(a);

    }

    return y;

}

var z  = x();

z();

//Output

7

So here we have not just returned the function y() but we have returned closure of function y(). So it is not giving errors like variable “a” is not defined.

**Hoisting Examples**

It scans the entire code in JavaScript and stores function object, variables in context.

Rules for hoisting.

Rule 1: Function Declarations are scanned and made available.

The below code will work correctly even if function called is before the function declaration.

**Work correctly**

tipper("5");

function tipper(a)

{

    var bill = parseInt(a);

    console.log(bill + 5);

}

**Work Correctly**

function tipper(a)

{

    var bill = parseInt(a);

    console.log(bill + 5);

}

tipper("5");

Rule 2: Variables declarations are scanned and made undefined.

The below code **will not work** properly and give error because biggerTip is undefined.

biggerTip("200")

var biggerTip = function(a)

{

    var bill = parseInt(a);

    console.log(bill + 15);

}

The below code **will work** properly

var biggerTip = function(a)

{

    var bill = parseInt(a);

    console.log(bill + 15);

}

biggerTip("200")

The below code will print “undefined” because global context knows you have a variable name “nam” but you are accessing it before the declaration.

console.log(nam);

var nam = "Mridul";

**Falsy Values**

Undefined, null, ‘’, 0 , NaN 🡪 all these falsy values are considered as false.

**Coercion**

Implicit conversion of one data type to another data type is coercion.

1. Non-string to String: When we try to add/loosely compare the string and non-string, the non-string get converted to string.

E.g 1

2 + “2”

Output

22

e.g 2

if(2 == "2")

    console.log("true");

output: true

console.log("val: " + true);

output: val: true (bool true converted to string)

1. Non-number to Number:

When we use operations -,/,\*,% the values that are not number are converted into number.

1. Bool to number: true(1) and false(0). When Boolean is added to the number it converts Boolean to number.

console.log(true + true);

output: 2

console.log(true + 5)

output: 6

1. The equality operator (==) It is used to compare the two values irrespective of the datatype. The non-number data gets converted to number.

E.g 1

true == ‘true’ 🡪 this will give false because ‘true’ will be coerced to NaN which is not equal to 1(Boolean true).

e.g 2

true == 1 🡪 TRUE. This will give answer true because Boolean true will become 1 and then 1==1 will give true answer.

**Variables 🡪** Names given to memory locations used to store a data. (containers to store data.)

**Global Variables**

Global variables are Variables outside any function or we can create global variables using window object also.

e.g window.myName = “Mridul”;

Strings in JS are text written within single or double quotes.

JS name must begin with underscore(\_), dollar($), (a-z, A-Z) subsequent characters can be letters, dollar, number or aplhabet.

We cannot re-declare the variable let and const in the same scope.

**JavaScript Arrays** is collection of data irrespective of the type.

**More About the Functions**

**function statement / function declaration**

function a(){}

**function expression**

var b = function(){}

The difference statement and expression lies in hoisting. For function expression memory is allocated for var b with value “undefiend” but that’s not the case for function a() (function statement).

**Anonymous Function**

A function without any name is called anonymous function.

function(){} // Syntax Errors

They are used when functions are used as a value.

Var a = function(){} //correct

**Named Function Expression**

var a = function b()

{

    console.log("hello");

}

a(); //No Error

b(); //Reference Error

**Parameters and arguments**

Function a(param1, param2){} //Here param1 and param2 are paramenters.

a(2 ,3) 🡪 when values are passed to the function those values are called as arguments.

**First Class Functions**

The ability to use function as the value and being able to pass the function as the argument as well as return the function is called first class function.

**Async Functions**

Functions or operations running parallel with the other functions or operations are called [asynchronous](https://www.geeksforgeeks.org/synchronous-and-asynchronous-in-javascript/) functions or operations in JavaScript. Asynchronous JavaScript code requires [Callback](https://www.geeksforgeeks.org/javascript-callbacks/) functions that get executed later after the desired time.

setTimeout() 🡪 is an asynchronous function, meaning that the timer function will not pause execution of other functions in the functions stack. In other words, you cannot use setTimeout() to create a "pause" before the next function in the function stack fires.

