JavaScript Crash Coures:

1.Variable declaration
->to declare a variable in js we have to use any of the 3 keyword that is :

i.var
ii.let
iii.const

```
let firstName = "Shivansh";
const lastName="Kumar";
var age=20;
console.log(firstName+" "+lastName+" age :"+age);
```

Console ×

```
Shivansh Kumar age :20
```

let is the most acurate way to declare a variable
as let declares a block scope variable

and var is a old way to declare a variable and
kind of deprecated way also as its function
scope that is if a variable is declared in function1
using var than that variable is limited to that function

But any variable declared outside of a function let's say in
a block not bounded by function it will be treated as global
variable which can be used in any function

a block is some bunch of code packed in {}

but the same thing is not true for let declared variable
a let declared variable is blocked scope so no matter if it
is in a function or just a simple block it will be treated as
local variable for that block but any variable outside of that
block is global

```
1   console.log('hello');
2   //global scope variable as it is not define in any block
3   let name = 'shivansh';
4   {//block 1
5
6   //this is a local variable for block 1 we cant use it in any other block or function
7     let lastName = 'kumar';
8   }
9
10  function sayHello() {
11    return 'hello' + name;
12  }
13  console.log(sayHello());
14  function sayBye() {
15    return 'Bye' + lastName;
16  }
17  console.log(sayBye());
18
```

Console ×

```
hello
helloshivansh
▸ ReferenceError: lastName is not defined
      at sayBye (<anonymous>:19:18)
      at <anonymous>:22:13
      at mn (<anonymous>:16:5455)
```

although this lastName variable is packed inside a block this will be treated as global variable unitl that block is not bounded by function cause we are using var to declare it

this age is function scoped using this age outside the function will give not defined error

script.js ×

```
1   //global scope variable as it is not define in any block
2   let name = 'shivansh';
3   {//block 1
4   // let lastName = 'kumar';
5   //this is a global variable for block 1 we can use it in any other block or function since it is declared using varr
6     var lastName="Kumar"
7   }
8
9   function sayHello() {
10    return 'hello' + name;
11  }
12  console.log(sayHello());
13
14  function sayBye() {
15    var age=20;
16    return 'Bye' + lastName;
17  }
18  console.log(sayBye());
19  console.log(age);
```

Console ×

```
helloshivansh
ByeKumar
▸ ReferenceError: age is not defined
      at <anonymous>:23:13
      at mn (<anonymous>:16:5455)
```

when we declare a variable using var inside a block that block has no significance unitl it's not bounded by any function and hence that block is redundat

```
//global scope variable as it is not define in any block
Bug Finder: Block is redundant. eslint
var lastName: string
View Problem (Alt+F8)  No quick fixes available
e can use it in any other block or function since it is declared using varr
 var lastName="Kumar"
}

function sayHello() {
  return 'hello' + name;
}
console.log(sayHello());

function sayBye() {
  var age=20;
  return 'Bye' + lastName;
}
console.log(sayBye());
console.log(age);
```

use of const:
->when a variable is declared using const it becomes immutable that means in future its value cannot change

```
const age=20;
age=18;//this will give error as age variable is constant and once declared its value cannot change
```

Console ×

Cannot assign to "age" because it is a constant

its important to assign value to constant variable at the time of declaration cause if we wont assign it than and try to assign it later it will give error:

as we can see when declared using const without a value it gave us a error of missing initializer

```
> const colorr;
const colorr;
      ^^^^^^

Uncaught SyntaxError: Missing initializer in const declaration
> let name2;
undefined
>
```

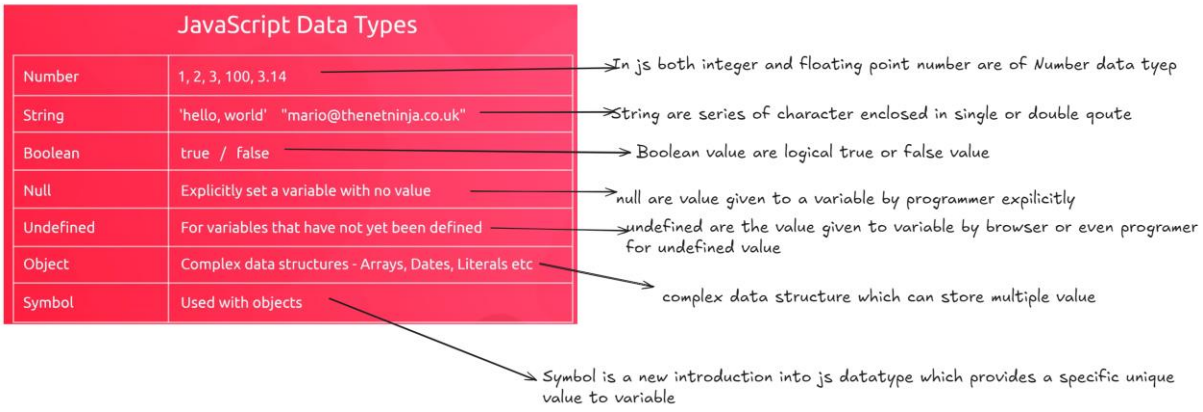
But when declared using let the variable name2 without a value it worked

that is becaue if we dont give any value to a const variable it will take null as the constant value and then we cant change it that why it produces error

Chapter 2:Printing somthing onto the console:
->to print something onto the console we us console.log(statement_to_print)

```
const color="red color";
// color="red";
console.log(color);
```

Chapter 3 :Datatypes in JavaScript:



```
let number1=400;
let number2=3.14;
console.log(number1+"->"+typeof number1);
console.log(number2+"->"+typeof number2);

let firstName="Shiv";
console.log(firstName+"->"+typeof firstName);
let logicalValue=true;
console.log(logicalValue+"->"+typeof logicalValue);

let nillValue=null;//null is a object so while printing its type it will give object but a null is a false value and using type conversion
//we  can check it
console.log(nillValue+"->"+typeof nillValue)
//type conversion of null
console.log(nillValue+" type conversion in boolean is "+Boolean(nillValue));

//Symbol
let symbol1=Symbol("Hey");
let symbol2=Symbol("Hey");
console.log(symbol1===symbol2);//it will give false cause each symbol is unique
/*
```

Console ×

```
400->number
3.14->number
Shiv->string
true->boolean
null->object
null type conversion in boolean is false
false
```

```
we can even provide it with a numbe or boolean value :
let symbol1=Symbol(1);
let symbol2=Symbol(1);r */

let arrayStorage=[];//this is how we declare array in js let/var/const array_name=[]
console.log(arrayStorage);
console.log(typeof arrayStorage);//array is a special object diffrent than conventional object
```

Console ×

```
hiv->string
rue->boolean
ull->object
ull type conversion in boolean is false
alse
[]
bject
```

Chapter4:String

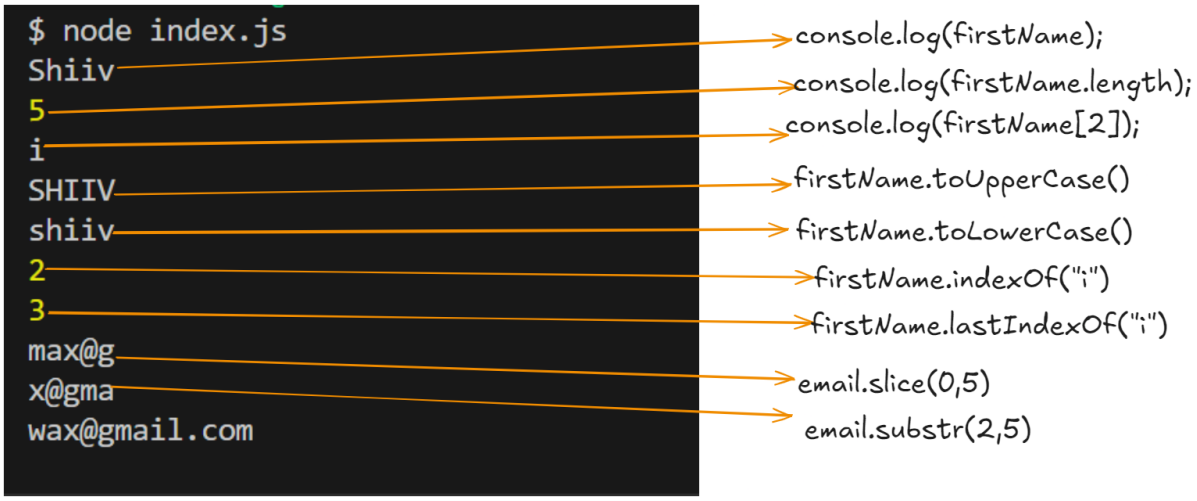
->to declare a string we enclose a series of character in double or single qoute " " or ' '

```
You, 2 hours ago | 1 author (You)
//Declaring a string to declare a string we enclose a series of character in double or single qoute " " or ' '
💡
let firstName="Shiiv";
console.log(firstName);
```

```
//String Property :
//->String is a object and a object has method and properties ,string also has one important property and that is :
//.length-->which gives back the total number of character forming the string
console.log(firstName.length);
```

```
//accesing string character so as said string is a sequance of character and each character takes up its own
//memory of 1 byte and each memory address is indexed from 0 to lenght of string-1
//firstName="Shiv"//length:4 index :0 to 3 ie
//0->S
//1->h
//2->i
//3->i
//4->v
console.log(firstName[2]);
```

```
//String methods :since string is a class/object it has property and method(function associated with object)
//just  like length property which can be accessed by the dot(.) string has methods

console.log(firstName.toUpperCase());//capatilize every character
console.log(firstName.toLowerCase());//coverts every character to lower case
console.log(firstName.indexOf('i'));//give me the first index of occurance i
console.log(firstName.lastIndexOf('i'));//gives back the index of last occurance of i

//some complicated string methods
let email="max@gmail.com";
// let sliceEmail=email.slice(start:0,end:5)//this will slice the string till 0 to end-1 ie 0 to 4
let sliceEmail=email.slice(0,5);
console.log(sliceEmail);

// let subString=email.substr(start:index,end:length/numberOfCharcterneeded);
let subString=email.substr(2,5);//start from index 2 and get me 5 character after index 2
console.log(subString);

let replaceWord=email.replace('m','w');//this will replace the very first inctance of m with w
console.log(replaceWord);
```



Chapter5:Operator

```
//There are some operator in js which can be used to perform mathematical operations like
/*
+ addition
- subtraction
* multiplication
/ division
% modulos remainder
** power
*/
let num1 = 10;
let num2 = 2;
let result1=num1/num2;
console.log(result1);
let rem=num1%3;
console.log(rem);


//Opeator Precedence:Bracket Exponential Divide Multiply Add Subtract(BEDMAS):
let pi=3.14;
let radiusMinusOne=2;
let area=pi*(radiusMinusOne+1)**2;
console.log(area);
```

```
//increment and decrement operator :
let counter1=10;
console.log(counter1++);//post increment operator meaning after reading this line counter1++ the counter will get executed
console.log(counter1);

let counter2=8;
counter2--;
console.log(counter2);


//assignment operator =,+=,-=,*=,/=.**=.%= :
let num3=10;
num3+=1;//add 1 to num3 variable and assign the new value to num3
console.log(num3);

num3-=3;
console.log(num3);
      You, 50 minutes ago • Operator added
num3*=4;
console.log(num3);
```

Chapter6:Conditions

Control Flow...

