***JavaScript OOP***

OOP => Objcet Oriented Programming.

OOP is A pradigm or style of code.

**Constructor Function**

function User(i, u, s) {

  this.id = i;

  this.username = u;

  this.salary = s;

}

let userOne = new User(105, "Bilal", 6000);

let userTwo = new User(125, "Hassan", 800);

let userThree = new User(128, "anas", 900);

console.log(userOne, userTwo, userThree);

SomeTimes we need a **‘blueprint’ مخطط** for creacting many Objects of the same ‘type’.

The way to create an ‘object type’, is to use an **object constructor function.**

In the exaple above , function User() is an object constructor function .

Objects of the same type are created by calling the constructor function with new keyword.

**Notes :**

It is considered good practice to name construtor funcitons with an upper-case first letter.

The constuctor function is very usefull especially when you have to create many obj.

***About this***

In a constructor function this does not have a value. It is a substitute for the new object. The value of this will become the new object when a new object is created .

**Constructor Function : new syntax**

In Es2015, a new syntax appear :

So for converting code from the old form to new one press on lamp after pressing 3 dots

The code will become like this :

class User {

    constructor(i, u, s) {

        this.id = i;

        this.username = u;

        this.salary = s;

    }

}

**Instance of : (نمودج من )**

For checking constructor , return boolean (true or false )

console.log(userOne instanceof User); // true

let literalString = "This is a literal string";

let stringObject = new String("String created with constructor");

literalString instanceof String; // false, string primitive is not a String

stringObject instanceof String; // true

literalString instanceof Object; // false, string primitive is not an Object

stringObject instanceof Object; // true

**constructor:**

The **construtor** property retunrs the function that created Object prototype.

console.log(userOne.constructor);

/\*

class User {

  constructor(i, u, s) {

    this.id = i;

    this.username = u;

    this.salary = s;

  }

}

\*/

**Deal with properties and Methods :**

Pls foucs in this code :

class User {

  constructor(i, u, s) {

    // properties

    this.id = i;

    this.username = u || "Unknown";

    this.salary = s < 6000 ? s + 500 : s;

    this.msg = function () {

      return `Hello ${this.username}, Your Salary is ${this.salary}`;

    };

  }

  // methods

  writeMsg() {

    return `Hello ${this.username}, Your Salary is ${this.salary}`;

  }

}

function User1(i, uN, s) {

  this.id = i;

  this.userName = uN;

  this.salary = s;

}

let userOne = new User(105, "Bilal", 6000);

let userTwo = new User(125, "", 800);

console.log(userOne, userTwo, userOne.msg());

// User {id: 105, username: 'Bilal', salary: 6000, msg: ƒ}

//User {id: 125, username: 'Unknown', salary: 1300, msg: ƒ}

//'Hello Bilal, Your Salary is 6000'

/\*

ƒ () {

      return `Hello ${this.username}, Your Salary is ${this.salary}`;

    }

\*/

console.log(userTwo.msg); // Native code

console.log(userOne.writeMsg); // native code

// method

console.log(userTwo.writeMsg()); // Hello Unknown, Your Salary is 1300

**Update Properties and built in consructores :**

class User {

  constructor(n, a, s) {

    this.name = n;

    this.age = a;

    this.salary = s;

  }

  updateName(newName) {

    this.name = newName;

  }

}

let userOne = new User("Bilal", 17, 5000);

console.log(userOne.name); // Bilal

userOne.updateName("Osama");

console.log(userOne); // Osama

in the example above, we used updateName method for updating name for user one, this method allow us to strat updating names dynamically .

hey : In JavaScript, there are primitive values and objects. Primitive values include strings, numbers, booleans, null, and undefined. Objects include arrays, functions, and other objects like the String object.

let stringOne = "Elzero";

let stringTwo = new String("Elzero");

console.log(typeof stringOne, typeof stringTwo); // string Object

but see that also :

console.log(stringOne instanceof String, stringTwo instanceof String);

// false true

I know you are confused right now, but i am sure that you don’t understand the qu, instance of mean with type of object is instanced the var ,

So sure stringTwo is instance of string object , but stringOne is sure it isn’t instanced of string object because he isn’t in origin a str .

**Class Static proproperties and methods**

class User {

  count = 0;

  constructor(n, a, s) {

    this.name = n;

    this.age = a;

    this.salary = s;

  }

}

let userOne = new User("Bilal", 17, 5000);

let userTwo = new User("Ahmed", 20, 6500);

console.log(userOne.count); // 0

console.log(User.count); // undefined

/\* we got undefined bacause the property isn't static wich mean the property isn't

specified for the class 'user' it self , so if want to access it, you have to choices,

the first one is to choose it from the object created, the second choice is to make it

static for all user, for that you can access it from user class\*/

but now if we make « count » property static , 0 will become undefined and the opposite is true .

class User {

  static count = 0;

  constructor(n, a, s) {

    this.name = n;

    this.age = a;

    this.salary = s;

  }

}

let userOne = new User("Bilal", 17, 5000);

let userTwo = new User("Ahmed", 20, 6500);

console.log(userOne.count); // undefined

console.log(User.count); // 0

**A small practice :**

class User {

  static count = 0;

  constructor(n, a, s) {

    this.name = n;

    this.age = a;

    this.salary = s;

    User.count++;

  }

  static membersCount() {

    return `${User.count} members created `;

  }

}

let userOne = new User("Bilal", 17, 5000);

let userTwo = new User("Ahmed", 20, 6500);

let userThree = new User("sayed", 30, 5500);

console.log(User.membersCount()); // 3 members created

**Class Inheritance :**

Imagine with me that you have to create many users but you know that there is some properties and methods are in commun with all users that you have to create, so here you have to work with inheritance system or you will repeat yoursel adding to that, with all users will not be any relation between users so if you want to change something you have to change for all users .

// Parent Class

class User {

  constructor(id, username) {

    this.i = id;

    this.u = username;

  }

  sayHello() {

    return `Hello ${this.u}`;

  }

}

// Derived Class

class Admin extends User {

  constructor(id, username, permissions) {

    super(id, username);

    this.p = permissions;

  }

}

let adminOne = new Admin(55, "Bilal", true);

console.log(adminOne.sayHello()); // Hello Bilal

// Derived Class From Admin

class Superman extends Admin {

  constructor(id, username, permissions, abbility) {

    super(id, username, permissions);

    this.a = abbility;

  }

}

let superp = new Superman(89, "tawa", true, 70);

console.log(superp);

// Superman {i: 89, u: 'tawa', p: true, a: 70}

To create a class inhertance, use the extends keyword.

A class created with a class inheritance inherits all methods from another class :

class Car {

  constructor(brand) {

    this.carname = brand;

  }

  present() {

    return `I have a ${this.carname}`;

  }

}

class Model extends Car {

  /\*

  arguments they could have had different names,

  as long as they are passed in the correct order

   \*/

  constructor(bra, mod) {

    super(bra);

    this.model = mod;

  }

  show() {

    return `${this.present()}, it's model is ${this.model}`;

  }

}

let car = new Model("Dacia", "Duster");

console.log(car.show());

// I have a Dacia, it's model is Duster

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 get access to the parent’s propeties and methods .

*Summary :*

So the first thing with defined the class and the class where we want to extend our new class

***Class …… extends …..***

Second we define all arguments the new one and old one ,

***Constructor (old arg , new arg ) .***

Then inside constructor we use super() keyword for calling parent’s constructor and get access form all its properties and methods .

***Super (old arg )***

And that it .

**Class Encapsulation**

Encapsulation is one of oop concepts ,Imagine that you have repository, instead of leaving customers who take their good from repo, you have two workers who do this, who are Setters (set goods in repo), and getters (get goods from repo ) .

So to do this, you have first to prevent customors from accessing the rebo,

So remove the access to their goods.

And in programmation that mean make var private .

***Private variables*** are variables who can be created only in class bodies and can only be accessed and changed by code that is also part of the same scope, but ***public vars*** can be accessed and changed by any code.

In the next code, i’ll use encapsulation :

class User {

  // defind privatie vars

  #password;

  constructor(u, e, p) {

    this.userName = u;

    this.email = e;

    // make password private

    this.#password = p;

  }

  setPass(newValue) {

    this.#password = newValue;

  }

  getPass() {

    let pi = this.#password.slice(1, -1).split("").length;

    return (

      this.#password.slice(0, 1) + "\*".repeat(pi) + this.#password.slice(-1)

    );

  }

}

let userOne = new User("Bilal", "Billa@gmail.com", "alo123");

console.log(userOne.getPass()); //a\*\*\*\*3

userOne.setPass("BBBBBBBBB");

console.log(userOne.getPass()); // B\*\*\*\*\*\*\*B

The meaning of Encapsulation, is to make sure that « sensitive » data is hidden from users. To ahcieve this, you must :

* Declare class variables / attributes as **private**
* Use **get** and **set** methods to access and update the value of a **private** variable .

Getters and setters aren’t methods but they are computed properties ( is a value can be derived from other stored value), so also the syntax will change.

**Using *methods* :**

class User {

  constructor(name, email) {

    this.name = name;

    this.email = email;

  }

  sayHello() {

    return `Hello ${this.name}`;

  }

  showInfo() {

    return `Name: ${this.name} \nEmail: ${this.email}`;

  }

}

let user1 = new User("Bilal", "Bilal@gmail.com");

console.log(user1.name, user1.email);

console.log(user1.showInfo());

/\*

Name: Bilal

Email: Bilal@gmail.com

\*/

**Using *Getters and setters* :**

class User {

  #name;

  constructor(name, email) {

    this.#name = name;

    this.email = email;

  }

  sayHello() {

    return `Hello ${this.name}`;

  }

  set ChangeName(newName) {

    this.name = newName;

  }

  get showInfo() {

    return `Name: ${this.#name} \nEmail: ${this.email}`;

  }

}

let user1 = new User("Bilal", "Bilal@gmail.com");

// pay attention to syntax there is no parentheses

console.log(user1.showInfo);

/\*

Name: Bilal

Email: Bilal@gmail.com

 \*/

user1.ChangeName = "Ahmed"; // pay attention to Syntax

console.log(user1.name); // Ahmed

**why Encapsulation :**

* Better control of class attributes and methods
* Class attributes can do read-only (if you only use the get ), or write-only (if you only use th eset method).
* Flexible : the programmmer can change one part one the conde without affecting other parts   
  Increased security of data

**why Using Getters And Setters :**

* it gives simpler syntax
* it is useful for doing things behind-the-scenses

**Important info :**

Get is used to get private info, after being prepared and packaged. For example : when you order password, Get prepare it by encrypting, or phone number by showing only the last chiffres.