**Callback Function**

A function passed as an argument to another function. A callback function can run after another function has finished executing.

Call Stack in inside the JS Engine and JS engine is inside the browser. The call stack doesn’t has a timer with it to run asynchronous functions. We use setTimeout() functions which is available through WebAPI and WebAPI is available in window object.

Once the desired time gets over the callback function goes inside the callback queue/microtask queue serially and whenever the call stack gets empty the event loop fetches the function from the microstask queue/callback queue and pushes it into the callstack in FIFO order.

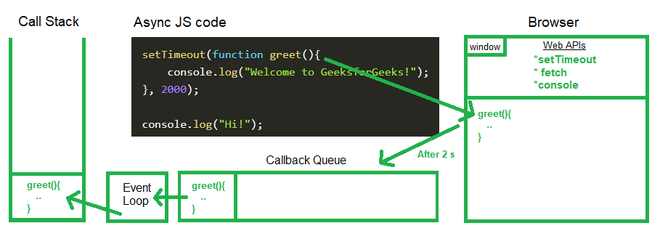
(Microtask Queue has high priority than Callback Queue)

**Event Loop**

**Callback Queue/ Task Queue:** After the timer gets expired, the callback function is put inside the Callback Queue, and the Event Loop checks if the Call Stack is empty and if empty, pushes the callback function from Callback Queue to Call Stack and the callback function gets removed from the Callback Queue. Then the Call Stack creates an Execution Context and executes it

**Microtask Queue:** Microtask Queue is like the Callback Queue, but Microtask Queue has**higher priority**. All the callback functions coming through [*Promises*](https://www.geeksforgeeks.org/javascript-promises/) and [*Mutation Observer*](https://developer.mozilla.org/en-US/docs/Web/API/MutationObserver) will go inside the Microtask Queue. For example, in the case of [.fetch()](https://www.geeksforgeeks.org/javascript-fetch-method/), the callback function gets to the Microtask Queue. Promise handling always has higher priority so the JavaScript engine executes all the tasks from Microtask Queue and then moves to the Callback Queue.

**Starvation of the callback queue:**



Microtask Queue (High priority than callback queue)

**Callback Hell 🡪** The callback function having a callback function having a callback function….. this becomes callback hell. (To avoid the callback hell we need Promises.)

Callback Hell

let production = ()=>{

    setTimeout(()=>{

        console.log("Production has started");

        setTimeout(()=>{

            console.log("the fruit has chopped.");

            setTimeout(()=>{

                console.log(`${stocks.liquid[0]} and ${stocks.liquid[1]} was added`);

                setTimeout(()=>{

                    console.log("the machine has started");

                    setTimeout(()=>{

                        console.log(`ice-cream was placed on ${stocks.holder[0]}`);

                        setTimeout(()=>{

                            console.log(`${stocks.toppings[0]} added as toppings.`);

                            setTimeout(()=>{

                                console.log("Serve the ice-cream");

                            }, 2000);

                        }, 3000);

                    },2000);

                }, 1000);

            },1000)

        }, 2000);

    }, 0000);

}

order(0, production);

**Promises: Promises are used to handle asynchronous operations. They are used when we want to avoid callback hell because it makes the code unmanageable to use. Promises are useful when you have to handle more than one asynchronous task, one after another.**

Benefits:

1. Better handling of asynchronous functions because better readability.
2. Better error handling.

The function which Promise constructor takes is called executor function.

let promise = new Promise(function(resolve, reject){});

Here function(resolve, reject){} is the executor function.

The resolve() and reject() are the callback functions. The executor function either runs one reject() or one resolve() the other reject() or other resolve() are ignored.

The promise has two internal properties **state and result.**

**State:**

Promise has following states:

1. Pending state: Neither promise resolved nor rejecting
2. Fulfilled: Operation related to the promise was completed successfully.
3. Rejected: Operation related to the promise was failed.