Use conditional statements for checking conditions

```
if ( some condition is true ) {
    do something...
}
```

In JavaScript, conditional statements are used to perform different actions based on different conditions. Here are the most commonly used conditional statements:

1. if Statement

The `if` statement executes a block of code if a specified condition is true.

javascript Copy code

```javascript
let age = 20;

if (age >= 18) {
    console.log("You are an adult.");
}
```

2. if...else Statement

You can use `if...else` to execute one block of code if the condition is true, and another block of code if it is false.

javascript Copy code

```javascript
let age = 16;

if (age >= 18) {
    console.log("You are an adult.");
} else {
    console.log("You are a minor.");
}
```

3. if...else if...else Statement

For multiple conditions, you can chain `if`, `else if`, and `else`.

javascript Copy code

```javascript
let score = 75;

if (score >= 90) {
    console.log("Grade: A");
} else if (score >= 80) {
    console.log("Grade: B");
} else if (score >= 70) {
    console.log("Grade: C");
} else {
    console.log("Grade: D");
}
```

4. Switch Statement

The `switch` statement is a more concise way to handle multiple conditions that involve the same expression.

javascript Copy code

```javascript
let fruit = "banana";

switch (fruit) {
    case "apple":
        console.log("You chose an apple.");
        break;
    case "banana":
        console.log("You chose a banana.");
        break;
    case "orange":
        console.log("You chose an orange.");
        break;
    default:
        console.log("Unknown fruit.");
}
```

5. Ternary Operator

The ternary operator is a shorthand for `if...else` statements. It has the form `condition ? expressionIfTrue : expressionIfFalse`.

javascript Copy code

```javascript
let age = 18;
let canVote = (age >= 18) ? "Yes, you can vote." : "No, you cannot vote.";
console.log(canVote);
```

Chapter7:Logical Gate:

i)and(&&)

T	F	T && F
T	F	F
T	T	T
F	T	F
F	F	F

ii)OR(||)

T	F	T || F
T	F	T
T	T	T
F	T	T
F	F	F

iii)Not(!)

Input	!Input
T	F
F	T

6. Logical Operators

You can also combine conditions using logical operators like `&&` (AND), `||` (OR), and `!` (NOT).

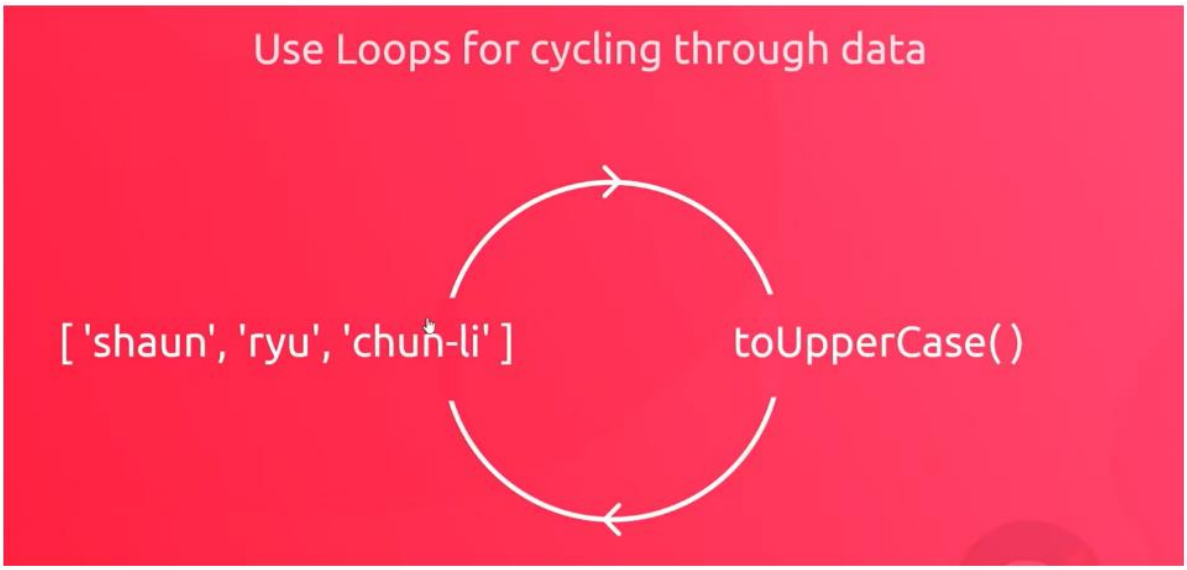
javascript Copy code

```javascript
let age = 20;
let hasPermission = true;

if (age >= 18 && hasPermission) {
    console.log("You can enter.");
} else {
    console.log("You cannot enter.");
}
```

Summary

- `if`: Executes code if the condition is true.
- `if...else`: Executes one block if true, another if false.
- `switch`: A cleaner way to handle multiple conditions based on the same variable.
- **Ternary operator**: A concise way to write simple conditional statements.
- **Logical operators**: Combine multiple conditions for more complex logic.

Chapter8:Loops



->3 types of Loop in JS i)for loop ii)while loop iii)do-while loop

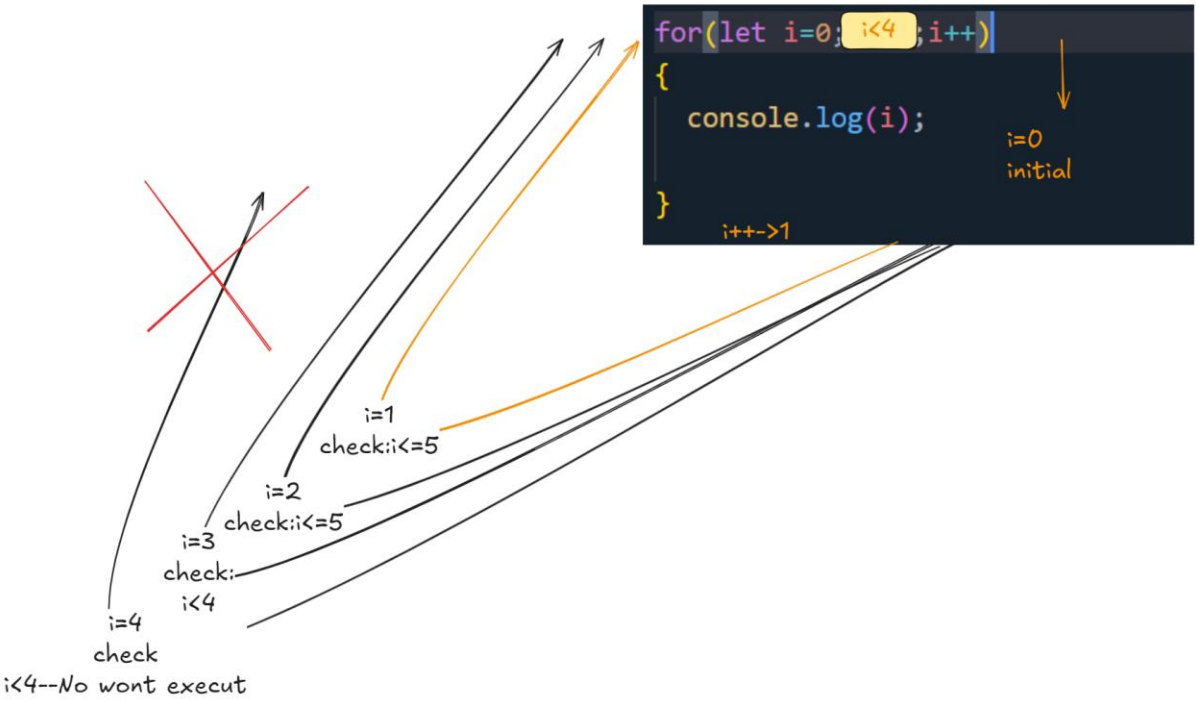
I)for loop:
Syntax of for loop:

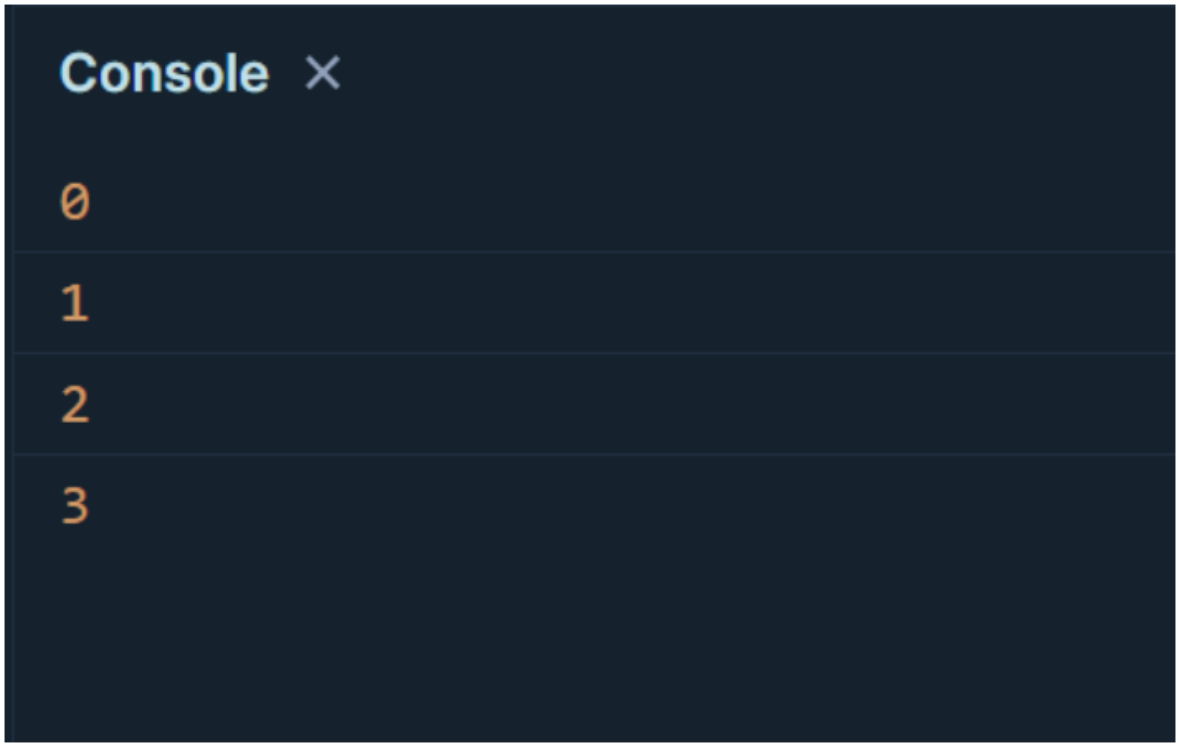
```
for(initialization;condition_toExecuteLoop;updation)
{
    //code
}
```

```
for(let i=0;i<4;i++)
{
  console.log(i);

}
```

As soon as we enter the loop the flow of execution is
first:execute let i=0;
second:execute i<4 --if true enter the loop
third:console.log(i)
fourth:after block of code is executed then:i++

Execution of the above loop will be :





ii)while loop:
->while loop is the simple version of for loop,lets recall the for loop

```
for(let i=0;i<4;i++)
{
  console.log(i);

}
```

As soon as we enter the loop the flow of execution is
first:execute let i=0;
second:execute i<4 --if true enter the loop
third:console.log(i)
fourth:after block of code is executed then:i++

so as mentioned the first line of for loop which get executed is :
1.let i=0;//initialization of iterator-of for loop
2.second line is which gets execcuted is condition :i<4 in for loop
3.console.log(i) that is the body of for loop
4.is updation of iterator :i++

->so we can write for loop in the level of execution

```
for(let i=0;i<4;i++)
{
  console.log(i);

}
```

```
let i=0;
for(/*remove the updation from loop*/;i<4;/*remove the updation from loop */)
{
  console.log(i);
  i++;

}
```

```
let i=0;
for(;i<4;)
{
  console.log(i);
  i++;

}
```

To simply this for loop JS introduced while loop

```
let i=0;
while(i<4)
{
  console.log(i);
  i++
}
```

iii)do-while loop:

The `do...while` loop executes the block of code once before checking the condition. This means that the code inside the loop will run at least once, even if the condition is false.