Set is used to update private info, after the validation of the new values

***Prototype***

We have allready spoken about prototype and speciay about **Object.create**

Which is a method to create a new object, using an existing object as the prototype of the newly created object.

let obj1 = {

  name: "Bilal",

  age: 17,

};

// create obj2 and make obj1 its prototype

let obj2 = Object.create(obj1);

obj2.work = true;

obj2.name = "Ahmed";

// as you see obj2 doesn't have age property

console.log(obj2); // {work: true, name: 'Ahmed'}

// but if there is no property obj2 will inherit it from its prototype

console.log(obj2.age); // 17

// but if it exists will take his own property

console.log(obj2.name); // Ahmed

A prototype is a blueprint for an object instance.

Blueprint : mokhatat , instance : namodaj

*\* All javaScript objects inherit properties and methods from a prototype* .

Add To prototype chain and Extend constructors Features

class User {

  constructor(id, username) {

    this.i = id;

    this.u = username;

  }

  sayHello() {

    return `Hello ${this.u}`;

  }

}

let userOne = new User(100, "Elezro");

console.log(User.prototype);

/\*

constructor: class User

sayHello: ƒ sayHello()

[[Prototype]]: Object

\*/

As you see in User.prototype you can see all component of a class constructor and its methods + you can see also the prototype where you can inherit all its propterties and methods, in the case the prototype is Object so you can inherit the normal Object properties

let admin = Object.create(userOne);

admin.u = "Ahmed";

console.log(admin);

/\*

u: "Ahmed"

[[Prototype]]: User

\*/

In the previous case the prototype is User Class, if we open prototype we will see all things that we can inherit :

1. **[[Prototype]]: User**
   1. **i: 100**
   2. **u: "Elezro"**

the question here is , known that we have access to prototype , is it possible to add methods to a constructor and inherit it from subElement ?

the answear : all yes .

User.prototype.welcome = function () {

  return `welcome ${this.u}`;

};

console.log(admin.welcome()); // welcome Ahmed

As you see we add welcome method to User constructor but admin obj can inherit it .

The idea here is to add some functionalities to some built in function, by adding some methods to its constructors

let client = "Bilal";

String.prototype.addMr = function () {

  return `Mr.` + this.valueOf();

};

console.log(client.addMr()); // Mr.Bilal

so client is a string and we know String is its constructor so we add the method to its prototype, and you can see it if you log Sring constructor

second example :

Object.prototype.love = function () {

  return `I love ${this.name || "you"}`;

};

console.log(admin.love()); // i love you

As you see this is a big prototype chain

1. ***User {u: 'Ahmed'}***
   1. **u: "Ahmed"**
   2. **[[Prototype]]: User**
      1. **i: 100**
      2. **u: "Elezro"**
      3. **[[Prototype]]: Object**
         1. **welcome: *ƒ ()***
         2. **constructor: *class User***
         3. **sayHello: *ƒ sayHello()***
         4. **[[Prototype]]: Object**
            1. **love: *ƒ ()***
            2. **constructor: *ƒ Object()***
            3. **hasOwnProperty: *ƒ hasOwnProperty()***
            4. **……..**

The previous chain, I think it explain very well the concept of inheritance in prototype no matter how chain is big

**Object Meta Data and Descriptor**

The **object.defineProperty()** method defines a new property directly on an object, or modifies an existing property on an object

Returned value : the object modified

const myObject = {

  a: 1,

  b: 2,

};

console.log(

  Object.defineProperty(myObject, "c", {

    writable: true,

    enumerable: true,

    configurable: true,

    value: 3,

  })

);

// {a: 1, b: 2, c: 3}

The thing special in this method it’s ***the full control***

Why ???

***Writable :*** determines whether you can assign a new value to a property. If writable is true, you can assign a new value to the property.if writable is false, the propety is read-only and connot be assigned.

***Enumereable :*** determins whether a property can be accessed in a loop,if enumerable is flazse, the property will be skipped in loops

***Configurable :*** controls ablility of deleting the propety and controls the ability of changing any param in the decriptor ( writable, enumerable )

Object.defineProperty(myObject, "c", {

    writable: false,

    enumerable: true,

    configurable: true,

    value: 3,

  })

);

// {a: 1, b: 2, c: 3}

myObject.c = 200; // the value will not be modified because \_ writable is false

console.log(myObject.c); // 3

\*\*

Object.defineProperty(myObject, "c", {

    writable: true,

    enumerable: false,

    configurable: true,

    value: 3,

  })

);

// {a: 1, b: 2, c: 3}

for (let key in myObject) {

  console.log(key);

}

// a

// b

// \*\*\*\*\*\*\* Note : as you see 'c' is kipped in loop

\*\*

Object.defineProperty(myObject, "c", {

    writable: true,

    enumerable: true,

    configurable: false,

    value: 3,

  })

);

console.log(delete myObject.c); // false

// As you see, the property will not be deleted because configurable is false

++ we can’t redifine the property of modify any parameter that control it .

Object.defineProperty(myObject, "c", {

  writable: true,

  enumerable: true,

  configurable: false,

  value: 3,

});

console.log(myObject);

Object.defineProperty(myObject, "c", {

  writable: false,

  enumerable: false,

  configurable: true,

  value: 3,

});

// now we got an error becasue we redfine property and we changed enumarable

// property and configurable property

**Note :**

we will not get an error if we redifine and change writable property only

Syntax :

**Object.defineProperty(obj, prop, descriptor)**

\* the default value of descriptor params is false, but sure value is required.

Descriptor contain (enumerable and confiuguable and writable) and value

-- And for accessing the property descriptor in an objec , use the Object.getOwn.PropertyDescriptor()

const myObject = {

  a: 1,

  b: 2,

};

Object.defineProperty(myObject, "c", {

  configurable: true,

  value: 3,

});

Object.defineProperty(myObject, "d", {

  configurable: true,

  value: 4,

});

Object.defineProperty(myObject, "e", {

  configurable: true,

  value: 5,

});

console.log(myObject);

console.log(Object.getOwnPropertyDescriptor(myObject, "a"));

// {value: 1, writable: true, enumerable: true, configurable: true}

console.log(Object.getOwnPropertyDescriptor(myObject, "d"));

// {value: 4, writable: false, enumerable: false, configurable: true}

The **Object.Own.PropertyDescriptor()** returns all own property descriptors of a given object

console.log(Object.getOwnPropertyDescriptors(myObject));

/\*

a: {value: 1, writable: true, enumerable: true, configurable: true}

b: {value: 2, writable: true, enumerable: true, configurable: true}

c: {value: 3, writable: false, enumerable: false, configurable: true}

d: {value: 4, writable: false, enumerable: false, configurable: true}

e: {value: 5, writable: false, enumerable: false, configurable: true}

[[Prototype]]: Object

\*/

***Time and Date***

UTC = **U**nit ed **T**ime **C**ordinated

Tangier, Morocco  🡺 UTC +1

-- To Track time you need starting point

-- **Unix time** in computer science is the number of seconds since : January 1 ,1970

For log it in the console :

let firstDate = new Date(0);

// Thu Jan 01 1970 01:00:00 GMT+0100 (West Africa Standard Time

## Unix time is a system for representing a point in time. It is the number of seconds that have elapsed since January 1st, 1970 00:00:00 UTC.

// decalre date that represent numbers of millseconds from Unix Epoch

let date = Date.now();

console.log(date); // 1686571198864

// trasform its value to years

date = date / 1000 / 60 / 60 / 24 / 365;

console.log(date); // 53.48

// verify result '1970'

let diffrence = 2023 - date;

console.log(diffrence); // (approximitive value) : 1969.519 ;

new date() creates a date object with the current date and time

const d = new Date();

new date(date string) creates a date Object from a **date string**

const d = new Date("October 13, 2014 11:13:00");

**Note :**

\*\* Date stirng formats will be described in the next chapter.

----------------------------------------------

let dateNow = new Date();

let birthday = new Date("30 May 2006 ");

console.log(dateNow); // Mon Jun 12 2023 13:12:19 GMT+0100 (West Africa Standard Time)

console.log(birthday); // Tue May 30 2006 00:00:00 GMT+0100 (West Africa Standard Time)

let diff = dateNow - birthday;

// transform result to years

diff = diff / 1000 / 60 / 60 / 24 / 365;

console.log(diff); // 17.04;

console.log(`I am ${diff.toFixed()} Years Old`);

// I am 17 yeas Old

**New Date (year, month, … )** create a date object with a specified date and time.

7 numbers specify year, month, day, hour, minute, second, and millsceond (in that order )

**Note :**

\*\* JavaScript counts months from 0 to 11 :

January = 0 .

December = 11.

Specifying a month higher than 11, will not result is an error but add the overflow to the next year.

Specifying :

const d = new Date(2023, 12, 23);

console.log(d); // Tue Jan 23 2024

is the same as :

const e = new Date(2024, 0, 23);

console.log(e); // // Tue Jan 23 2024

Look at the result they are the same .

const d = new Date(2023, 5, 39);

console.log(d); //Sun Jul 09 2023

const e = new Date(2023, 6, 9);

console.log(e); // //Sun Jul 09 2023

Notes :

JavaScript stores dates as number of miliseconds since January 01, 1970

**Zero time is January 01, 1970 00:00:00 UTC**.

One day (24 hours) is 86 400 000 milliseconds.