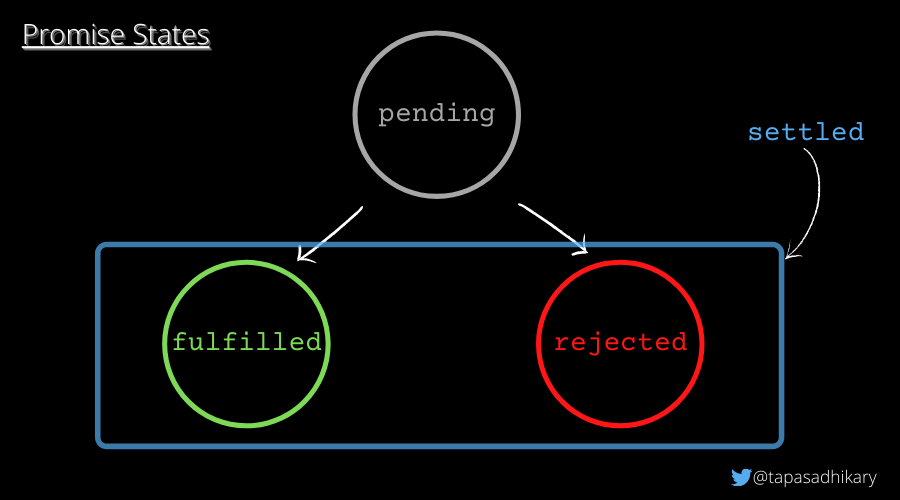
**Result:**

result – This property can have the following values:

* undefined: Initially when the state value is pending.
* value: When resolve(value) is called.
* error: When reject(error) is called.

We cannot access the state and result but can inspect them from the debugger tools.

\*A promise when reject or resolved is called as **settled.**



There are promise handlers .then(), .catch() and .finally()

.then() 🡪 .then() method has two callback functions as arguments one for promise being successful/fulfilled/resolved and one for promise being rejected.

Examples

Eg 1: Here the promise is reject and in .then() the (error)=>{} callback function is called.

let promise = new Promise(function(resolve, reject){

    // resolve("The work is resolved")

    reject("work rejected")

});

promise.then(

    (result)=>{console.log(result);},

    (error)=>{console.log(error);}

);

e.g 2: here resolve callback function is called from the promise and further in .then() function (result)=>{} is called.

let promise = new Promise(function(resolve, reject){

    resolve("The work is resolved")

    reject("work rejected")

});

promise.then(

    (result)=>{console.log(result);},

    (error)=>{console.log(error);}

);

e.g 3: If we car only interested when promise is rejected then this is the syntax.

promise.then(

    null,

    (error)=>{console.log(error);}

);

e.g 4: If we are only interested in promise being resolved.

promise.then(

    (result)=>{console.log(result);}

);

**.catch()** 🡪 This method is used for the error handling purposes. Whenever the promise is being rejected the catch method is called.

.catch() method internally calls .then(null, errorHandler) method.

let promise = new Promise((resolve, reject)=>{

  reject("Promise is rejected")

})

.catch((err)=>{console.log(err);})

**.finally()** method is called either when the operation is resolved or rejected. It is not necessary that after .finally(), .then() and .catch() functions cannot be called.

Use case 1:

let promise = new Promise((resolve, reject)=>{

  // resolve("Promise is resolved"),

  reject("promise is rejected")

})

let error\_consumer = ()=>{

  promise.then(()=>{console.log("Promise resolved successfully");})

  .finally(()=>{console.log("Finally called");})

  .then(()=>{console.log("promise after then");})

  .catch((err)=>{console.log(err);})

}

error\_consumer();

output:

Finally called

promise is rejected

Promise Chain

promise.then() 🡪 It is returning the promise with the start value state: pending and result: undefined.

Example:

Promise variable has promise.