Syntax

javascript Copy code

```javascript
do {
    // Code to be executed
} while (condition);
```

Example

javascript Copy code

```javascript
let count = 0;

do {
    console.log("Count is: " + count);
    count++;
} while (count < 5);
```

Output

csharp Copy code

```
Count is: 0
Count is: 1
Count is: 2
Count is: 3
Count is: 4
```

iii)do-while loop VS while loop

In JavaScript, both `while` and `do...while` loops are used to execute a block of code repeatedly as long as a specified condition is true. However, they differ in how they evaluate the condition.

1. While Loop

The `while` loop checks the condition before executing the block of code. If the condition is false at the beginning, the code inside the loop will not run at all.

Syntax

javascript Copy code

```javascript
while (condition) {
    // Code to be executed
}
```

Example

javascript Copy code

```javascript
let count = 0;

while (count < 5) {
    console.log("Count is: " + count);
    count++;
}
```

2. Do...While Loop

The `do...while` loop executes the block of code once before checking the condition. This means that the code inside the loop will run at least once, even if the condition is false.

Syntax

javascript Copy code

```javascript
do {
    // Code to be executed
} while (condition);
```

Example

javascript Copy code

```javascript
let count = 0;

do {
    console.log("Count is: " + count);
    count++;
} while (count < 5);
```

Key Differences

- **Condition Check**:
  - In a `while` loop, the condition is checked before the loop's body is executed.
  - In a `do...while` loop, the loop's body is executed once before the condition is checked.
- **Execution**:
  - A `while` loop may not execute at all if the condition is false initially.
  - A `do...while` loop will always execute at least once.

Summary

- **Use a `while` loop** when you want to run a block of code as long as a condition is true, and you are not sure if it will run at all.
- **Use a `do...while` loop** when you want to ensure that the code runs at least once before checking the condition.

Chapter9:Array
Use an Array to Store a Collection of Data

```
let simpleArray = ['one', 2, 'three', true, false, undefined, null];
console.log(simpleArray.length);
```

The above is an example of the simplest implementation of an array data structure. This is known as a one-dimensional array, meaning it only has one level, or that it does not have any other arrays nested within it. Notice it contains booleans, strings, and numbers, among other valid JavaScript data types:

The console.log call displays 7.

All arrays have a length property, which as shown above, can be very easily accessed with the syntax Array.length. A more complex implementation of an array can be seen below. This is known as a multi-dimensional array, or an array that contains other arrays. Notice that this array also contains JavaScript objects, which we will examine very closely in our next section, but for now, all you need to know is that arrays are also capable of storing complex objects.

```
let complexArray = [
  [
    {
      one: 1,
      two: 2
    },
    {
      three: 3,
      four: 4
    }
  ],
  [
    {
      a: "a",
      b: "b"
    },
    {
      c: "c",
      d: "d"
    }
  ]
];
```

```
109  //to define a array we use the same technique that we use to declare variable let followed by
110  //variable name to store array:In js array is denoted using square braces
111  let yourArray=[1,"2","shivansh",true,false,Symbol("a"),3.15];
112  console.log(yourArray)
```

Console ×

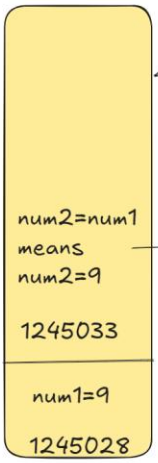
```
▼ (7) [1, "2", "shivansh", true, false, Sy...]
    0: 1
    1: "2"
    2: "shivansh"
    3: true
    4: false
    5: Symbol(a)
    6: 3.15
  ▶ [[Prototype]]: []
```

Chapter 10:Memory allocation of Array
->to understand how array works we have to take a example

```
114  let num1=9;
115  let num2=num1;
116
117  num2=10;
118  console.log(num1);
                  num1
```

```
Console ×
9
```

Here when we say num1=9 and
num2=num1
what we are actually doing is we
are making a copy of num1 value and storing
it at a new location in memory space where
num2 is created



Memory Stack

So here when we pass num2-num1 we are not passing num1 address
instead we are passing num1 value

if we could pass address of num1 to num2 like this:
num2=address(num1) then changing num2=20; will change num1 value
also cause we will be accessing num1 address directly but in js just like
Java we can not access primitive data type address

But js array is not a primitive data type ie a well define data type with perticular size to occupy but its a object in array,and objects are dynamic in nature that is there size can gorw and shrink based on the data type they store inside them and that's why they are stored dynamically in heap memory rather than stack memory :all the object are stored in heap memory cause heap provide dynamic memory allocation

In order for us to store a object in heap we use "new" key word to allocate that object memory in heap

But wait a minute when we declared array we didnt use new Keyword

```
let yourArray=[1,"2","shivansh",true,false,Symbol("a"),3.15];
```

Well although we can directly use initialize array
js interpreter actually makes our task easier as soon as
js interpreter encounter these brackets js interpreted does
some backgorund task and that is

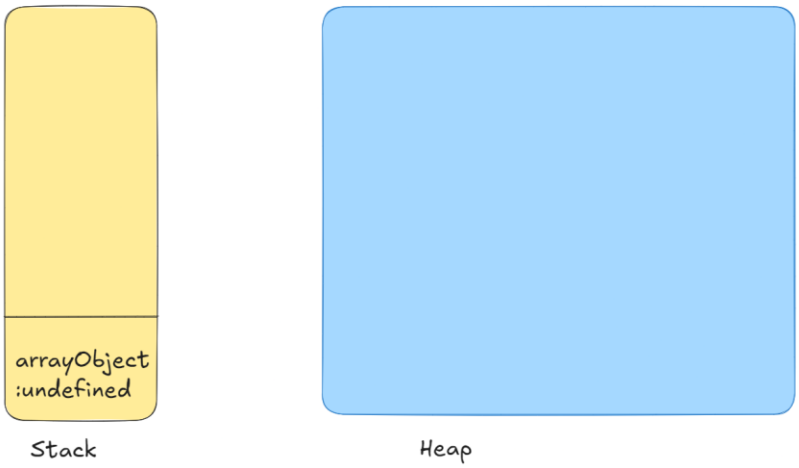
```
let yourArray=new Array(7);
yourArray[0]=1;
yourArray[1]="2";
yourArray[2]="shivansh";
yourArray[3]=true;
yourArray[4] =false;
yourArray[5]= Symbol("a");
yourArray[6]=3.14;
console.log(yourArray);
```

```
Console ×
▸ (7) [1, "2", "shivansh", true, false, Sy...]
```

so behind the scene js interepreter converts it into
new Array() object and allocates the memory dynamically in heap

```
let arrayObject;
arrayObject=new Array();
```

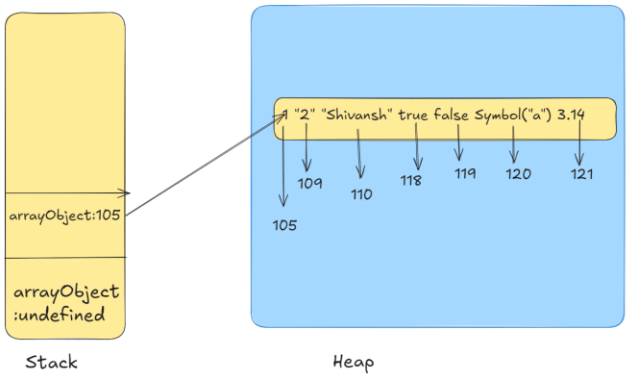
so when we declare arrayObject variable
js automatically assign it a data type of undefine




```
let arrayObject=new Array(7);
arrayObject[0]=1;
arrayObject[1]="2";
arrayObject[2]="shivansh";
arrayObject[3]=true;
arrayObject[4] =false;
arrayObject[5]= Symbol("a");
arrayObject[6]=3.14;
console.log(arrayObject);
```

So when we provide a value in form of object
new Array(7);--it allocates a continous memory
one after the another of 7 blocks

and since we cannot access heap memory directly it
gives the refrence of starting point of array to our
variable arrayObject



arrayObject[0]=1;---Number takes 4 byte:105 to 108 memory
arrayObject[1]="2";--string takes byte based on number of char 1char-1 byte
arrayObject[2]="shivansh";--takes 8 byte so 110-117
arrayObject[3]=true;--takes 1 byte 118
arrayObject[4] =false;--takes 1byte 119
arrayObject[5]= Symbol("a");--takes 1 byte 120
arrayObject[6]=3.14;--takes 4 byte 121-124

```
let arrayObject=new Array(7);
arrayObject[0]=1;
arrayObject[1]="2";
arrayObject[2]="shivansh";
arrayObject[3]=true;
arrayObject[4] =false;
arrayObject[5]= Symbol("a");
arrayObject[6]=3.14;

arrayObj2=arrayObject;
arrayObj2[2]=45;
console.log(arrayObject);
```

Console ×

▸ (7) [1, "2", 45, true, false, Symbol("a"...]

So now here as we can see that arrayObject[2]="Shivansh" carries shivansh String

Here we are making another variable arrayObj2 and assigining it the variable arrayObject now here arrayObject is actually carrying refrence to the actual array stored in heap so when we say arrayObje2--it also points to that refrence 105 in heap where our actual element/array is stored

so unlike primitive data type where when we said
let num1=9;
let num2=num1;
num2=10;
//here only the value of num2 will change and not num1 cause num2 is holding the
//value of num1 and not the address of num1

but when it comes to array when we type:
let arr1=[1,2,3];
let arr2=arr1;//arr2 holds the refrnce copy or the memorry address of array in heap
//sp changing any element of arr2 will change the element of arr1 also

```
arrayObj2=arrayObject;
arrayObj2[2]=45;
console.log(arrayObject);

```

Console ×

▸ (7) [1, "2", 45, true, false, Symbol("a"...]