**Date Get Methods**

|  |  |
| --- | --- |
| Method | Description |
| getFullYear() | Get year as four digit number (yyyy) |
| getMonth() | Get month as a number (0-11) |
| getDate() | Get date as a number (1-31) |
| getDay | Get weekday as a number (0-6) |
| getHours() | Get hour (0-23) |
| getMinutes() | Get minute (0-59) |
| getSeconds() | Get second (0-59) |
| getMilisconds() | Get miliscond (0-999) |
| getTime() | Get time (miliseconds since unix Epoch) |

Testing this methods :

let dateNow = new Date();

console.log(dateNow.getFullYear()); // 2023

console.log(dateNow.getMonth()); // 5 (meaning month juin 6)

console.log(dateNow.getDate()); // 12

console.log(dateNow.getDay()); // 1 (meaning monday)

console.log(dateNow.getHours()); // 14

console.log(dateNow.getMinutes()); // 35

console.log(dateNow.getSeconds()); // 40

console.log(dateNow.getMilliseconds()); // 328

console.log(dateNow.getTime()); //1686577002768

in Js, the first day of the week (day 0 ) is Sunday

You can use an array of names, and getDay() to returns weekday as a name :

const days = [

  "Sunday",

  "Monday",

  "Tuesday",

  "Wednesday",

  "Thursday",

  "Friday",

  "Saturday",

];

console.log(days[dateNow.getDay()]); //Monday

**Date Set Methods**

Set Date methods let you se date values (years, months, days, hours, minutes, seconds, miliseconds) for a Date object

|  |  |
| --- | --- |
| Method | Description |
| setFullYear() | Set year (optionally month and day) |
| setMonth() | Set month as a number (0-11) |
| setDate() | Set date as a number (1-31) |
| setDay | Set the weekday as a number (0-6) |
| setHours() | Set the hour (0-23) |
| setMinutes() | Set the minute (0-59) |
| setSeconds() | Set the seconds (0-59) |
| setMilisconds() | Set the milisconds (0-999) |
| setTime() | Set the time (miliseconds since unix Epoch) |

Testing set methods :

let dateNow = new Date();

console.log(dateNow); // Tue Jun 13 2023 12:10:03

dateNow.setDate(15);

console.log(dateNow); // Thu Jun 15 2023 12:10:03

dateNow.setFullYear(2015);

console.log(dateNow); // Mon Jun 15 2015 12:10:03

dateNow.setMonth(4);

console.log(dateNow); // Fri May 15 2015 12:10:53

dateNow.setHours(19);

console.log(dateNow); // Fri May 15 2015 19:10:45

dateNow.setMinutes(58);

console.log(dateNow); // Fri May 15 2015 19:58:37

Note :

In setDate() the overflow will be add, and 0 mean the last day in the previous month -1 mean 29 , -2 mean 28 …… (i wish you understand me )

let dateNow = new Date();

console.log(dateNow); // Tue Jun 13 2023

dateNow.setDate(0);

console.log(dateNow); // Wed May 31 2023

as we said setFullYear(), we can set date and month and also hear the overflow ……

let dateNow = new Date();

console.log(dateNow); // Tue Jun 13 2023

dateNow.setFullYear(2023, 14, 37);

console.log(dateNow); // Sat Apr 06 2024

// Hey bro try to understand

**Formating Date and Time :**

|  |  |
| --- | --- |
| Type | Example |
| ISO Date | ‘2015-03-25’ (The International Standard Oragnization ) |
| Short Date | ‘03/25/2015’ |
| Long Date | ‘Mar 25 2015’ or ’25 Mar 2015’ |

**Date.parse()** returns the number of milliseconds between the date and Juanuary 1,1790

let dateNow = new Date(0);

console.log(Date.parse("2006-05-30")); //Thu Jan 01 1970

console.log(1148947200000 / 1000 / 60 / 60 / 24 / 365 + 1970); // 2006.43

**Tracking Operations Time :**

/ Start Time

let start = new Date();

// operation

for (let i = 0; i < 10000; i++) {

  let div = document.createElement("Div");

  let divText = document.createTextNode("Div Number : " + i);

  div.appendChild(divText);

  document.body.appendChild(div);

}

// End Time

let end = new Date();

// Operation Duration

console.log(end - start); // 86

// the time depends on many parameters

Generator Function :

* Generator Fuction run its code when required
* Generator are Iterable

In JavaScript, a generator function is a special type of function that can be paused and resumed *during its execution*. It allows you to create an iterator that generates a squence of values, *one at time, without having to generate them all at once and store them in memory.*This can be useful for working with large data sets or for creating sequences that are too large to fit in memory .

In JavaScript, a generator function is defined using the « **function\***» syntax and the « **yield** » keyword is used to pause the function and return a value to the iterator.

const genFunction = function\* () {

  console.log("Hello");

  console.log("I am the last ");

  yield "Yield 1 Value";

  console.log("World");

  yield "Yield 2 Value";

  console.log("And Galaxy");

  yield "Yield 3 Value";

  console.log("best");

};

const gObj = genFunction();

console.log(gObj);

output :

1. ***genFunction {<suspended>}***
   1. **[[GeneratorLocation]]: main.js:6**
   2. **[[Prototype]]: Generator**
      1. **[[Prototype]]: Generator**
         1. **constructor: GeneratorFunction {prototype: Generator, Symbol(Symbol.toStringTag): 'GeneratorFunction', constructor: *ƒ*}**
         2. **next: *ƒ next()***
         3. **return: *ƒ return()***
         4. **throw: *ƒ throw()***
         5. **Symbol(Symbol.toStringTag): "Generator"**
         6. **[[Prototype]]: Object**
   3. **[[GeneratorState]]: "suspended"**
   4. **[[GeneratorFunction]]: *ƒ\* ()***
   5. **[[GeneratorReceiver]]: Window**
   6. **[[Scopes]]: Scopes[3]**

console.log(gObj.next());

/\*

I am the last

{value: 'Yield 1 Value', done: false}

\*/

Continue the iteration using next

console.log(gObj.next());

console.log(gObj.next());

console.log(gObj.next());

console.log(gObj.next());

/\*

World

main.js:25 {value: 'Yield 2 Value', done: false}

And Galaxy

{value: 'Yield 3 Value', done: false}

14 best

{value: undefined, done: true}

{value: undefined, done: true}

\*/

So we can say when a generator function is called, it does not call the function instead it returns a generator object, and next() method will start the excution till the yield operator , next() method return an object, which has two keys

Yield itself is capable of returning any value.

Generator are iterable, so we can use loops

for (let o of gObj) {

  console.log(o);

}

Output :

Hello

8 I am the last

21 Yield 1 Value

10 World

21 Yield 2 Value

12 And Galaxy

21 Yield 3 Value

14 best

As it is iterable you can use the speread operator, it will create an array with returned values

const gObj = [...genFunction()];

console.log(gObj);

console.log(gObj); // (3) ['Yield 1 Value', 'Yield 2 Value', 'Yield 3 Value']

**Note :**

I think : The iteration has memory if status is done, so nothing will be displayed if you start another type of iteration on it .

A small practice :

function\* genaraten() {

  let n = 0;

  while (true) {

    n++;

    yield n \* n;

  }

}

let nNumber = genaraten();

this function will generate unlimite numbers

console.log(nNumber.next().value); // 1

console.log(nNumber.next().value); // 4

console.log(nNumber.next().value); // 9

console.log(nNumber.next().value); // 16

// .....

We can’t loop because the computer will stop working because we will make in unifinite loop, for loop on all result we have to limit it using a max value

function\* genaraten(max) {

  let n = 0;

  while (n < max) {

    n++;

    yield n \* n;

  }

}

let nNumber = genaraten(6);

for (let number of nNumber) {

  console.log(number);

}

// 1 4 9 16 25 36

* **next()** : this method is used to advance the generator to the next value in the sequence.When the gene func is called, it returns an iterator object, and here we use ‘next()’ to get advanced in the iteration, the next method return an object with 2 properties , ‘’value’’ which contains the current value of generator, and ‘’done’’, which is a boolean indication whether the generator has finished generting values
* **retrun()** : This method is used to terminate the generator early. When the ‘return()’ method is called, the gen guction is terminated the the ‘’done’’ property of the iterator object is set to ‘true’. The method can also be used to return a final value from the gen

const genFucntion = function\* () {

  yield 1;

  yield 2;

  return "Stop the iteration";

  yield 3;

  yield 4;

};

let genObj = genFucntion();

console.log(genObj.next()); // {value: 1, done: false}

console.log(genObj.next()); // {value: 2, done: false}

console.log(genObj.next()); // {value: 'Stop the iteration', done: true}

console.log(genObj.next()); /// {value: undefined, done: true}

As you see the iteration stopped at yield 2, and in the last example too we use return to show the last value which is ‘ *stop the iteration*’

Delegate Generator Function:

Imagine that you have to create a gen func that one to yield to other gen func

function\* generateNumbers() {

  yield 1;

  yield 2;

  yield 3;

}

function\* generateLetters() {

  yield "A";

  yield "B";

  yield "C";

}

function\* generateAll() {

  yield generateNumbers();

  yield generateLetters();

}

let genAll = generateAll();

console.log(genAll.next()); // {value: generateNumbers, done: false}

console.log(genAll.next()); // {value: generateNumbers, done: false}

console.log(genAll.next()); // {value: undefined, done: true}

look to the values, we want the yields of the gen func, for that :

function\* generateAll() {

  yield\* generateNumbers();

  yield\* generateLetters();

}

let genAll = generateAll();

console.log(genAll.next()); // {value: 1, done: false}

console.log(genAll.next()); // {value: 2, done: false}

console.log(genAll.next()); // {value: 3, done: false}

console.log(genAll.next()); // {value: 'A', done: false}

console.log(genAll.next()); // {value: 'B', done: false}

console.log(genAll.next()); // {value: 'C', done: false}

console.log(genAll.next()); // {value: undefined, done: true}

the solution is \*(asterisk symbol), and this same thing for arrays

function\* generateAll() {

  yield\* generateNumbers();

  yield\* [4, 5, 6];

  yield\* generateLetters();

  yield\* ["D", "E", "F"];

}

let genAll = generateAll();

for (let n of genAll) {

  console.log(n);

}

// 1 2 3 4 5 6 A B C D E F

And this method is name ‘delegate’

Modules and Export

Modules are a way to organize and reuse code. They allow you to break up into smaller. More manegeable

Think of a module as a box of code that contains functions, variables, or other pieces of code. You can then use these modules in other parts of your code by importing them. This makes it easier to maintain you code, as you can update or change a module without affecting the rest of you codebase.

- First it very necessary to declare the type of any external js file :

<script src="module.js" type="module"></script>

<script src="main.js" type="module"></script>

**Export**

**Named Export**

Let us create a file name module.js, and fill it with the things we want to export.