First\_consumer is calling a .then() function on the promise and returning 1 in the callback function is promise is resolved but this return value will be used my next .then() only and the variable “first\_consumer” will not have the value 1, it is storing a promise with status: pending and result: undefined.

let promise = new Promise((resolve, reject)=>{resolve("promise is resolved")});

let first\_consumer = promise.then((result)=>{return 1});

console.log(first\_consumer)

output:

Promise { <pending> }

When the first .then() method returns the promise the next .then() method uses that promise and next .then() method will use the recent method and this phenomenon is called Promise Chain.

Some use cases:

Example 1:

The first .then() is using the value resolve(“Promise is resolved”) of the previous promise. So it prints “Promise is resolved” and the second .then() prints “undefined” because its result has the value undefined as previous .then() doesn’t return/resolve anything.

So the .then() function is dependent on the previous .then() function so we say they are chained.

let promise = new Promise((resolve, reject)=>{

  resolve("Promise is resolved")

})

.then((result)=>{console.log(result);})

.then((result)=>{console.log(result);})

Output:

Promise is resolved

undefined

Example 2:

Here both the promise.then() are treated as separate promise means that they are not chained.

let promise = new Promise((resolve, reject)=>{

  resolve("Promise is resolved")

})

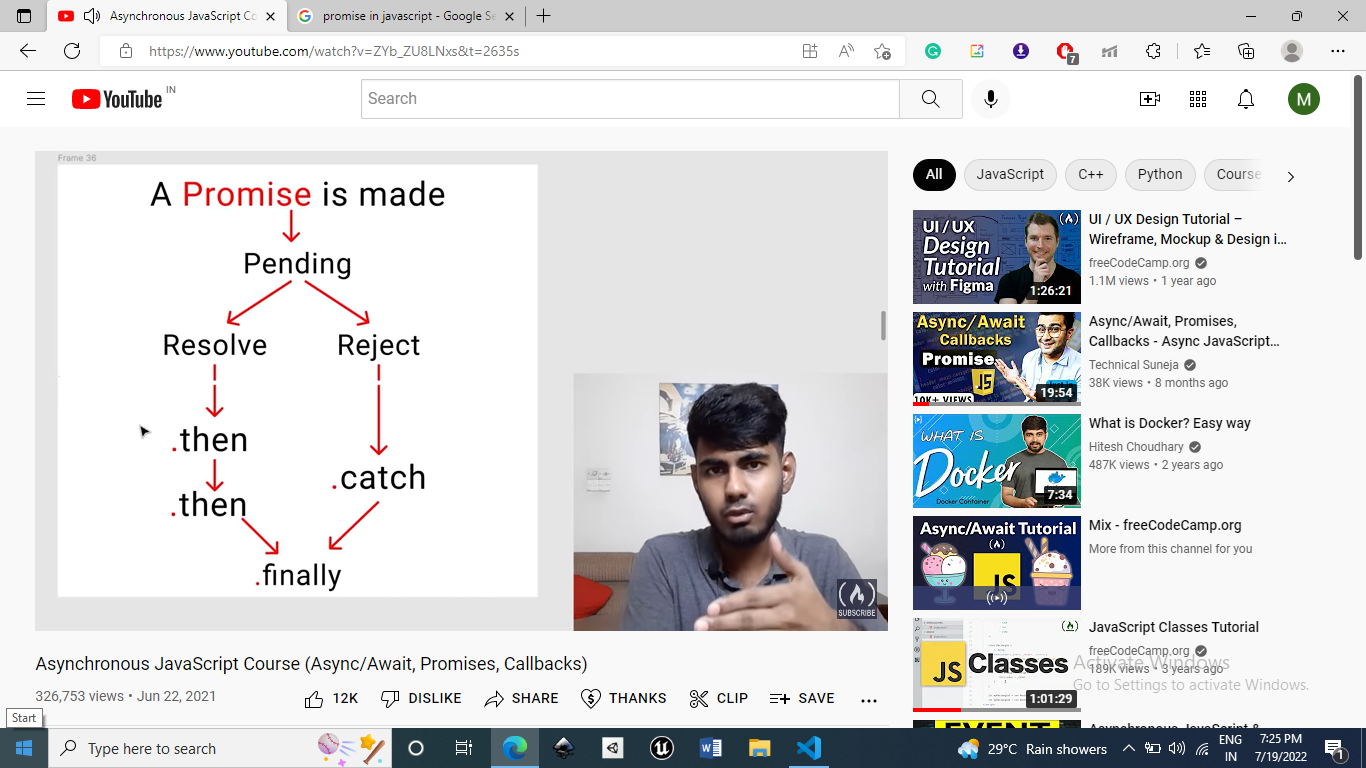
promise.then((resolve)=>{console.log(resolve);})