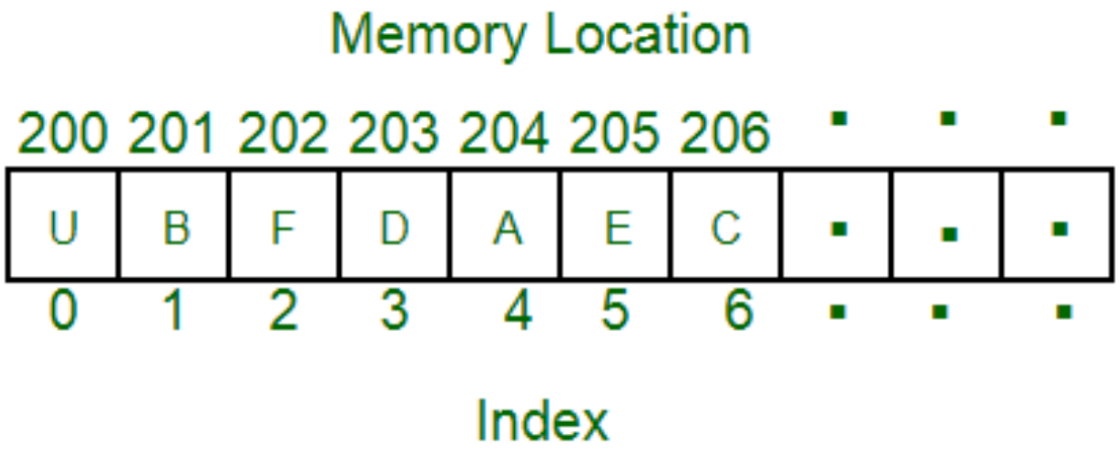
```
arrayObj2=arrayObject;
arrayObj2[2]=45;
console.log(arrayObject);

```

Console ×

▸ (7) [1, "2", 45, true, false, Symbol("a"...]

```
#include <iostream>

int main() {
    // Write C++ code here
    int arr[3]={1,2,3};
    int *ptrArray=arr;//this will store the memory address of the first
    //element in the array
    std::cout<<(*ptrArray+0);//from current memory address add 0
    std::cout<<std::endl;
    std::cout<<(*ptrArray+1);//form current memory address get the next
    std::cout<<std::endl;
    std::cout<<(*ptrArray+2);//from current memory address get me the 2nd
    //element
    std::cout<<std::endl;

    ptrArray[1]=6;
    std::cout<<(arr[0]);//from current memory address add 0
    std::cout<<std::endl;
    std::cout<<(arr[1]);//form current memory address get the next
    std::cout<<std::endl;
    std::cout<<(arr[2]);//from current memory address get me the 2nd
    //element

    return 0;
}
```

```
1
2
3
1
6
3

=== Code Execution Successful ===
```


Memory Location
200 201 202 203 204 205 206
U B F D A E C
0 1 2 3 4 5 6
Index

Access an Array's Contents Using Bracket Notation

The fundamental feature of any data structure is, of course, the ability to not only store data, but to be able to retrieve that data on command. So, now that we've learned how to create an array, let's begin to think about how we can access that array's information.

When we define a simple array as seen below, there are 3 items in it:

```
let ourArray = ["a", "b", "c"];
```

In an array, each array item has an *index*. This index doubles as the position of that item in the array, and how you reference it. However, it is important to note, that JavaScript arrays are *zero-indexed*, meaning that the first element of an array is actually at the ***zeroth*** position, not the first. In order to retrieve an element from an array we can enclose an index in brackets and append it to the end of an array, or more commonly, to a variable which references an array object. This is known as *bracket notation*. For example, if we want to retrieve the `a` from `ourArray` and assign it to a variable, we can do so with the following code:

```
let ourVariable = ourArray[0];
```

Now `ourVariable` has the value of `a`.

Now `ourVariable` has the value of `a`.

In addition to accessing the value associated with an index, you can also *set* an index to a value using the same notation:

```
ourArray[1] = "not b anymore";
```

Using bracket notation, we have now reset the item at index 1 from the string `b`, to `not b anymore`. Now `ourArray` is `["a", "not b anymore", "c"]`.

Chapter11:push,pop,unshift and shift

Add Items to an Array with push() and unshift()

An array's length, like the data types it can contain, is not fixed. Arrays can be defined with a length of any number of elements, and elements can be added or removed over time; in other words, arrays are *mutable*. In this challenge, we will look at two methods with which we can programmatically modify an array: `Array.push()` and `Array.unshift()`.

Both methods take one or more elements as parameters and add those elements to the array the method is being called on; the `push()` method adds elements to the end of an array, and `unshift()` adds elements to the beginning. Consider the following:

Both methods take one or more elements as parameters and add those elements to the array the method is being called on; the `push()` method adds elements to the end of an array, and `unshift()` adds elements to the beginning. Consider the following:

```
let twentyThree = 'XXIII';
let romanNumerals = ['XXI', 'XXII'];

romanNumerals.unshift('XIX', 'XX');
```

`romanNumerals` would have the value `['XIX', 'XX', 'XXI', 'XXII']`.

```
romanNumerals.push(twentyThree);
```

`romanNumerals` would have the value `['XIX', 'XX', 'XXI', 'XXII', 'XXIII']`. Notice that we can also pass variables, which allows us even greater flexibility in dynamically modifying our array's data.

Remove Items from an Array with pop() and shift()

Both `push()` and `unshift()` have corresponding methods that are nearly functional opposites: `pop()` and `shift()`. As you may have guessed by now, instead of adding, `pop()` *removes* an element from the end of an array, while `shift()` removes an element from the beginning. The key difference between `pop()` and `shift()` and their cousins `push()` and `unshift()`, is that neither method takes parameters, and each only allows an array to be modified by a single element at a time.

Let's take a look:

```
let greetings = ['whats up?', 'hello', 'see ya!'];

greetings.pop();
```

`greetings` would have the value `['whats up?', 'hello']`.

`greetings` would have the value `['whats up?', 'hello']`.

```
greetings.shift();
```

`greetings` would have the value `['hello']`.

We can also return the value of the removed element with either method like this:

```
let popped = greetings.pop();
```

`greetings` would have the value `[]`, and `popped` would have the value `hello`.

Remove Items Using splice()

Ok, so we've learned how to remove elements from the beginning and end of arrays using `shift()` and `pop()`, but what if we want to remove an element from somewhere in the middle? Or remove more than one element at once? Well, that's where `splice()` comes in. `splice()` allows us to do just that: **remove any number of consecutive elements** from anywhere in an array.

`splice()` can take up to 3 parameters, but for now, we'll focus on just the first 2. The first two parameters of `splice()` are integers which represent indexes, or positions, of items in the array that `splice()` is being called upon. And remember, arrays are *zero-indexed*, so to indicate the first element of an array, we would use `0`. `splice()`'s first parameter represents the index on the array from which to begin removing elements, while the second parameter indicates the number of elements to delete. For example:

```
let array = ['today', 'was', 'not', 'so', 'great'];

array.splice(2, 2);
```

Here we remove 2 elements, beginning with the third element (at index 2). `array` would have the value `['today', 'was', 'great']`.

`splice()` not only modifies the array it's being called on, but it also returns a new array containing the value of the removed elements:

```
let array = ['I', 'am', 'feeling', 'really', 'happy'];

let newArray = array.splice(3, 2);
```

`newArray` has the value `['really', 'happy']`.

Add Items Using splice()

Remember in the last challenge we mentioned that `splice()` can take up to three parameters? Well, you can use the third parameter, comprised of one or more element(s), to add to the array. This can be incredibly useful for quickly switching out an element, or a set of elements, for another.

```
const numbers = [10, 11, 12, 12, 15];
const startIndex = 3;
const amountToDelete = 1;

numbers.splice(startIndex, amountToDelete, 13, 14);
console.log(numbers);
```

The second occurrence of `12` is removed, and we add `13` and `14` at the same index. The `numbers` array would now be `[ 10, 11, 12, 13, 14, 15 ]`.

Here, we begin with an array of numbers. Then, we pass the following to `splice()`: The index at which to begin deleting elements (3), the number of elements to be deleted (1), and the remaining arguments (13, 14) will be inserted starting at that same index. Note that there can be any number of elements (separated by commas) following `amountToDelete`, each of which gets inserted.

Copy Array Items Using slice()

The next method we will cover is `slice()`. Rather than modifying an array, `slice()` copies or *extracts* a given number of elements to a new array, leaving the array it is called upon untouched. `slice()` takes only 2 parameters — the first is the index at which to begin extraction, and the second is the index at which to stop extraction (extraction will occur up to, but not including the element at this index). Consider this:

```
let weatherConditions = ['rain', 'snow', 'sleet', 'hail', 'clear'];

let todaysWeather = weatherConditions.slice(1, 3);
```

`todaysWeather` would have the value `['snow', 'sleet']`, while `weatherConditions` would still have `['rain', 'snow', 'sleet', 'hail', 'clear']`.

In effect, we have created a new array by extracting elements from an existing array.

Copy an Array with the Spread Operator

While `slice()` allows us to be selective about what elements of an array to copy, among several other useful tasks, ES6's new *spread operator* allows us to easily copy *all* of an array's elements, in order, with a simple and highly readable syntax. The spread syntax simply looks like this: `...`

In practice, we can use the spread operator to copy an array like so:

```
let thisArray = [true, true, undefined, false, null];
let thatArray = [...thisArray];
```

`thatArray` equals `[true, true, undefined, false, null]`. `thisArray` remains unchanged and `thatArray` contains the same elements as `thisArray`.

Chapter12:Spread Operator

Copy an Array with the Spread Operator

While `slice()` allows us to be selective about what elements of an array to copy, among several other useful tasks, ES6's new *spread operator* allows us to easily copy *all* of an array's elements, in order, with a simple and highly readable syntax. The spread syntax simply looks like this: `...`

In practice, we can use the spread operator to copy an array like so:

```
let thisArray = [true, true, undefined, false, null];
let thatArray = [...thisArray];
```

`thatArray` equals `[true, true, undefined, false, null]`. `thisArray` remains unchanged and `thatArray` contains the same elements as `thisArray`.

Combine Arrays with the Spread Operator

Another huge advantage of the *spread* operator is the ability to combine arrays, or to insert all the elements of one array into another, at any index. With more traditional syntaxes, we can concatenate arrays, but this only allows us to combine arrays at the end of one, and at the start of another. Spread syntax makes the following operation extremely simple:

```
let thisArray = ['sage', 'rosemary', 'parsley', 'thyme'];

let thatArray = ['basil', 'cilantro', ...thisArray, 'coriander'];
```

`thatArray` would have the value `['basil', 'cilantro', 'sage', 'rosemary', 'parsley', 'thyme', 'coriander']`.

Using spread syntax, we have just achieved an operation that would have been more complex and more verbose had we used traditional methods.