You can create named exports two ways. *In-line individually*, or *all at once* at the bottom

* ***In-line individually :***

export let a = 10;

export let arr = [1, 2, 3, 4];

export function saySomething() {

  return "Something";

}

* ***All at once :***

et a = 10;

let arr = [1, 2, 3, 4];

function saySomething() {

  return "Something";

}

export { a, arr, saySomething };

**Default Export**

Export something in a file that can have a name or can be anonymous, and you can also import it with any name you want it.

And for that , You can only have one default export in a file.

export default function () {

  return "Bilal yes ";

}

**Import**

You can import modules into a file in two ways, based on if they are nmaed exports or default exports.

Named exports are constructed using curly braces. Default exports are not.

- **Import named exports :**

import { a, arr, saySomething } from "./module.js";

- **Import default exports :**

import Bilal from "./module.js";

in import defalut exports, you can choose any name you want it

- **and for both :**

import Bilal, { a, arr, saySomething } from "./module.js";

***Note :***

You can add ilyas in export and import using ‘’as’’ keyword

import { a as num, arr, saySomething } from "./module.js";

console.log(num); // 10

the previous example we have var who has exported as ‘a’ but when we import it , we give it an alias

the same in export :

export { a as var1, arr, saySomething };

a small example of defalut exporting and its importing :

// module.js

export default function () {

  return "Bilal yes ";

}

// main.js file

import bilal from "./module.js";

console.log(bilal()); // Billa yes

and you can use the wild card « \* » for importing all things in a module .

import \* as all from "./module.js";

**From Assignment :**

The **toLocaleString()** method returns a Date object as string, using locale settings.

The default language depends on the local setup on your computer.

*Syntax*

Date.toLocaleString(Locales, options)

|  |  |
| --- | --- |
| Locles | Optional. Which language specific format to use.  Legal Values :  ar-SA Arabic (Saudi Arabia)  en-US US English  en-GB British English  fr-FR Standard French (especially in France)  fr-BE Belgian French fr-CA Canadian French |
| Options | Optional. An object where you can set some properties.  **Legal Properties : | | Legal Values :**   |  |  | | --- | --- | | dateStyle : | **full**  **long**  **medium**  **short** | | timeStyle : | **full**  **long**  **medium**  **short** | | hour12 : | **false**  **true** | | Weekday : | **long**  **short**  **narrow** | | Year : | **2-digit**  **Numeric** | | month : | **2-digit**  **long**  **narrow**  **numeric**  **short** | | day : | **2-digit**  **Numeric** | | hour : | **2-digit**  **Numeric** | | minute : | **2-digit**  **numeric** | | second : | **2-digit**  **numeric** | | timeZoneName : | **long**  **short** | |

Let Try it :

let secondDate = new Date().toLocaleString("en-Us", {

  year: "numeric",

  month: "long",

  day: "numeric",

  hour: "numeric",

  minute: "numeric",

  timeZoneName: "short",

});

console.log(secondDate); // June 18, 2023 at 7:09 PM GMT+1

// don't forget you can log whatever you want , "full control here"

**JSON**

**What is JSON ?**

* JavaScript Object Notation
* Format For Sharing Data Between Serveur And Client
* JSON Derived From JavaScript
* Alternate To XML
* File Extension is .json

**Why JSON ?**

* Easy to Use and Read
* Used by most programming languages and its frameworks
* You can convert JSON object to Js obect and vice Versa

Both JSON and XML can be used to receive data from a web serveur .

The follwing JSON and XML examples both define an employees object, with an array of 3 employees :

**JSON Example :**

{

  "employees": [

    { "firstName": "John", "lastName": "Doe" },

    { "firstName": "Anna", "lastName": "Smith" },

    { "firstName": "Peter", "lastName": "Jones" }

  ]

}

**XML Example :**

<employees>

  <employee>

    <firstName>John</firstName> <lastName>Doe</lastName>

  </employee>

  <employee>

    <firstName>Anna</firstName> <lastName>Smith</lastName>

  </employee>

  <employee>

    <firstName>Peter</firstName> <lastName>Jones</lastName>

  </employee>

</employees>

**JSON vs XML**

JSON                      VS      XML

===========================================================

= LightWeight                    = Heavier

= Does not Use Tags              = Using Tags

= Shorter                        = Not short

= Can Use Arrays                 = Cannot Use Arrays

= Not Support Comment            = Support Comments

============================================================

**JSON Syntax :**

As we now JSON do sharing data , so there is no function or any other calculation

**JSON Syntax :**

* Data Added inside Curly Braces { }
* Data added with key/Value
* Key should be double quotted
* Data Seperated by comma
* Square Brackets [] For Arrays
* Curly Braces {} For Objects

**Available Data Types :**

* String
* Number
* Object
* Array
* Bollean Values
* null

{

  "String": "Bilal",

  "Number": 17,

  "Skills": ["HTML", "CSS", "JS"],

  "object": {

    "Tangier": "Bni Makada",

    "Rabat": "Bwitat"

  },

  "boolean": true,

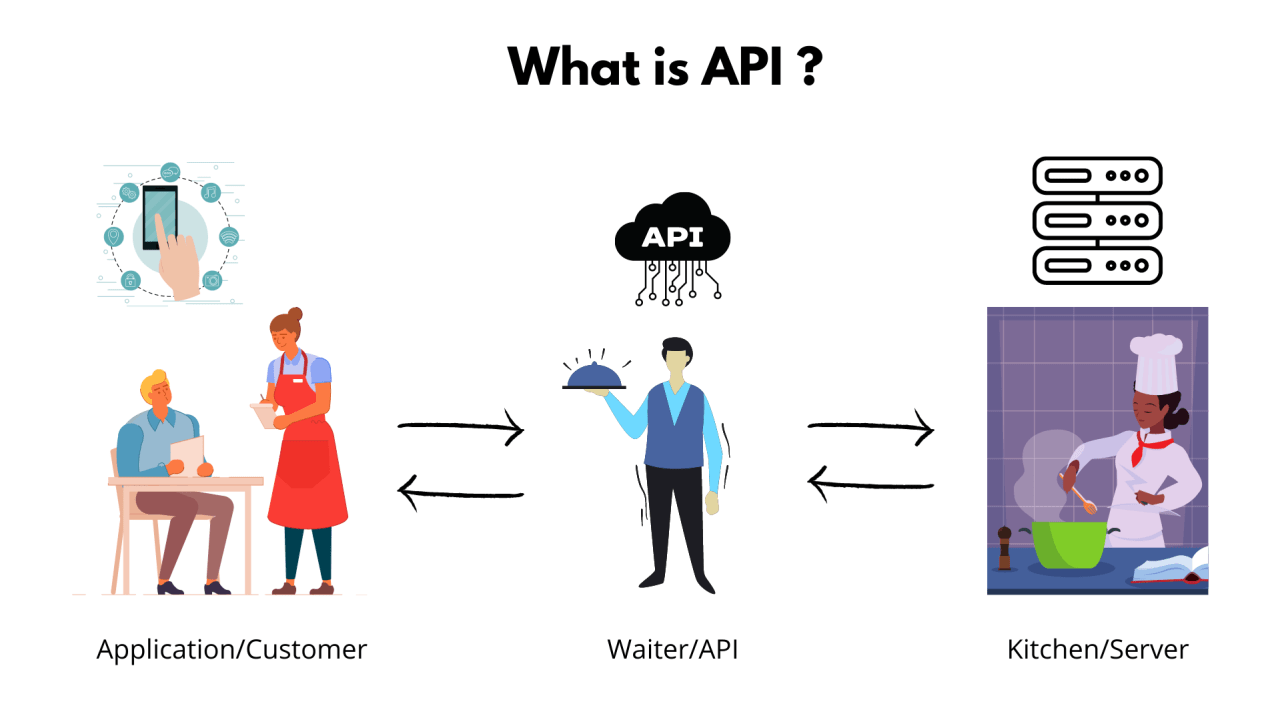
  "null": null

}

And for hard word there is many JSON editor

What Is API ?

API Stands For **A**pplication **P**rogramming **I**nterface.

API give us some information as JSON object

Parse And Stringify

JSON.parse ==> Convert Text Data To JS Object

JSON.stringify ==> Convert JS Object To JSON

const myJsonObjectFromServeur = '{"Username":"Elzero","Age":39}';

console.log(typeof myJsonObjectFromServeur); // String

console.log(myJsonObjectFromServeur); // {"Username":"Elzero","Age":39}

const myJsObject = JSON.parse(myJsonObjectFromServeur);

console.log(typeof myJsObject); // object

console.log(myJsObject); // {Username: 'Elzero', Age: 39}

// Ok, we have recieved the data from the serveur

// And Now we gone update data and then we will send the data to serveur

myJsObject.Username = "Bilal";

myJsObject.Age = 17;

const myNewJsonObjectFromServeur = JSON.stringify(myJsObject);

console.log(myNewJsonObjectFromServeur); // {"Username":"Bilal","Age":17}

Asynchronous vs Synchronous Programming

* **Synchronous**
* Operations Runs is Sequence
* Each Operation Must Wait For the previous one to complete
* **Story From Real Life** : Imagine that you want to take your ticket, but in the ticket place there is a queue full of people, so you have to wait ur turn and all the people who are before you take their tacket too, and then you take you own.