promise.then((resolve)=>{console.log(resolve);})

Output:

Promise is resolved

Promise is resolved



**How to handle multiple promises?**

**Promise.all() 🡪 This is also returning the promise with status: pending and result: undefined.**

Promise.all([promises]) accepts a collection (for example, an array) of promises as an argument and executes them in parallel.

This method waits for all the promises to resolve and returns the array of promise results. If any of the promises reject or execute to fail due to an error, all other promise results will be ignored.

If all the promises are resolved then it will return the result of all the promises in an array.

Examples

Ex 1 (all promises getting resolved)

Here promise.all([array of promises]). When

var XMLHttpRequest  = require('xhr2');

function getPromise(URL) {

  let promise = new Promise(function (resolve, reject) {

    let req = new XMLHttpRequest();

    req.open("GET", URL);

    req.onload = function () {

      if (req.status == 200) {

        resolve(req.response);

      } else {

        reject("There is an Error!");

      }

    };

    req.send();

  });

  return promise;

}

const BULBASAUR\_POKEMONS\_URL = 'https://pokeapi.co/api/v2/pokemon/bulbasaur';

const RATICATE\_POKEMONS\_URL = 'https://pokeapi.co/api/v2/pokemon/raticate';

const KAKUNA\_POKEMONS\_URL = 'https://pokeapi.co/api/v2/pokemon/kakuna';

let promise\_1 = getPromise(BULBASAUR\_POKEMONS\_URL);

let promise\_2 = getPromise(RATICATE\_POKEMONS\_URL);

let promise\_3 = getPromise(KAKUNA\_POKEMONS\_URL);

Promise.all([promise\_1, promise\_2, promise\_3])

.then((result)=>{console.log(result.length);})

Output:

3

Example 2: Any one promise is not resolved. Here we have passed a bad URL in 2nd promise so it rejects the promise and .catch() method handles the error and print s it.

const BULBASAUR\_POKEMONS\_URL = 'https://pokeapi.co/api/v2/pokemon/bulbasaur';

const RATICATE\_POKEMONS\_URL = 'https://pokeapi.co/api/v2/pokemon/raticate';

const KAKUNA\_POKEMONS\_URL = 'https://pokeapi.co/api/v2/pokemon/kakuna';

let promise\_1 = getPromise(BULBASAUR\_POKEMONS\_URL);

let promise\_2 = getPromise(RATICATE\_POKEMONS\_URL + 'e');

let promise\_3 = getPromise(KAKUNA\_POKEMONS\_URL);

// --------------Promise.all()

Promise.all([promise\_1, promise\_2, promise\_3])

.then((result)=>{console.log(result.length);})

.catch(err => console.log(err))

Output:

There is an Error!

**Promise.any([promises])**

Promise.any([promises]) - Similar to the all() method, .any() also accepts an array of promises to execute them in parallel. This method doesn't wait for all the promises to resolve. It is done when any one of the promises is settled.