Chapter:13-Array Traversing

Iterate Through All an Array's Items Using For Loops

Sometimes when working with arrays, it is very handy to be able to iterate through each item to find one or more elements that we might need, or to manipulate an array based on which data items meet a certain set of criteria. JavaScript offers several built in methods that each iterate over arrays in slightly different ways to achieve different results (such as `every()`, `forEach()`, `map()`, etc.), however the technique which is most flexible and offers us the greatest amount of control is a simple `for` loop.

Consider the following:

```
function greaterThanTen(arr) {
  let newArr = [];
  for (let i = 0; i < arr.length; i++) {
    if (arr[i] > 10) {
      newArr.push(arr[i]);
    }
  }
  return newArr;
}

greaterThanTen([2, 12, 8, 14, 80, 0, 1]);
```

Using a `for` loop, this function iterates through and accesses each element of the array, and subjects it to a simple test that we have created. In this way, we have easily and programmatically determined which data items are greater than `10`, and returned a new array, `[12, 14, 80]`, containing those items.

In JavaScript, there are several methods to iterate over arrays. Here are some of the most common iteration methods:

1. forEach()

The `forEach()` method executes a provided function once for each array element.

javascript Copy code

```javascript
const fruits = ['apple', 'banana', 'cherry'];

fruits.forEach((fruit) => {
    console.log(fruit);
});
```

2. for Loop

The traditional `for` loop allows for greater control over the iteration process.

javascript Copy code

```javascript
const fruits = ['apple', 'banana', 'cherry'];

for (let i = 0; i < fruits.length; i++) {
    console.log(fruits[i]);
}
```

3. for...of Loop

The `for...of` loop provides a simpler syntax for iterating over iterable objects (like arrays).

javascript Copy code

```javascript
const fruits = ['apple', 'banana', 'cherry'];

for (const fruit of fruits) {
    console.log(fruit);
}
```

4. map()

The `map()` method creates a new array with the results of calling a provided function on every element in the calling array.

javascript Copy code

```javascript
const fruits = ['apple', 'banana', 'cherry'];

const uppercasedFruits = fruits.map(fruit => fruit.toUpperCase());
console.log(uppercasedFruits); // ['APPLE', 'BANANA', 'CHERRY']
```

5. filter()

The `filter()` method creates a new array with all elements that pass the test implemented by the provided function.

javascript Copy code

```javascript
const numbers = [1, 2, 3, 4, 5];

const evenNumbers = numbers.filter(num => num % 2 === 0);
console.log(evenNumbers); // [2, 4]
```

6. reduce()

The `reduce()` method executes a reducer function on each element of the array, resulting in a single output value.

javascript Copy code

```javascript
const numbers = [1, 2, 3, 4, 5];

const sum = numbers.reduce((accumulator, current) => accumulator + current, 0);
console.log(sum); // 15
```

7. some() and every()

- **some()** checks if at least one element passes the test.
- **every()** checks if all elements pass the test.

javascript Copy code

```javascript
const numbers = [1, 2, 3, 4, 5];

const hasEven = numbers.some(num => num % 2 === 0);
console.log(hasEven); // true

const allEven = numbers.every(num => num % 2 === 0);
console.log(allEven); // false
```

Summary

- **forEach()** for simple iteration.
- **for** and **for...of** loops for more control.
- **map()**, **filter()**, and **reduce()** for functional programming approaches.
- **some()** and **every()** for condition checks.

will forEach() make changes in original array

The `forEach()` method in JavaScript does not inherently modify the original array. It simply iterates over each element and executes a provided function on it. However, if the callback function you provide to `forEach()` modifies the elements of the array directly (if they are objects, for example), then those changes will reflect in the original array.

Example 1: No Modification

If you use `forEach()` with a primitive type (like numbers or strings), the original array remains unchanged:

javascript Copy code

```javascript
const numbers = [1, 2, 3];

numbers.forEach((num) => {
    num = num * 2; // This does not affect the original array
});

console.log(numbers); // Output: [1, 2, 3]
```

Example 2: Modifying Objects

If you have an array of objects and modify the properties of those objects, the original array will be affected:

javascript Copy code

```javascript
const fruits = [{ name: 'apple' }, { name: 'banana' }];

fruits.forEach((fruit) => {
    fruit.name = fruit.name.toUpperCase(); // Modifying object property
});

console.log(fruits); // Output: [{ name: 'APPLE' }, { name: 'BANANA' }]
```

Summary

- `forEach()` **does not change the original array** unless you modify the elements directly within the callback (e.g., changing properties of objects).
- It is primarily used for executing side effects, such as logging or calling functions, without returning a new array.

```
const fruits = ['apple', 'banana', 'cherry'];
console.log(fruits[0].toUpperCase());
const uppercasedFruits = fruits.map((fruit) => {
   fruit.toUpperCase();
 why it is goving me undefined array
```

The reason you're getting `undefined` in your `uppercasedFruits` array is that the `map()` method is not returning any value from the arrow function. In JavaScript, if you don't explicitly return a value from a function, it returns `undefined` by default.

Correcting the Code

To fix this, you need to add a `return` statement in your arrow function within the `map()` method. Here's how you can do it:

Correcting the Code

To fix this, you need to add a `return` statement in your arrow function within the `map()` method. Here's how you can do it:

javascript Copy code

```javascript
const fruits = ['apple', 'banana', 'cherry'];

// This logs the first fruit in uppercase
console.log(fruits[0].toUpperCase());

// Corrected map function
const uppercasedFruits = fruits.map((fruit) => {
    return fruit.toUpperCase(); // Explicitly return the uppercase fruit
});

// Alternatively, you can use concise body syntax
// const uppercasedFruits = fruits.map(fruit => fruit.toUpperCase());

console.log(uppercasedFruits); // Output: ['APPLE', 'BANANA', 'CHERRY']
```

Summary

- **Return Value**: Ensure to use a `return` statement in the arrow function when using `map()`, or use concise syntax to return the value directly.
- **Understanding `map()`**: The `map()` method creates a new array populated with the results of calling a provided function on every element in the calling array.

filter function

The `filter()` function in JavaScript is used to create a new array containing all the elements of the original array that pass a specified test (i.e., satisfy a condition defined by a provided function).

Syntax

javascript Copy code

```
const newArray = array.filter(callback(element[, index[, array]])[, thisArg]);
```

- **callback**: A function that tests each element. It receives three arguments:
  - **element**: The current element being processed.
  - **index** (optional): The index of the current element.
  - **array** (optional): The array `filter` was called upon.
- **thisArg** (optional): Value to use as `this` when executing the callback.

Example

Here's a simple example that demonstrates how to use the `filter()` function:

javascript Copy code

```
const numbers = [1, 2, 3, 4, 5, 6];

// Filter out even numbers
const evenNumbers = numbers.filter((num) => num % 2 === 0);

console.log(evenNumbers); // Output: [2, 4, 6]
```

Another Example: Filtering Objects

You can also use `filter()` to work with arrays of objects. For instance, consider the following example where we filter an array of people based on their age:

javascript Copy code

```javascript
const people = [
    { name: 'Alice', age: 25 },
    { name: 'Bob', age: 30 },
    { name: 'Charlie', age: 18 },
    { name: 'Diana', age: 22 }
];

// Filter people who are 21 or older
const adults = people.filter(person => person.age >= 21);

console.log(adults);
// Output: [{ name: 'Alice', age: 25 }, { name: 'Bob', age: 30 }, { name: 'Diana', age: 22
```

Chapter14:functions

A function in JavaScript is a reusable block of code that performs a specific task. You define it once, and then you can run (or "call") it whenever you need that task done in your program.

> *A JavaScript function runs when it is "called" by some part of your code.*

Syntax: The basic syntax to create a function in JavaScript is shown below.

```
function functionName(Parameter1, Parameter2, ...)
{
    // Function body
}
```

To create a function in JavaScript, we have to first use the **keyword *function***, separated by the name of the function and parameters within parenthesis. The part of the function inside the curly braces **{}** is the body of the function.

Why Functions?

- Functions can be used multiple times, reducing redundancy.
- Break down complex problems into manageable pieces.
- Manage complexity by hiding implementation details.
- Can call themselves to solve problems recursively.

Function Invocation

The function code you have written will be executed whenever it is called.

- Triggered by an event (e.g., a button click by a user).
- When explicitly called from JavaScript code.
- Automatically executed, such as in self-invoking functions.

Function Definition

Before, using a user-defined function in JavaScript we have to create one. We can use the above syntax to create a function in JavaScript.

A function definition is sometimes also termed a function declaration or function statement. Below are the rules for creating a function in JavaScript:

- Every function should begin with the keyword *function* followed by,
- A user-defined function name that should be unique,
- A list of parameters enclosed within parentheses and separated by commas,
- A list of statements composing the body of the function enclosed within curly braces {}.

Example: This example shows a basic declaration of a function in javascript.

JavaScript

```
function calcAddition(number1, number2) {
    return number1 + number2;
}
console.log(calcAddition(6,9));
```

Output

```
15
```

In javascript, functions can be used in the same way as variables for assignments, or calculations.

In JavaScript, there are several types of functions, each with its own characteristics and use cases. Here's an overview of the main types:

1. Function Declarations

A function declaration defines a named function that can be called later in the code.

javascript Copy code

```javascript
function greet() {
    console.log("Hello, World!");
}

greet(); // Output: Hello, World!
```

2. Function Expressions

A function expression defines a function and can be assigned to a variable. This can be anonymous or named.

Anonymous Function Expression:

javascript Copy code

```javascript
const greet = function() {
    console.log("Hello, World!");
};

greet(); // Output: Hello, World!
```

Named Function Expression:

javascript Copy code

```javascript
const greet = function greetFunction() {
    console.log("Hello, World!");
};

greet(); // Output: Hello, World!
```

3. Arrow Functions

Arrow functions provide a more concise syntax for writing functions and lexically bind the `this` value.

javascript Copy code

```javascript
const greet = () => {
    console.log("Hello, World!");
};

greet(); // Output: Hello, World!
```

4. IIFE (Immediately Invoked Function Expressions)

IIFEs are functions that are executed immediately after they are defined.

javascript Copy code

```javascript
(function() {
    console.log("This runs immediately!");
})(); // Output: This runs immediately!
```

5. Higher-Order Functions

Higher-order functions are functions that take other functions as arguments or return functions as their result.

javascript Copy code

```javascript
function higherOrderFunction(callback) {
    callback();
}

higherOrderFunction(() => console.log("Called from a higher-order function!")); // Output:
```

6. Callback Functions

A callback function is a function passed into another function as an argument, which can then be called within that function.

javascript Copy code

```javascript
function processUserInput(callback) {
    const name = "Alice";
    callback(name);
}

processUserInput((name) => {
    console.log(`Hello, ${name}!`);
}); // Output: Hello, Alice!
```

7. Generator Functions

Generator functions allow you to define an iterative algorithm by using the `function*` syntax and `yield` keyword.

javascript Copy code

```javascript
function* numberGenerator() {
    yield 1;
    yield 2;
    yield 3;
}

const gen = numberGenerator();
console.log(gen.next().value); // Output: 1
console.log(gen.next().value); // Output: 2
console.log(gen.next().value); // Output: 3
```

8. Async Functions

Async functions return a promise and allow you to write asynchronous code that looks synchronous.

javascript Copy code

```javascript
async function fetchData() {
    return "Data fetched!";
}

fetchData().then(data => console.log(data)); // Output: Data fetched!
```

Chapter15:HighOrder function and Callback

High Order function and call backs both are inter related :
1.Higher-order functions are functions that take other functions as arguments or return functions as their result.
2.A callback function is a function passed into another function as an argument, which can then be called within that function.