// Synchronous (sync)

console.log(1);

console.log(2);

alert("Operation");

console.log(3);

* **Asynchronous**
* Operations Runs in Parallel
* This Means that an operaton can occur whille another one is still being processed
* **Story From Real Life** : Imagine That you go to a restaurant, their many waitters, so here you have not to wait other people to order but you can order even that they are no order yet, so there is no relation, i hope u understand

// Asynchronous (Async)

console.log(1);

console.log(2);

setTimeout(() => console.log("3 min"), 3000);

console.log(3);

Call Stack And Web API

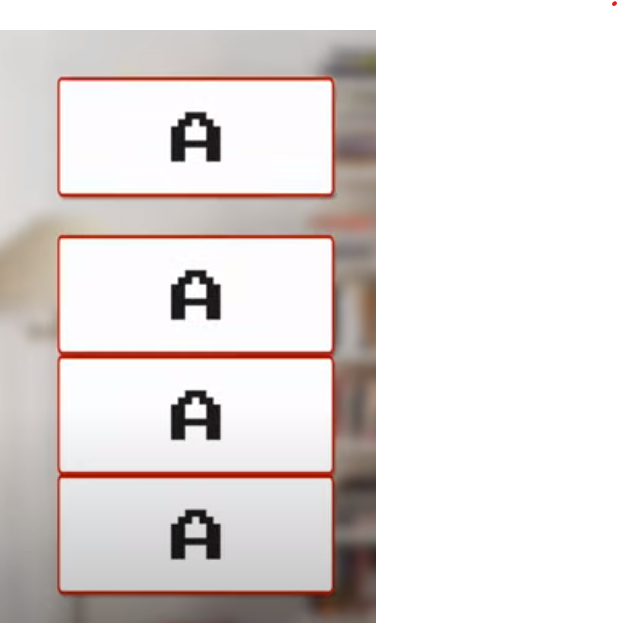
First, JavaScript is a Single Threaded Programming Language, meaning it has one Thread ==> it has one Call Stack ==> it Deal with one thing at a time

So you will never find JS Deal with 2 fonction in the same time .

A **call stack** is a type of data structure, the most important thing that makes it special is that when we want to put in it data, we always put it at the top of the list.

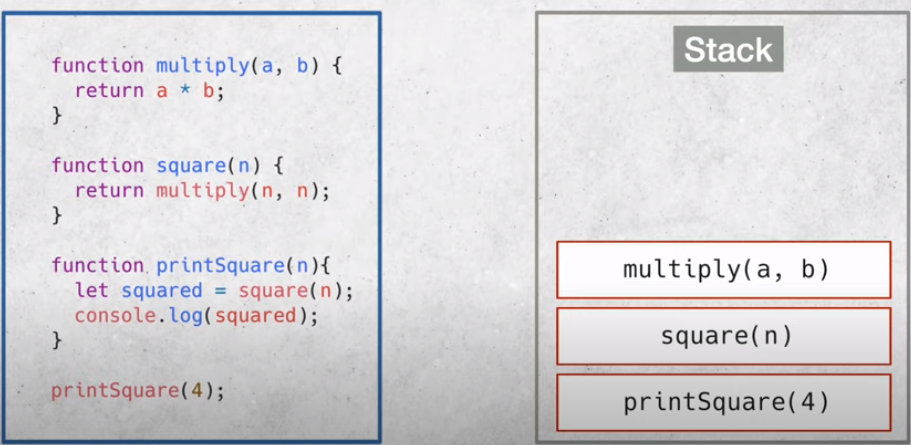
And also when want want to withdraw (سحب) some infomation, we pull it from the above.

So it follows ***Last-In-First-Out principle*** (LIFO)

Just think its very logic, think of it like a deep box, that hold books, when you put some books any, in pulling you have to pull out the last or the recent one first .

I think the img explain very much the principle

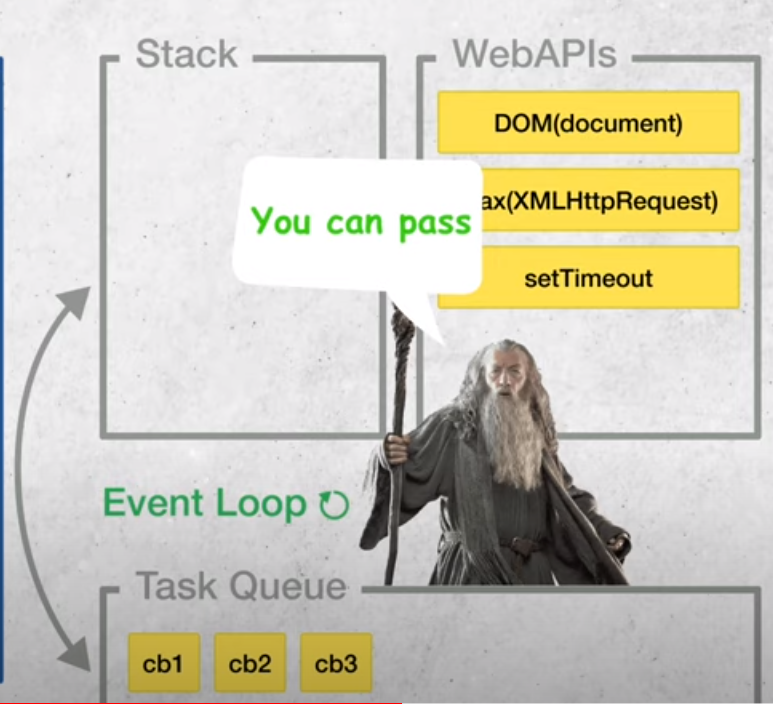
Js depend on call stack to know which function it has to run now and when finishing runing that function it has to run which function in next .

Let take an example : 

So as when js run printSquare() it put it to the stack but it find Square() and also it put it into the stack and init it find multiply() and also it put it into the stack and now it has all mission ordered so it will start by multiply and then it will remove it from the stack and then it will move to the next function, after excution it will remove it and now it arrived to the last one, it will run the last then it will find console.log it will excute it and then it will remove it and remove printSquare() also from the stack .

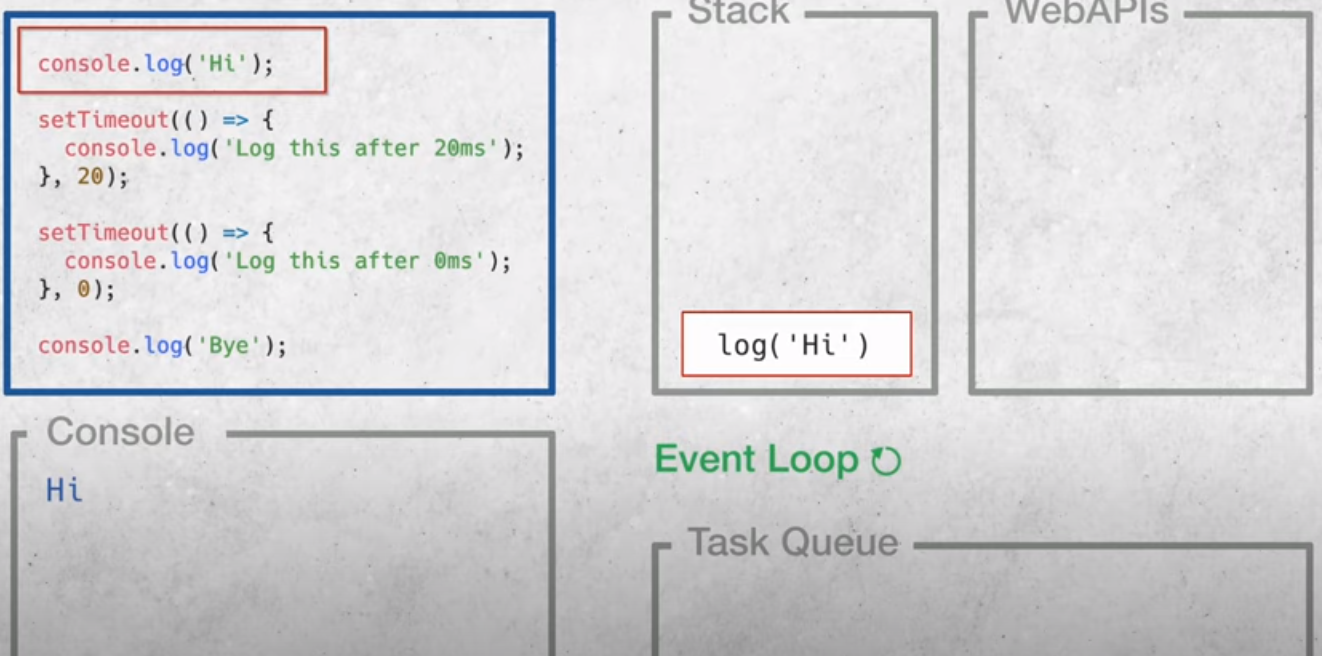
As we know the code Above is sync, because each function have to wait the other one to complete, so this how call stack organize this mission and this how it handle it . But imagine that we use async code how stack will deal with it, espicially since JS in single thread, and any async code must be excuted out of this thread for not to stop or delay any process, and here comes the role of the browser, that provides for us WebAPIs and also **Call Back Queue** where webAPIs call back putted for get ready to enter to call stack ,(*it really a real queue just think, a queue of call back that wait here its turn for accessing the stack and be processed*), the Queue also a type of data structure as the stack, and the most thing that make it special is

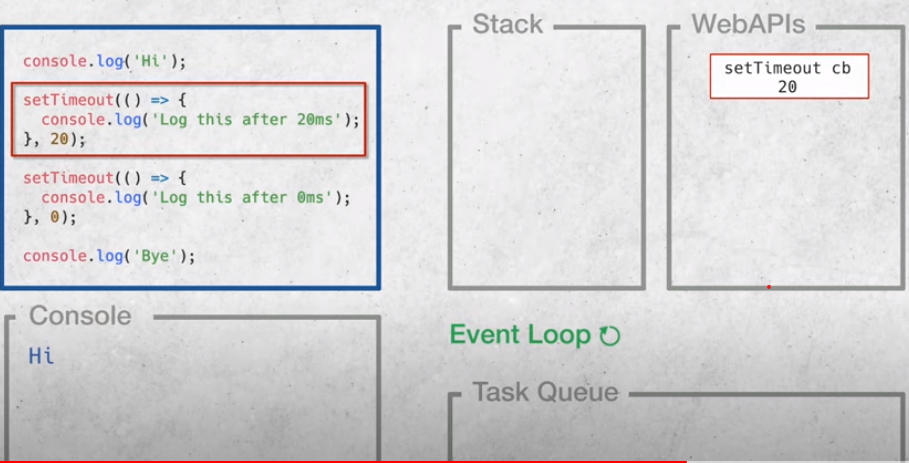
***First-In-First-Out principle*** (FIFO)