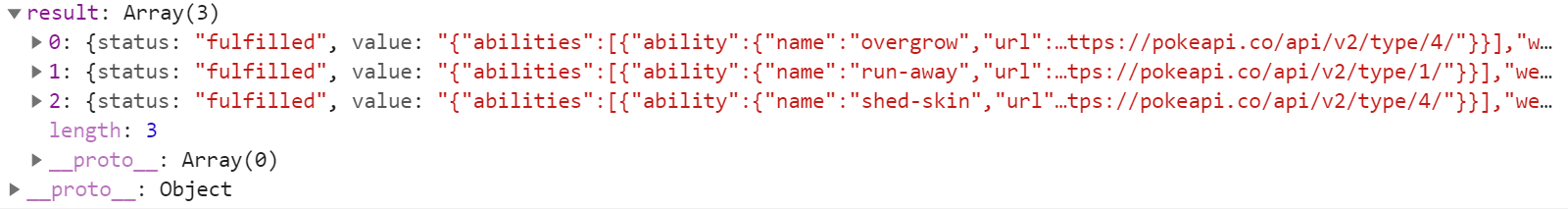
It will return the result of the first promise which is resolved.

**Promises.allSetteled([promises])**

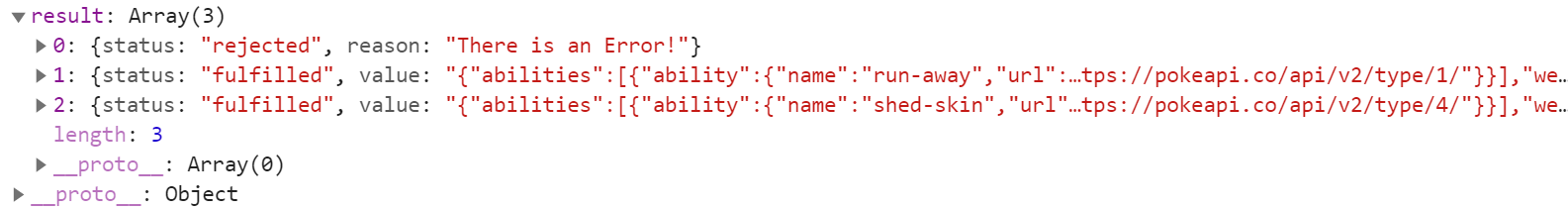
The method waits for all the promises to get settled(resolved/ rejected). It returns result of all promises as an array of objects and each result will have status information (fulfilled/rejected) of the promise and value, if promise fulfilled and if promise rejected then reason for promise rejected.

Example;

When all the promises are resolved.



When some promises are resolved and some are rejected.



**Promise.race([promises])**

It will return the result of that promise that gets settled quickly (the promise that gets settled first

Output will be the result of the promise which gets settled first.

**Async/Await**

Writing async keyword before any function makes the function return the promise.

async function myFunction() {  
  return "Hello";  
}

is same as

function myFunction() {  
  return Promise.resolve("Hello");  
}

try and catch blocks

**try:** We use “try” keyword to create a try block and keep our code/logic within that block.

**Catch:** “catch” block is used to handle any error. (The error which is throw by the code in try block.)

**Finally:** The code in finally block executes weather the code in try bock has thrown error or not.

Example

myFunction() is returning the promise and we are using the .then() handler on that promise.

async function myFunction()

{

  return "Function is resolved."

}

myFunction().then(

  function(value){console.log(value)},

  function(err){}

)

**Await:** It could be used within the async block only. It makes the code wait until the promise settels and returns a result. It only makes the async block wait

**OOPS in JavaScript**

Objects are the any real world entity which has properties and methods.

Objects in JS is group of key: value pairs.

Creating Objects in JavaScript.

Method 1: object literal

const circle ={

    radius: 1,

    location: {

        x:1,

        y:1

    },

    draw: function()

    {

        console.log('draw');

    }

};

When we use object literal syntax then JS uses object constructor to create the object.

Method 2: Creating objects using constructor

**this keyword**

“this” points to the global object. In browser it is window and in node environment it is just a global object {}.

function Circle(radius)

{

    this.radius = radius;

    this.draw = function(){

        console.log("Radius is: " + this.radius);

    }

}

let circle2 = new Circle(2);

When we use “new” keyword to create the object or call any function of the object at that time “this” points to the current object (the one which we just created).