//High-order works as inputTaker and callback is input

```
function inputTaker(p1,p2,func)
{
    return func(p1,p2);
}
function add(n1,n2)
{
    return n1+n2;
}
```

```
let result=inputTaker(5,6,add);
```

Here remeber when ever we pass a named function we pass its refrence instead of actual function call that is name+brackets()

Why ???

```
let result2=inputTaker(5,6,add(5,6));//this will give error cause when called inputTaker--will call this
//func(p1,p2): add(5,6)(5,6) where func holds add(5,6)

//thats why we pass refrence to the function via name inputTaker(5,6,add) so here func(p1,p2):func-->add p1:5 and p2:6


console.log(result2);}
```

Caue if we call the highorder function with actuall function call in parameter that function will get executed and when the function will be called inside high order it wont be following the correct syntax of funnction

```
function inputTaker(p1,p2,func)
{
    return func(p1,p2);
}
function add(n1,n2)
{
    return n1+n2;
}
```

```
let result2=inputTaker(5,6,add(5,6));
```

this will immediately call this function

and when this will call the function behind the scene this is what it will look like

```
function inputTaker(p1=5,p2=6,func=add(5,6))
{
    return func=add(5,6)(p1=5,p2=6);
}
```

and this is not the correct syntax of function call :
add(5,6)(5,6) this will give error that why while giving a name function as parameter to a high order function as call back always give refrence and not the actual function call

```
/*
few example of high order function which is pre built into js interpreter or runtime that are:
1.map()
2.filter()
3.reduce()

each takes a function as input and execute it on array element
*/

let arr=[1,2,3];
let resultArray=arr.map((Element)=>{Element=Element*2;return Element+2;});
console.log(resultArray);
```

```
arr.map((Element)=>{Element=Element*2;return Element+2;});
```

here u may say hmm we just discussed above that when we pass a call back we should invoke it directly but here

```
let resultArray=arr.map((Element)=>{
    Element=Element*2;return Element+2;});
```

I mean forget about invoking it immediately we have defined it inside a high order function

will it not get executed immdeately ans is no cause we have defined a anonymous arrow function

inside our high order function map() but we haven't call it in order to call a function suppose my function name is

sayHello we need to add this open and closing parenthesis () untill we don't do that it wont get invoke

Similarly when we play with a anonymous function unitl we dont attach these () using iife(Immediately Invoked Function Expressions) they wont get execute insted there function defination will go into map() function define in array class

Hmm a bit confusing isn't it lets try to understand it by taking a peak inside how map() method in array class is written

```
function mapArray2Func(arrayPara,func)
{
    let createNew=[];
    for(let i=0;i<arrayPara.length;i++)
    {
        let valueCatcher=arrayPara[i];
        createNew.push(func(valueCatcher));//
    }
    console.log(createNew);
}

mapArray2Func([1,2,3],(elem)=>{return elem*2});
```

suppose this a pseudo or dummy code for our map function which take a array as input parameter and 2nd input parameter is function

```
mapArray2Func(
    [1,2,3]
    ,
    (elem)=>{return elem*2}
);
```

```
function mapArray2Func(
    arrayPara
    ,        You, 19 minutes ago • Uncommitted chang
    func)
{
    let createNew=[];
    for(let i=0;i<arrayPara.length;i++)
    {
        let valueCatcher=arrayPara[i];
        createNew.push(func(valueCatcher));//
    }
    console.log(createNew);
}
```

When we call this argument will map them selfes to the parameter that is

```
function mapArray2Func(
    arrayPara=[1,2,3]
    ,
    func=(elem)=>{return elem*2}

    )
{
    let createNew=[];
    for(let i=0;i<arrayPara.length;i++)
    {
        let valueCatcher=arrayPara[i];
        createNew.push(/*func:*/((elem)=>{return elem*2})(valueCatcher));//
    }
    console.log(createNew);
}
```

so as we can see that anonymous function or arrow function can not be evoked automatically so when we define them in high order function we are defining the logic of that anonymous function but that wont get evoked instead its refrence will get maped with func parameter of mapArray2Func() and then using IIFE it's called

```
// function mapArray2Func(arrayPara,func)//func:(elem)=>{return elem*2}
// {
//     let createNew=[];
//     for(let i=0;i<arrayPara.length;i++)
//     {
//         let valueCatcher=arrayPara[i];
//         createNew.push(func(valueCatcher));//func:(elem)=>{return elem*2} elem:valueCatcher

          //createNew.push(((elem)=>{return elem*2})(valueCather));
//     }
//     console.log(createNew);
// }
```

```
function mapArray2FuncDemo(arrayPara)
{
    let createNew=[];
    for(let i=0;i<arrayPara.length;i++)
    {
        let valueCatcher=arrayPara[i];
        // createNew.push(((elem)=>{console.log(elem*2)})(valueCatcher));
        createNew.push(((elem)=>{return (elem*2)})(valueCatcher));
    }
    console.log(createNew);
}

mapArray2FuncDemo([1,2,3]);
```

Chapter 16 :Single threaded Nature of JS:

First of all before getting into the discussion about what is single threaded and multi-thereded programming language are :

let's see what is a thread
In programming, a thread is a sequence of instructions that a computer's CPU and the computer itself must execute. It's the smallest unit of a process that a scheduler can manage independently

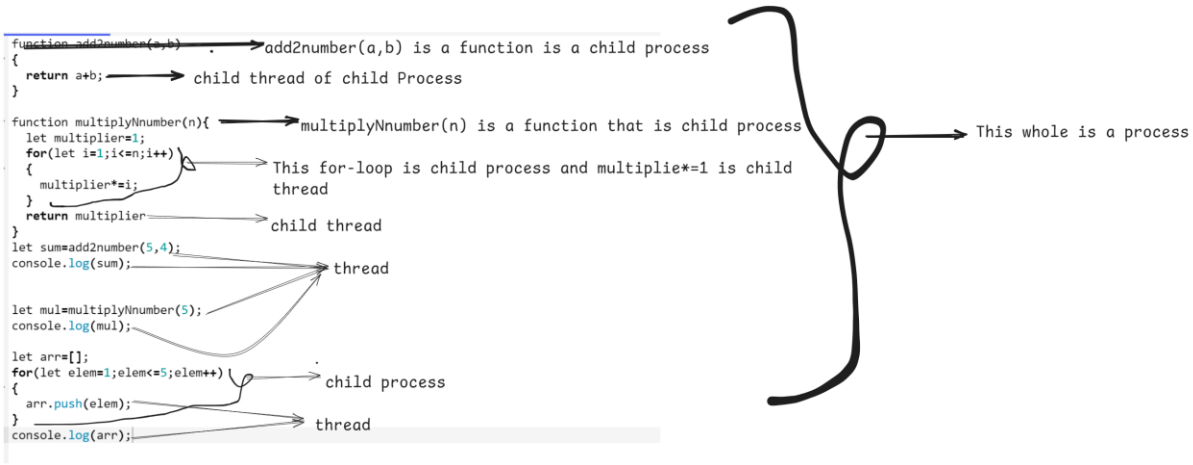
What is Process?

Processes are basically the programs that are dispatched from the ready state and are scheduled in the CPU for execution. PCB (Process Control Block) holds the context of process. A process can create other processes which are known as Child Processes. The process takes more time to terminate, and it is isolated means it does not share the memory with any other process. The process can have the following states new, ready, running, waiting, terminated, and suspended.

What is Thread?

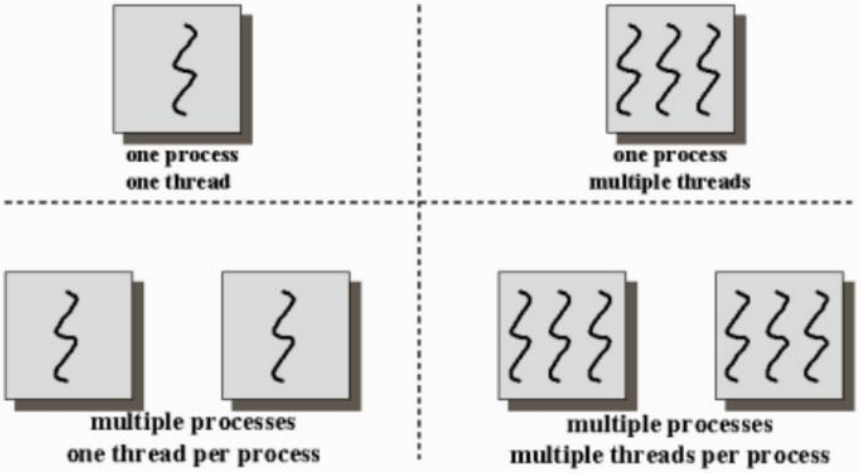
Thread is the segment of a process which means a process can have multiple threads and these multiple threads are contained within a process. A thread has three states: Running, Ready, and Blocked.

The thread takes less time to terminate as compared to the process but unlike the process, threads do not isolate.



Process	Thread
Process means any program is in execution.	Thread means a segment of a process.
The process takes more time to terminate.	The thread takes less time to terminate.
It takes more time for creation.	It takes less time for creation.
It also takes more time for context switching.	It takes less time for context switching.
The process is less efficient in terms of communication.	Thread is more efficient in terms of communication.
Multiprogramming holds the concepts of multi-process.	We don't need multi programs in action for multiple threads because a single process consists of multiple threads.
The process is isolated.	Threads share memory.
The process is called the heavyweight process.	A Thread is lightweight as each thread in a process shares code, data, and resources.
Process switching uses an interface in an operating system.	Thread switching does not require calling an operating system and causes an interrupt to the kernel.
If one process is blocked, then it will not affect the execution of other processes.	If a user-level thread is blocked, then all other user-level threads are blocked.
The process has its own Process Control Block, Stack, and Address Space.	Thread has Parents' PCB, its own Thread Control Block, and Stack and common Address space.
Changes to the parent process do not affect child processes.	Since all threads of the same process share address space and other resources so any changes to the main thread may affect the behavior of the other threads of the process.
A system call is involved in it.	No system call is involved, it is created using APIs.
The process does not share data with each other.	Threads share data with each other.