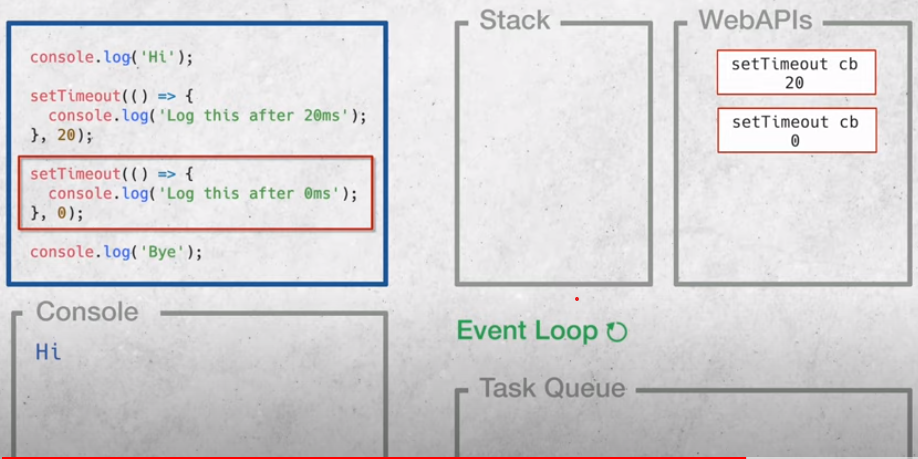
You can think of **Event Loop** as gateKeeper because he is responsible for entering the call from the call back queue to the call back stack by checking if the stack is free, if the stack free he allow the call to pass

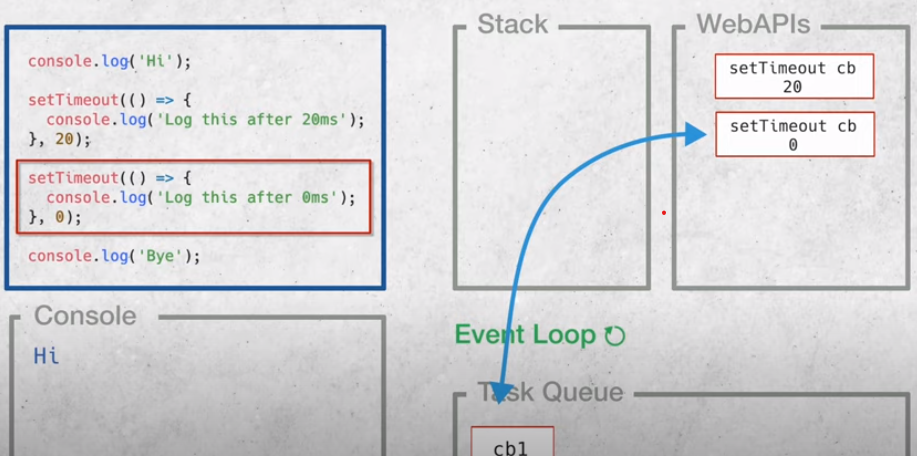
Let take an example :

First, console.log will enter the stack and then it will excuted and then it will be romove from the stack

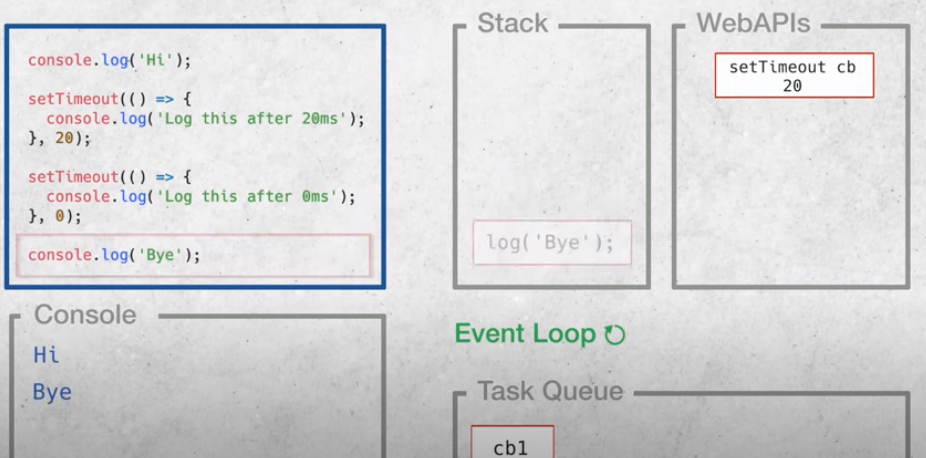
Then setTimeout() will enter to the stack who will send it to webAPI after find it an async code 

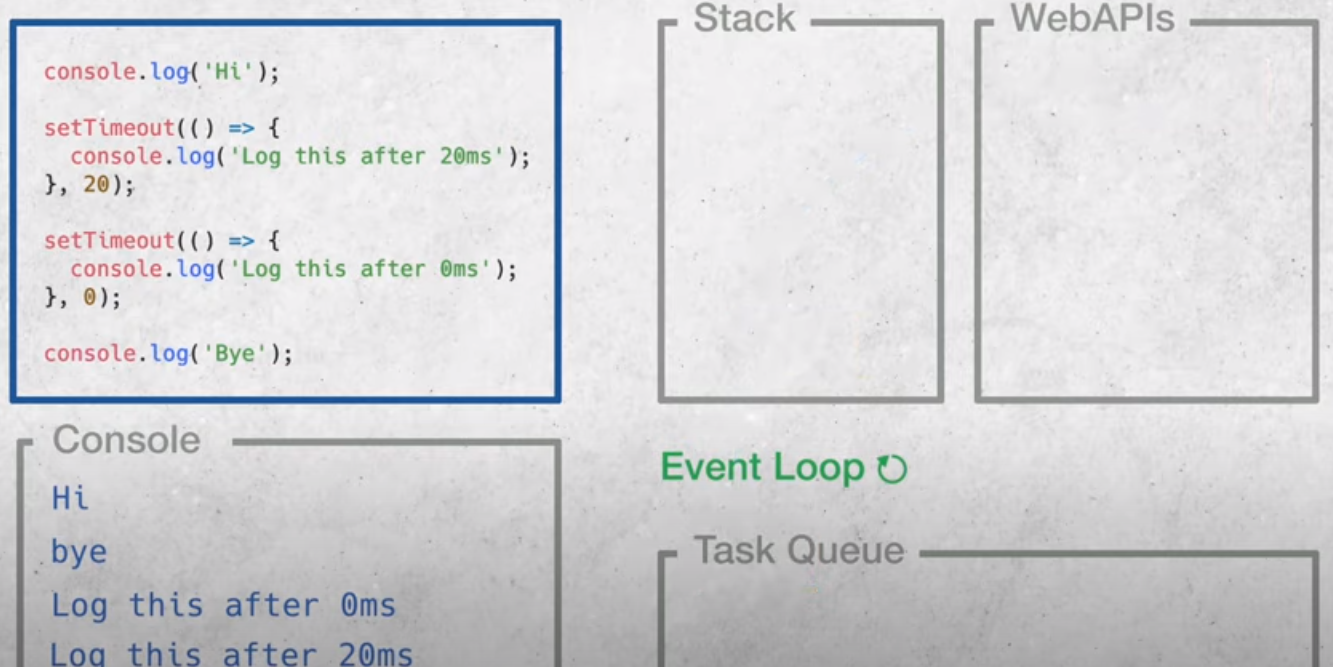
And then it will move to the next part of code, and what we named it meaning Async code will run byside the main code.

The stack will send the second timeout to the browser because it is also async code 

This timeout has no delay so webAPI we send it directly to call Queue

The event loop will not allow cb1 to enter to the stack because it still busySo it will move to the next part of code Console.log will enter the stack and it will removed from the stack after that.



And now Event Loop will allow cb1 to enter the stack and be excuted, and after the dalay cb2 will enter the call Queue also and then it will passed to the stach passing by Event Loop

**Une image contenant capture d’écran, texte, diagramme, ligne

Description générée automatiquement**

**Concurrency Model js**

Foucus in the next code pls :

console.log("One");

setTimeout(() => {

  console.log("Four");

}, 20);

setTimeout(() => {

  console.log("Three");

}, 0);

console.log("Two");

/\* Expected Output in the console:

One

Two

Three

Four

\*/

See the next also :

let myVar = 100;

myVar += 100;

console.log(myVar);

in the previous code we declare a var and then we log it in the console

console.log(myVar);

let myVar = 100;

myVar += 100;

Here we will get an error, because we have accessed myVar before intitialzation ,

setTimeout(() => {

  console.log(myVar);

}, 0);

let myVar = 100;

myVar += 100;

// Expected output : 200

And the think the previous code explain everything about all the story that i heve told you about .

What is AJAX ?

AJAX = **A**synchronous **J**avascript **A**nd **X**ML

AJAX is not a programming language

AJAX just uses a combination of :

* A browser built-in **XMLHttpRequest** object (to request data from a web server)
* JavaScript and HTML DOM (to display or use the data )

AJAX allows web pages to be updated asynchronously by exchanging data with a web server behind the scenes. This means that it is possible to update parts of a web page, without reloading the whole page.