Method 3: Factory Function method

//Factory Function

function createCircle(radius)

{

    return {

        radius,

        draw: function()

        {

            console.log("draw");

        }

    };

}

When we define

Let x = {} JS translates it to let x = new Object()

JS uses Function() constructor to create the function.

let circleFunction = new Function('radius', `

this.radius = radius;

this.draw = function()

{

    console.log("draw");

}

` )

const circle4 = new circleFunction(4);

circle4.draw();

**Primitives / Value Type**

Number, String, Boolean, Symbol, undefined, null

let x = 20;

let y = x;

x= 10;

Print(x); // 10

Print(y); // 20

**Reference Type**

Object, Function, Array

let x = {value: 20};

let y = x;

x.value = 10;

Print(x); //{value: 10}

Print(y); //{value: 10}

Primitives are copied by their value and reference types are copied by their reference.

**Adding/Removing Properties**

Adding properties:

//Method 1: circle2.location = {x: 1, y:1}

//Method 2: circle2['color'] = "red";

It is better to use method 2 if we want to name any property like

{ my-name: ‘Mridul’}

Circle2.my-name (This is wrong).

Circle2[‘my-name’] (This is correct)

For deleting

delete objectname.propertyname;

**Class:**

Class is a template/blueprint to create object. Class is an user defined datatype that contains data and method acting on that data.

**constructor():** It is the first function to be called whenever we instantiate the object.

**NODE JS**

Node.js is an open-source and cross-platform runtime environment built on Chrome’s V8 JavaScript engine for executing JavaScript code outside of a browser.

It provides an event-driven, non-blocking (asynchronous) I/O and cross-platform runtime environment for building highly scalable server-side applications using JavaScript.

Event – Driven: The event driven system means the control flow of the program is depending

PHP file request

1. Sends the task to the computer's file system.
2. Waits while the file system opens and reads the file.
3. Returns the content to the client.
4. Ready to handle the next request.

Node JS handling the file

1. Sends the task to the computer's file system.
2. Ready to handle the next request.
3. When the file system has opened and read the file, the server returns the content to the client.

What is a Module?

A modules are like JavaScript libraries that consists of set of functions which we can use in our application.

JavaScript files are considered as modules.

To include a module we use require() function with name of module. Require() function is local to each module.

**Module Wrapper Function**

Function(exports, require, model, \_\_filename, \_\_dirname)

Node must not be used for CPU-intensive apps it must be used for data intensive and network intensive apps.

In node we have global object “global”.

Path module

const path = require('path');

basename\_ext = path.basename('D:\\NodeJS\\content\\subfolder\\text.txt');

//output: text.txt

basename\_no\_ext = path.basename('D:\\NodeJS\\content\\subfolder\\text.txt', '.txt');

//output: text

console.log(basename\_ext);

console.log(basename\_no\_ext);

dirname\_ = path.dirname('content\\subfolder\\text.txt');

console.log(dirname\_);

//content/subfolder

console.log(path.extname('app.js')); //.js

/\*

path.format(pathObject)

The path.format() method returns a path string from an object.

priority rules:

1. pathObject.root is ignored if pathObject.dir is provided

2. pathObject.ext and pathObject.name are ignored if pathObject.base exists

\*/

//root is ignored as dir is provided.

let path\_1 = path.format({

    root: '/ignored',

    dir: '/home/user/dir',

    base: 'file.txt'

  });

console.log(path\_1); // /home/user/dir\file.txt

// ext is ignored as base is provided.

let path\_2 = path.format({

    root: '/',

    base: 'file.txt',

    ext: 'ignored'

  });

console.log(path\_2);

// name + ext is used as base not provided.

let path\_3 = path.format({

  root: '/',

  name: 'file',

  ext: '.txt'

});

console.log(path\_3);

OS module

const os = require('os');

let totalMemory = os.totalmem(); //returns total memory in bytes.

let freeMemory = os.freemem(); //unit bytes

console.log(`Total Memmory is ${totalMemory} and freeMemory is ${freeMemory}`);

HTTP Methods

GET() 🡪 to get the data.

POST()🡪 to create the data.

PUT()🡪

PATCH() 🡪

DELETE() 🡪 to delete the data.