Threads and Processes



Single Threaded Programming language:

Single-threaded programming

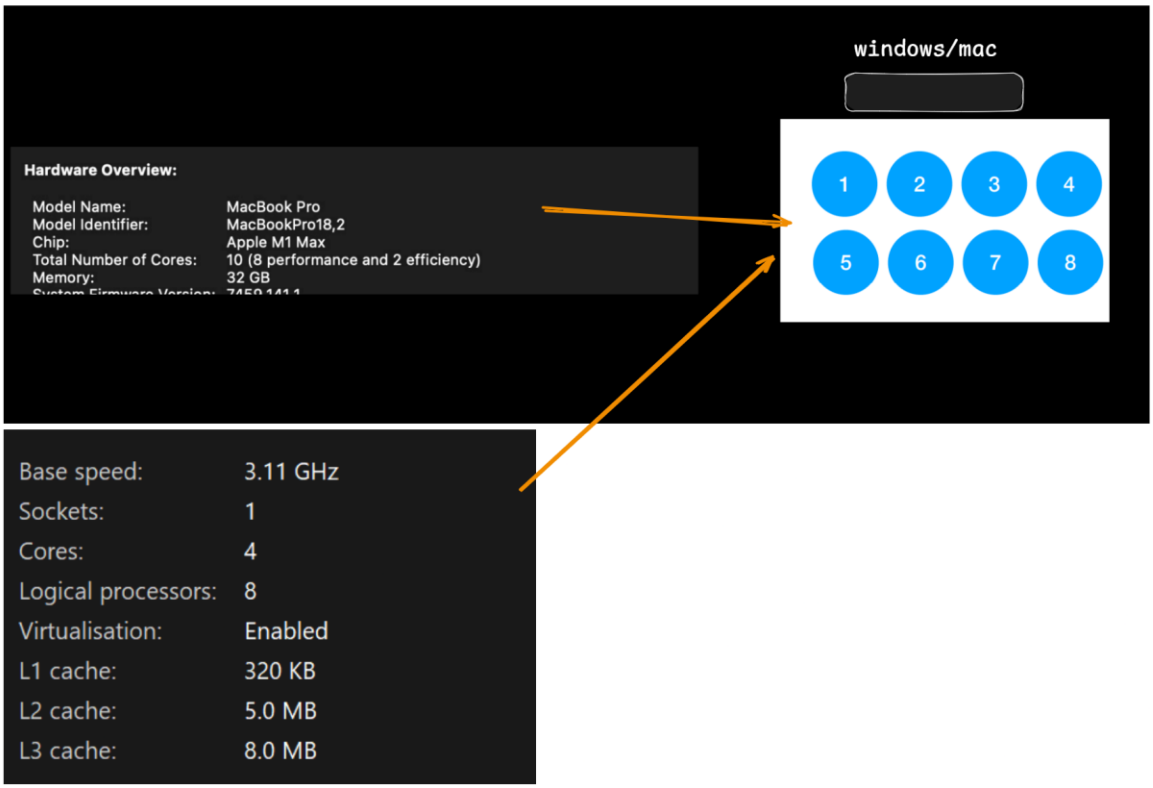
In single-threaded programs, tasks are executed in sequence, and each task must finish before the next one can start. Single-threaded programs are well-suited for simple applications or those with low computational complexity. They are also easy to debug and maintain.

Multi-threaded programming

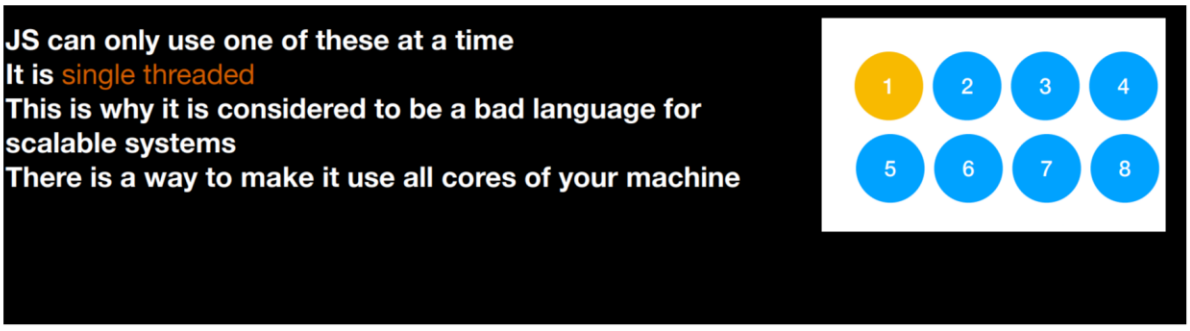
In multi-threaded programs, multiple threads are used to handle tasks concurrently. This can improve performance and responsiveness. Multi-threaded programs are better suited for CPU-intensive tasks. However, the overhead of creating and managing threads can reduce efficiency when dealing with a large number of concurrent connections.

Single threaded Nature of js:

To understand why JavaScript is single-threaded, we first need to define what it means for a language to be single-threaded. A single-threaded language is one that can execute only one task at a time. The program will execute the tasks in sequence, and each task must complete before the next task starts.



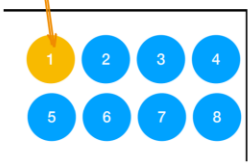
Base speed:	3.11 GHz
Sockets:	1
Cores:	4
Logical processors:	8
Virtualisation:	Enabled
L1 cache:	320 KB
L2 cache:	5.0 MB
L3 cache:	8.0 MB



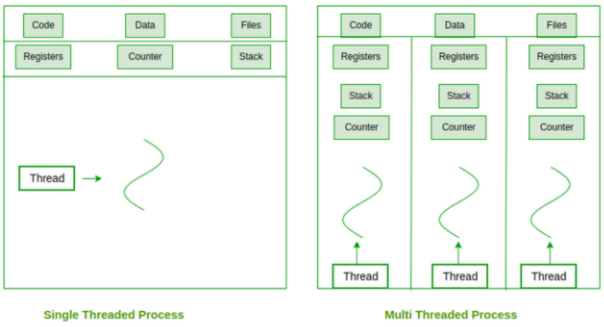
```
let sum=0;
let mul=0
for(let i=0;i<=100000000000000;i++)
{
    sum+=i;
    mul*=i;
}
console.log("Hello World");
// console.log(sum);
// console.log(mul);
```

This here is a single child process with multiple thread

as we can see that this is a fairly large calculation and since
js is single threaded until it compelets its execution js interpreter wont
be able to execute a fairly simple console.log("Hello World Program);
cause the whole calculation will be hogging a 1 entire core

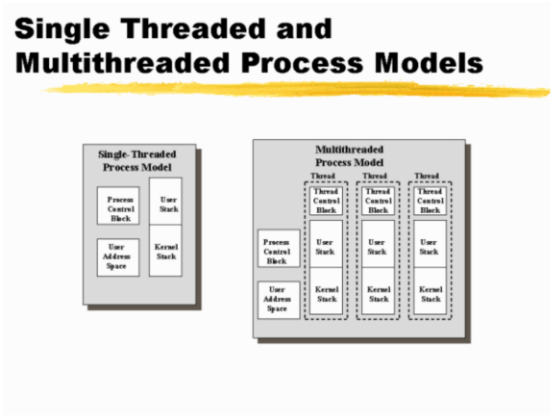


If we were dealing with a multi-programming language like java or python our memory stack would have been divide into multiple thread stack isolating each thread stack from one another



Single Threaded Process

Multi Threaded Process



In multi-threading programming language each thread can be executed on diffrent core making each task much more faster to execute and much more efficient

lets understand with example:

Multithreading in Java

Java has built-in support for multithreading, allowing multiple threads to run concurrently. Here's a simple example:

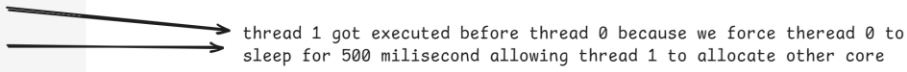
java Copy code

```java
// Simple Multithreading in Java

class SimpleThread extends Thread {
    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println(Thread.currentThread().getName() + " - Count: " + i);
            try {
                Thread.sleep(500); // Sleep for 500 milliseconds
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

public class MultiThreadExample {
    public static void main(String[] args) {
        SimpleThread thread1 = new SimpleThread();
        SimpleThread thread2 = new SimpleThread();

        thread1.start(); // Start first thread
        thread2.start(); // Start second thread
    }
}
```

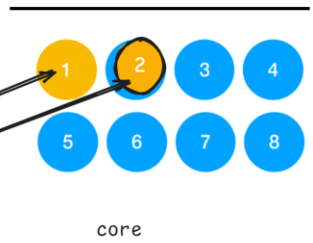
```java
class SimpleThread extends Thread {
    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println(Thread.currentThread().getName() + " -
                Count: " + i);
            try {
                //we are forcing our thread/perticular task to sleep
                //in order for other thread to start on other core
                Thread.sleep(500); // Sleep for 500 milliseconds
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

public class Main {
    public static void main(String[] args) {
        SimpleThread thread1 = new SimpleThread();
        SimpleThread thread3 = new SimpleThread();

        thread1.start(); // Start first thread
        thread3.start(); // Start second thread
    }
}
```

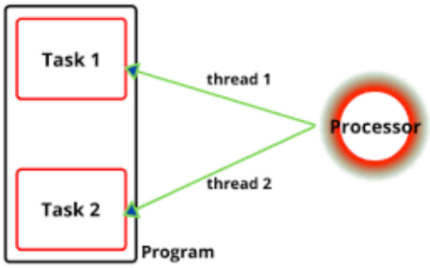
Thread Naming: By default, Java names threads as "Thread-0", "Thread-1", etc.

```
Thread-1 - Count: 0
Thread-0 - Count: 0
Thread-1 - Count: 1
Thread-0 - Count: 1
Thread-1 - Count: 2
Thread-0 - Count: 2
Thread-1 - Count: 3
Thread-0 - Count: 3
Thread-1 - Count: 4
Thread-0 - Count: 4
```



```java
class SimpleThread extends Thread {
    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println(Thread.currentThread().getName() + " -
                Count: " + i);
            try {
                //we are forcing our thread/perticular task to sleep
                //in order for other thread to start on other core
                Thread.sleep(500); // Sleep for 500 milliseconds
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

public class Main {
    public static void main(String[] args) {
        SimpleThread thread1 = new SimpleThread();
        SimpleThread thread2 = new SimpleThread();

        thread1.start(); // Start first thread
        thread2.start(); // Start second thread
    }
}
```



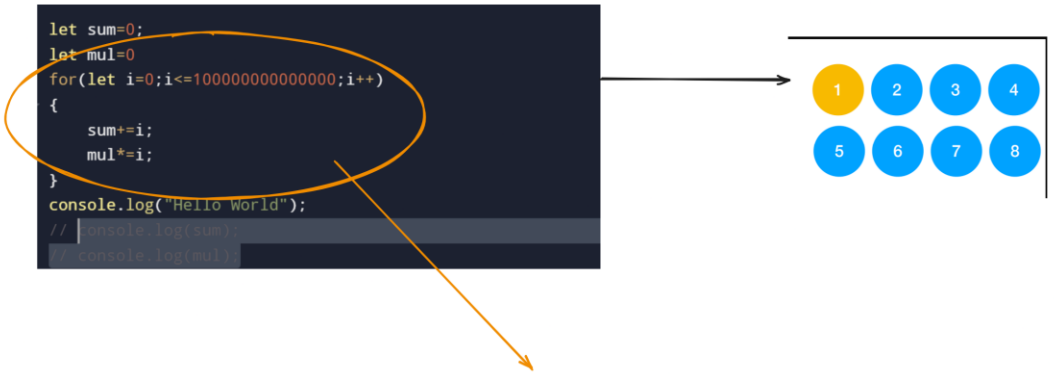


The main difference between a processor and a core is that a processor is the entire computer chip, while a core is a single processing unit within that chip.