**Status Code :**

Their many Status codes but the general rule is :

1. information responses (100 – 199)

2 . successful responses (200 – 299) (right)

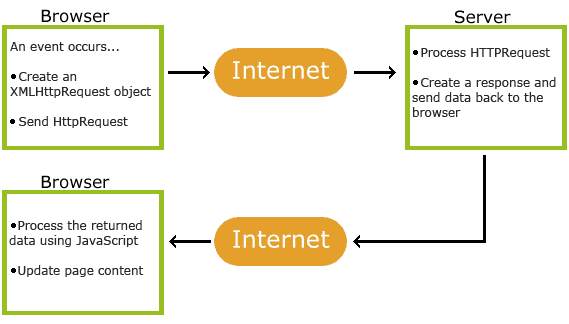
3 . redirection messages (300 – 399 )

4. Client error resposes (400 – 499 )

5. Server error resoponsese ( 500 – 599 )

AJAX Use "XMLHttpRequest" object to Interact with The Serveur, this object contain many handsome methdos

**How AJAX Works :**



As known XMLHttpRequest object is used to exchange Data With a sever

Send a Request To a Server

To send a request to a server, we use the open() and send() methods of the XMLHttpRequest object :

|  |  |
| --- | --- |
| Method | Description |
| Open (method, url , async ) | Method : Specifies the type of request  Method: the type of request : GET or POST  url : the serveur (file) location  external or internal  async : true = async or false = sync |
| Send() | Sends the request to the server |

The second two things that are the condition that we have to check :

**- Ready State => Status of The request**

[0] : Request Not initialized

[1] : server connection established

[2] : Request Received

[3] Processing Request

[4] Request is finished and response

**- Status => Status of The response**

[200] : Response is Successful

[404] Not Found

**Note :**

The **readyState** property returns the status of the current document

The **readyState** is read-only

A Small practice for this, using GitHub API

/ --- title

let pageTitle = document.createElement("h1");

pageTitle.textContent = "MY Repos";

document.body.appendChild(pageTitle);

pageTitle.style.cssText =

  "text-align:center;font-size:35px;font-Family:monospace;margin-bottom:70px";

// --------

// Start Code

let myRequest = new XMLHttpRequest();

myRequest.open("GET", "https://api.github.com/users/bilal-friend/repos");

myRequest.send();

myRequest.onreadystatechange = function () {

  console.log(myRequest.readyState);

  console.log(myRequest.status);

  // check that we are the right condition

  if (this.readyState === 4 && this.status === 200) {

    // console.log(this.responseText);

    let jsData = JSON.parse(this.response);

    console.log(jsData);

    // create the container

    let container = document.createElement("div");

    // add id to the container

    container.id = "container";

    // loop on all repos

    for (let i = 0; i < jsData.length; i++) {

      // just for testing in the console

      console.log(jsData[i].url);

      // Create Box

      let box = document.createElement("div");

      // add className

      box.className = "box";

      // -- Create Box's Title

      let title = document.createElement("h1");

      let titleText = document.createTextNode(jsData[i].name);

      // -- create link

      let link = document.createElement("a");

      // adding Url to the link

      link.href = jsData[i].html\_url;

      // open in new window

      link.target = "\_blank";

      // add text for link

      link.textContent = "==> Go Repos";

      // append the text to the title

      title.appendChild(titleText);

      // append the title to the box

      box.appendChild(title);

      // append link to the box

      box.append(link);

      container.appendChild(box);

      // style the link

      link.style.cssText =

        "color:black;text-decoration:none;font-Fimily:sans-serif;font-size:19px";

      // Style the box

      box.style.cssText =

        "border: 1px solid dodgerBlue;width:400px;padding:10px;text-align:center;height: 150px;color:dodgerBlue;font-Family:cursive;border-radius:10px";

    }

    document.body.appendChild(container);

    // Style Container

    document.getElementById("container").style.cssText =

      "display:flex;justify-content: center;align-items: center;flex-wrap: wrap;gap:40px 20px;";

    // adding margin to the body

    document.body.style.margin = "40px";

  }

};

console.log(myRequest);

**my qu : What is a callback ?????**

A callback is a function passed as an arg to another function.

When you pass a function as an arg, remember no to use parenthesis

**Right:** myCalculator(5, 5, myDisplayer);

**~~Wrong:~~** ~~myCalculator(5, 5, myDisplayer());~~

const myNumbers = [4, 1, -20, -7, 5, 9, -6];

const posNumbers = removingNeg(myNumbers, (x) => x >= 0);

console.log(posNumbers); // (4) [4, 1, 5, 9]

function removingNeg(numbers, callBack) {

  const myArray = [];

  for (let x of numbers) {

    if (callBack(x)) {

      myArray.push(x);

    }

  }

  return myArray;

}

**Callback Hell or Pyramid of Doom : هرم الموت**

Imagine that you have 4 tiltles, and you are required to show each title alone so that each one appears after 1 sec of the previous, and in the same time the waiting have not to influence the other proccess;

<h3 id="title1" style="visibility: hidden">Title 1</h3>

    <h3 id="title2" style="visibility: hidden">Title 2</h3>

    <h3 id="title3" style="visibility: hidden">Title 3</h3>

    <h3 id="title4" style="visibility: hidden">Title 4</h3>

In js :

setTimeout(function () {

  document.getElementById("title1").style.visibility = "visible";

  setTimeout(function () {

    document.getElementById("title2").style.visibility = "visible";

    setTimeout(function () {

      document.getElementById("title3").style.visibility = "visible";

      setTimeout(function () {

        document.getElementById("title4").style.visibility = "visible";

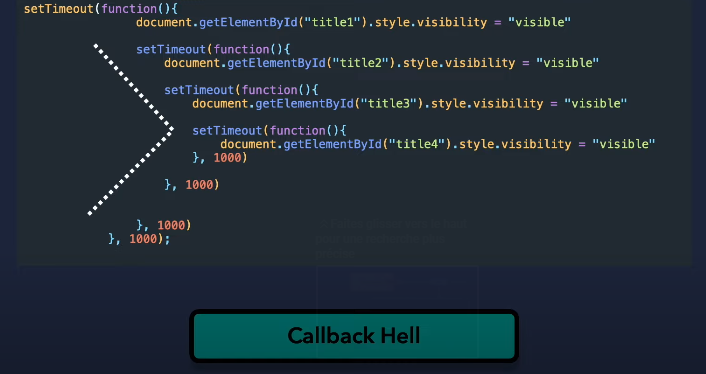
      }, 1000);

    }, 1000);

  }, 1000);

}, 1000);

I think you have noticed how difficult the code is and how the code is interwind, and for that imagine that you have many titles, and i order you to modify the delay of a title,

And also you can noticed the pyramid formed.

Callback Hell and the cause is the dificulty of reading and modifying .

The alternative solution is **Promise**

Promise

Imagine that a youtuber promise its followers to upload a video(promise), so youtuber has worked on the video(proccess) ,so after that the video may be not apploaded (rejected) or may be uploaded (resolved) .

Une image contenant texte, Police, capture d’écran, Graphique

Description générée automatiquement

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

  // do some operation ..

  let completed;

  completed = true;

  // completed = false;

  if (completed) {

    resolve("The video is uploaded ");

  } else {

    reject("the video is not uploaded ");

These are function that we use it above

  }

});

myPromise

  .then(function (msg) {

    console.log("Message From then : " + msg);

  })

  .catch(function (msg) {

    console.log("Message From Catch " + msg);

  });

// Message From then : The video is uploaded

In simpler terms, a promise in JavaScript is like a promise someone makes to you in real life. For example, if your friend promises to lend you their bike tomorrow, you trust that they will keep their promise and give you the bike. Similarly, in JavaScript, a promise is an object that represents a value that may not be available yet, but will be available in the future. When you create a promise in JavaScript, you can attach functions to it that will be executed when the promise is fulfilled or rejected. This is useful for handling asynchronous operations, like fetching data from a server or reading a file from disk, where you don't know exactly when the operation will complete. Once the promise is fulfilled, you can use the value that it represents in your code. If the promise is rejected, you can handle the error and take appropriate action. Overall, promises are a way to handle asynchronous operations in a more organized and efficient way in JavaScript.

**Operation status :**

A promise in js can have one of three possible states :

**1- Pending :** this is the initial state of a promise, which means that it is neither fulfilled nor rejected yet.

**2- Fulfilled :** This means that the promise has been successfully resolved and a value is available.

**3-Rejected :** This means that the promise has failed and an error has occurred.

When you create a promise, you can attach functions to it that will be excuted when the promise is fulfilled or rejected. This allows you to handle the different states of the promise in you code.

**The Story of Promise :**

* One A promise has been called, it will start in a Pending State
* The created promise will eventuallly end in a resolved state or in a rejected state.
* Calling the callback functions (passed to Then and Catch) upon Finishing

**Then, Catch, and finally :**

Then => Promise is successfull Use the Resolved Data

Catch => Promise is Failed, Catch the Error

Finally => Promise successful or Failed(no matter), but do something

**Note :**

You can define reject function in Then also .

----------------------------------------------------------------

We use then to make like steps to follow

We use Error for log the error and line where it is

const myPromise = new Promise((resolveFunction, rejectFunction) => {

  let employees = ["Osama", "Ahmed"];

  if (employees.length === 4) {

    resolveFunction(employees);

  } else {

    rejectFunction(Error("The number of Employees isn't four"));

  }

});

myPromise

  .then((resResult) => {

    resResult.length = 2;

    console.log(resResult);

    return resResult;

  })

  .then((theLast2emplloyes) => {

    theLast2emplloyes.length = 1;

    return theLast2emplloyes;

  })

  .then((theChoosen) => console.log(theChoosen))

  .catch((x) => console.log(x))

  .finally(console.log("The operation is done !"));

**The Promise and XHR : XMLHttpRequest**

const getData = (apiLink) => {

  return new Promise((resolve, reject) => {

    let myRequet = new XMLHttpRequest();

    myRequet.onload = function () {

      if (this.readyState === 4 && this.status === 200) {

        resolve(JSON.parse(this.responseText));

      } else Error("No Data Found");

    };

    myRequet.open("GET", apiLink);

    myRequet.send();

  })

    .then((data) => {

      data.length = 10;

      console.log(data);

      return data;

    })

    .then((data) => {

      console.log(data[0]);

      return data[0];

    })

    .then((data) => {

      console.log(data.name);

      return data.name;

    })

    .catch((rej) => console.log(rej))

    .finally(console.log("The mission is done"));

};

console.log(getData("https://api.github.com/users/bilal-friend/repos"));

Fetch API

fetch("https://api.github.com/users/bilal-friend/repos")

  .then((result) => {

    console.log(result);

    /\*Response {type: 'cors', url: 'https://api.github.com/users/bilal-friend/repos', redirected: false, status: 200, ok: true, …}\*/

    let myData = result.json();

    console.log(myData); /\* Promise {<pending>}

    [[Prototype]]: Promise

    [[PromiseState]]: "fulfilled"

    [[PromiseResult]]: Array(19)

\*/

    return myData;

  })

  .then((data) => {

    let firstRepoName = data[0].name;

    console.log(firstRepoName); // Background-Video

  });

The **response.json()** method is used to exctract the JSON body content from an http response,and it returns a promise

The **fetch()** method starts the process of fetching a resource from a server .

The **fetch()** method returns a Promise that resolves to a Response object.

😀 No need for XMLHttpRequest anymore.

Promise All and All Settled And Race

**Promise.all()**

The **promise.all()** is a method in js that simplifies working with multiple promises. It takes an array of promises as input and returns a single promise that resolves when all promises in the input array has resolved. The returned promise resolves with array of the resolved values from each of the input promise, in the order in which they were passed to **promise.all**.

Promise.all allows you to wait for all of them to finish and then the process their result together.

If a promise is rejected **promise.all** stops the process and get the rejected value only of the first rejected promise that **promise.all** meet it on the road

const myFirstPromise = new Promise((res, rej) => {

  setTimeout(() => {

    res("I am the first Promise");

  }, 5000);

});

const mySecondPromise = new Promise((res, rej) => {

  setTimeout(() => {

    res("I am the Second Promise");

  }, 1000);

});

const myThirdPromise = new Promise((res, rej) => {

  setTimeout(() => {

    res("I am the Third Promise");

  }, 2000);

});

Promise.all([myFirstPromise, mySecondPromise, myThirdPromise]).then(

  (resolvedValue) => console.log(resolvedValue),

  (rejctedValue) => console.log("rejected: " + rejctedValue)

);

// ['I am the first Promise', 'I am the Second Promise', 'I am the Third Promise']

But let rej one of them.

const myFirstPromise = new Promise((res, rej) => {

  setTimeout(() => {

    res("I am the first Promise");

  }, 5000);

});

const mySecondPromise = new Promise((res, rej) => {

  setTimeout(() => {

    res("I am the Second Promise");

  }, 1000);

});

const myThirdPromise = new Promise((res, rej) => {

  setTimeout(() => {

    rej("I am the Third Promise");

  }, 2000);

});

Promise.all([myFirstPromise, mySecondPromise, myThirdPromise]).then(

  (resolvedValue) => console.log(resolvedValue),

  (rejctedValue) => console.log("rejected: " + rejctedValue)

);

// rejected: I am the Third Promise

As you see all the process has stoped and it log all rejValue

**Promise.allSettled()**

**Promise.allSettled()** is a method in js that is similar to Promise.all(), but instead of rejecting the returned promise any of the input promise reject, if wait for all promises to either resolve or reject, and then returns an array of objects that describe the outccome of each promise.

const myFirstPromise = new Promise((res, rej) => {

  setTimeout(() => {

    res("I am the first Promise");

  }, 5000);

});

const mySecondPromise = new Promise((res, rej) => {

  setTimeout(() => {

    res("I am the Second Promise");

  }, 1000);

});

const myThirdPromise = new Promise((res, rej) => {

  setTimeout(() => {

    rej("I am the Third Promise");

  }, 2000);

});

Promise.allSettled([myFirstPromise, mySecondPromise, myThirdPromise]).then(

  (resolvedValue) => console.log(resolvedValue),

  (rejctedValue) => console.log("rejected: " + rejctedValue)

);

/\*

(3) [{…}, {…}, {…}]

0 : {status: 'fulfilled', value: 'I am the first Promise'}

1 : {status: 'fulfilled', value: 'I am the Second Promise'}

2 : {status: 'rejected', reason: 'I am the Third Promise'}

length : 3

\*/

**Promise.race()**

**Promise.race()** is a method in js that takes an array of pormises as input and returns a new promise that resolves or rejects, it depends on the first promise that finish processing. So you will not wait all the promises, the first one that finish process all result will be build on it no need to the other ones

const myFirstPromise = new Promise((res, rej) => {

  setTimeout(() => {

    res("I am the first Promise");

  }, 5000);

});

const mySecondPromise = new Promise((res, rej) => {

  setTimeout(() => {

    res("I am the Second Promise");

  }, 1000);

});

const myThirdPromise = new Promise((res, rej) => {

  setTimeout(() => {

    rej("I am the Third Promise");

  }, 2000);

});

Promise.race([myFirstPromise, mySecondPromise, myThirdPromise]).then(

  (resolvedValue) => console.log(resolvedValue),

  (rejctedValue) => console.log("rejected: " + rejctedValue)

);

/\*

I am the Second Promise

\*/

The reson tha the second promise that is logged, is that is the first one who is proccessed take a look to the delays

Async And Await

The Keyword **async** before function makes the function return a pormise.

async function asyncFunction() {

  return "Hello";

}

// the same as :

function myFunction() {

  return Promise.resolve("Hello");

}

console.log(myFunction()); // Promise {<fulfilled>: 'Hello'}

console.log(asyncFunction()); // Promise {<fulfilled>: 'Hello'}

The **await** Keyword can only be used inside an **async** function.

The **await** keyword makes the funtion pause the excution and wait for a resolved promise before it continues . meaning **Await** make js wait for the promise result

Exemple:

async function myDisplay() {

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

    resolve("I love you !!");

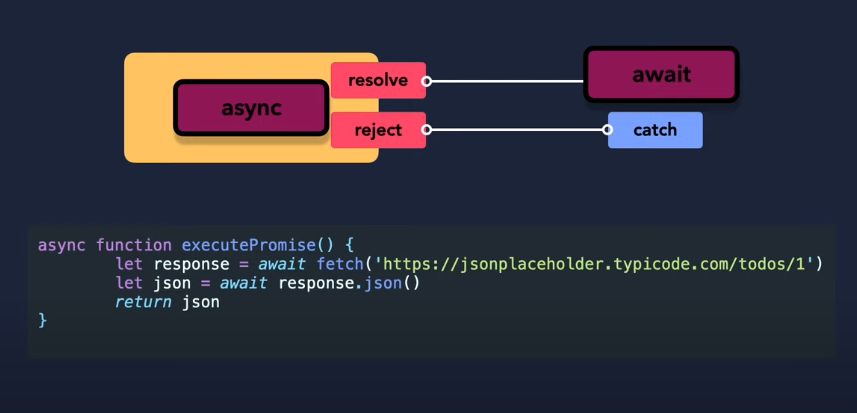
  });

  console.log(await myPromise);

}

myDisplay(); // I love you

**so, we have replaced Promise by async, and then with await**



The two arguments (resolve and reject) are pre-defined by js.

We Will not create them, but call one of them when the excutor function is ready.

const myPromise = new Promise((res, rej) => {

  setTimeout(() => {

    res("Iam a Good Promise");

  }, 2000);

});

async function readData() {

  console.log("Before Promise");

  console.log(await myPromise);

  console.log("After Promise");

}

readData();

// Before Promise

// Iam a Good Promise

// After Promise

Let remove async :

const myPromise = new Promise((res, rej) => {

  setTimeout(() => {

    res("Iam a Good Promise");

  }, 2000);

});

function readData() {

  console.log("Before Promise");

  //  console.log(myPromise);

  myPromise.then((res) => console.log(res));

  console.log("After Promise");

}

readData();

// Before Promise

// After Promise

// Iam a Good Promise

**Problem and callback and promise and async/await**

const students = [

  { fullName: "John Doe", age: 23 },

  { fullName: "Michael Smith", age: 25 },

];

function addStudent(student) {

  setTimeout(() => {

    students.push(student);

  }, 1000);

}

function getStudents() {

  setTimeout(() => {

    console.log(students);

  }, 500);

}

//

addStudent({ fullName: "Bilal Elemrani", age: 17 });

getStudents();

/\*

0: {fullName: 'John Doe', age: 23}

1: {fullName: 'Michael Smith', age: 25}

\*/

I think you noiteced the problem the sutdent is not added to the array

const students = [

  { fullName: "John Doe", age: 23 },

  { fullName: "Michael Smith", age: 25 },

];

function addStudent(student, callback) {

  setTimeout(() => {

    students.push(student);

    callback();

  }, 2000);

}

function getStudents() {

  setTimeout(() => {

    console.log(students);

  }, 500);

}

//

addStudent({ fullName: "Bilal Elemrani", age: 17 }, getStudents);

/\*

0: {fullName: 'John Doe', age: 23}

1: {fullName: 'Michael Smith', age: 25}

2: {fullName: 'Bilal Elemrani', age: 17}

\*/

The problem is solved but don't forget call backhell so :

const students = [

  { fullName: "John Doe", age: 23 },

  { fullName: "Michael Smith", age: 25 },

];

function addStudent(student) {

  return new Promise((res, rej) => {

    setTimeout(() => students.push(student), 1000);

    res();

**So Here we create like an order, first create then log in the console**

  });

}

addStudent({

  fullName: "Bilal Elemrani",

  age: 17,

}).then(() => {

  console.log(students);

});

/\*

0: {fullName: 'John Doe', age: 23}

1: {fullName: 'Michael Smith', age: 25}

2: {fullName: 'Bilal Elemrani', age: 17}

\*/

But also promise can drop us in many "then" in case of many steps

const students = [

  { fullName: "John Doe", age: 23 },

  { fullName: "Michael Smith", age: 25 },

];

function addStudent(student) {

  return new Promise((res, rej) => {

    setTimeout(() => {

      students.push(student);

      res();

    }, 1000);

  });

}

function getStudents() {

  setTimeout(() => {

    console.log(students);

  }, 500);

}

async function main() {

  await addStudent({ fullName: "Bilal Elemrani", age: 17 });

  getStudents();

}

main();

Try, Catch And Finally With Fetch

// const myPromise = new Promise((res, rej) => {

//   setTimeout(() => {

//     res("Iam The good Promise");

//   }, 3000);

// });

// async function readData() {

//   console.log("Before Promise");

//   try {

//     console.log(await myPromise);

//   } catch (reason) {

//     console.log(console.log("Reason: " + reason));

//   } finally {

//     console.log("After Promise");

//   }

// }

// readData();

async function fetchData() {

  console.log("Before Promise");

  try {

    let myData = await fetch(

      "https://mocki.io/v1/a07da115-9cfa-438a-abe7-529d6e6fe77b"

    );

    console.log(myData);

    let jsData = await myData.json();

    console.log(jsData);

  } catch (reason) {

    console.log(console.log("Reason: " + reason));

  } finally {

    console.log("After Promise");

  }

}

fetchData();

/\*

Before Promise

Response {type: 'cors', url: 'https://mocki.io/v1/a07da115-9cfa-438a-abe7-529d6e6fe77b',

redirected: false, status: 200, ok: true, …}

After Promise

or rej

Reason: TypeError: Failed to fetch

\*/

Note :

You can create some fake APIs just for testing (using in education), there is many services that do this service.

The End