JS can only use one of these at a time
It is single threaded
This is why it is considered to be a bad language for scalable systems
There is a way to make it use all cores of your machine

Chapter 17:Solution for Synchronous Nature of JavaScript :

->We have seen that JavaScript is single threaded that is if a process is running in js then the thread inside that process will run sequentially and once a thread has been executed then only the flow of execution will shif to next thread or else that perticular thread will hog the entire cpu-core



Now the problem with a synchronous task is until this for-loop has not executed the console.log("Hello world") wont execute

```
let sum=0;
let mul=1;
for(let i=0;i<=10000000000000000;i++)
{
    sum+=i;
    mul*=i;
}
```

This loop is fairly doing a large computation and long steps
of iteration while also executing two child thread ,the problem here
is large computaion and so many number of iteration cause of which our js
thread will stucked on this perticular loop more 5-6 minute easily

and in real word example also sometime when we are reading a file
,or trying to fetch data from database pased on many parameter like internet
speed or server optimization or to check if server is up and running or is it
down it may take some time to fetch those data and we dont our program executor
to get stucked on one thread if the execution takes some time ,and this is one of
the disadvantage while dealing with pure js is its synchronous

if there was a way that we can make those task async which can take long time
to execute so that while those task are being executed our thread-executor
can shift to other task which can get execute in less time

Well the answer is there is a way that we can replicate the property of
multi threading programming of java in javascript also

and that is by using webApis :so unlike browser our new browser comes with set
of pre built function which are actually not written in JavaScript V8 engine
but they are written for bowser which can be integrated with these pre built function
for web

and even the latest nodejs -a local runtime which takes js code outside of broswer comes
with v8 engine + webAPI bundled together

Note :Javascript interpreter v8 engine itself dont have these function but some very skill full
and brilient people out there wrote these webAPI for browser engine which is diffrent from
v8 engine ,and we as developer can intgrate those webAPI in our program to make our js code a bit more
optimal and replicate the multi-threading behaviour

Note:we are just replicating the multi threading behaviour js is not multi threaded ,these webAPI will make it
look like its multi threaded but its not and we will see how they work behind the scene

What is Web API?

API stands for **A**pplication **P**rogramming **I**nterface.

A Web API is an application programming interface for the Web.

A Browser API can extend the functionality of a web browser.

Javascript is a single threaded language. This means it has one call stack and one memory heap. As expected, it executes code in order and must finish executing a piece code before moving onto the next. It's synchronous, but at times that can be harmful. For example, if a function takes awhile to execute or has to wait on something, it freezes everything up in the meanwhile.

A good example of this happening is the window alert function. `alert("Hello World")`

You can't interact with the webpage at all until you hit OK and dismiss the alert. You're stuck.

So how do we get asynchronous code with Javascript then?

Well, we can thank the Javascript engine (V8, Spidermonkey, JavaScriptCore, etc...) for that, which has Web API that handle these tasks in the background. The call stack recognizes functions of the Web API and hands them off to be handled by the browser. Once those tasks are finished by the browser, they return and are pushed onto the stack as a callback.

So how do we get asynchronous code with Javascript then?

Well, we can thank the Javascript engine (V8, Spidermonkey, JavaScriptCore, etc...) for that, which has Web API that handle these tasks in the background. The call stack recognizes functions of the Web API and hands them off to be handled by the browser. Once those tasks are finished by the browser, they return and are pushed onto the stack as a callback.

Open your console and type `window` then press enter. You'll see most everything the Web API has to offer. This includes things like ajax calls, event listeners, the fetch API, and setTimeout. Javascript uses low level programming languages like C++ to perform these behind the scenes.

Let's look at a simple example, run this code your console:

```
console.log("first")
setTimeout(() => {
    console.log("second")
}, 1000)
console.log("third")
```

What did we get back?

```
first
third
undefined
second
```

Feels odd, right? Well, let's break this down line by line:

`console.log("first")` is on the stack first, so it gets printed. Next, the engine notices setTimeout, which isn't handled by Javascript and pushes it off to the WebAPI to be done asynchronously. The call stack moves on without caring about the code handed off to the Web APIs and `console.log("three")` is printed.

Next, the Javascript engine's event loop kicks in, like a little kid asking "Are we there yet?" on a road trip. It starts firing, waiting for events to be pushed into it. Since the `setTimeout` isn't finished, it returns `undefined`, as the default, well because it hasn't been given the value yet. Once the callback finally does hits we get `console.log("second")` printed.

link for demo: https://i.giphy.com/media/v1.Y2lkPTc5MGI3NjExajN6bnd4NjNudGoweDdsNDI0b3VubzlnZjB4a3cwa2Mza2djb3lrMyZlcD12MV9pbnRlcm5hbF9naWZfYnlfaWQmY3Q9Zw/PoGJhyh6hM12hjWeiS/giphy.gif

Now to understand how the webAPI works
we need to understand about call stack vs webAPI stack
vs callback queue and event loop

there is a video on youtube taken at JSconf must watch

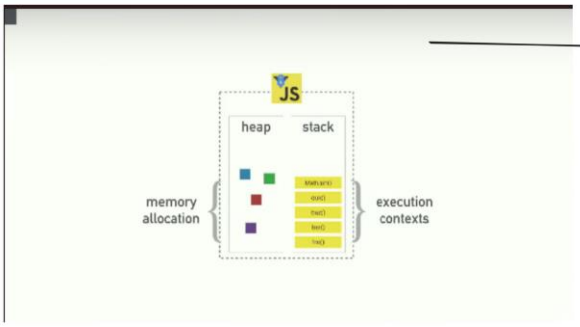
https://youtu.be/8aGhZQkoFbQ?si=MfaCjjUZY1sXdQAG

Link to demo:
https://i.giphy.com/media/v1.Y2lkPTc5MGI3NjExajN6bnd4NjNudGoweDdsNDI0b3VubzlnZjB4a3cwa2Mza2djb3lrMyZlcD12MV9pbnRlcm5hbF9naWZfYnlfaWQmY3Q9Zw/PoGJhyh6hM12hjWeiS/giphy.gif

Link to video:

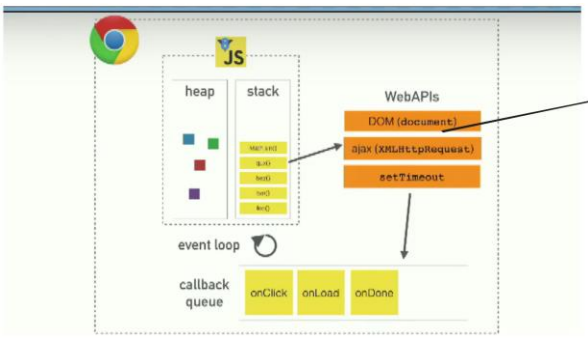
https://youtu.be/8aGhZQkoFbQ?si=MfaCjjUZY1sXdQAG

Lets try to understand js runtime architecture :



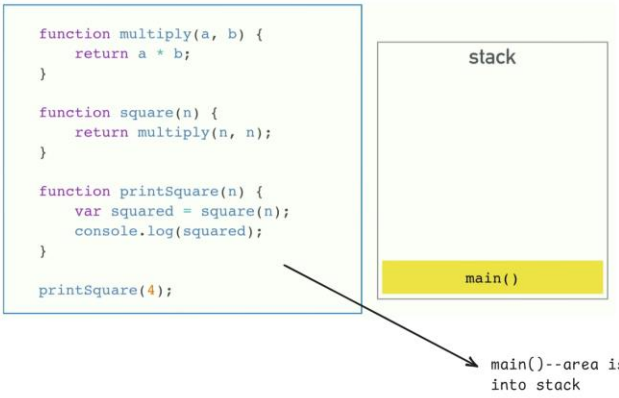
this what js browser v8 engine runtime comes bundeled
with and that is heap for object-dynamic memory allocation
and stack which executes thread and process onto the core

so if we goto v8 engine repo and try to find webAPI like
setTimeout we won't be able to find it cause that's not built into it

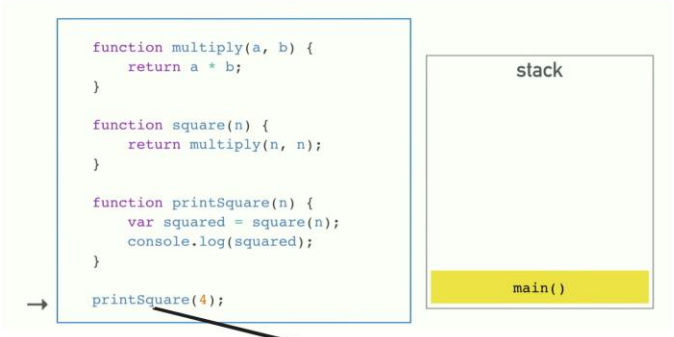


These webAPI are built in Borwser and not in v8 but our js program
can be integrated with these webAPI and replicate the multi-threading behaviour

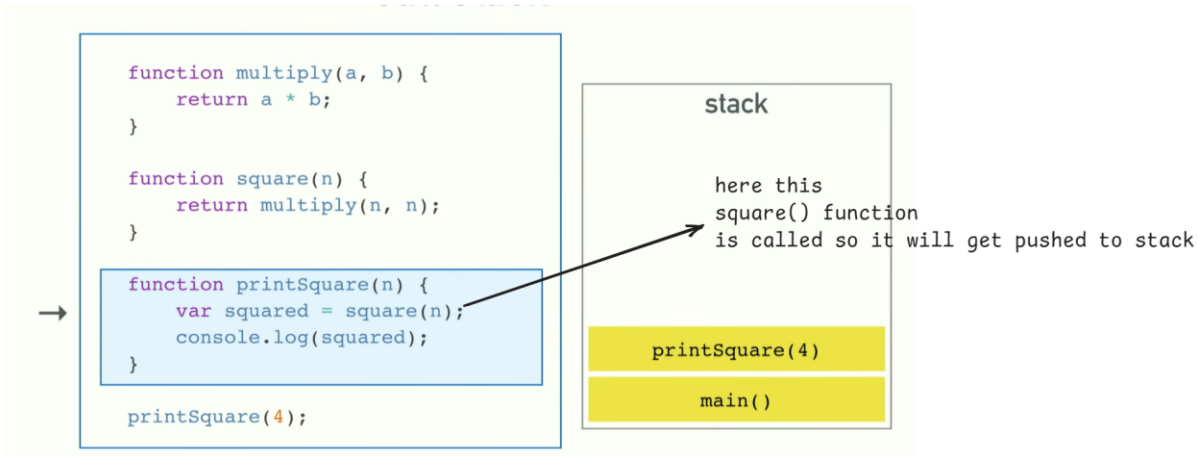
Lets see how the single-thread js execute it's thread and process in call stack:

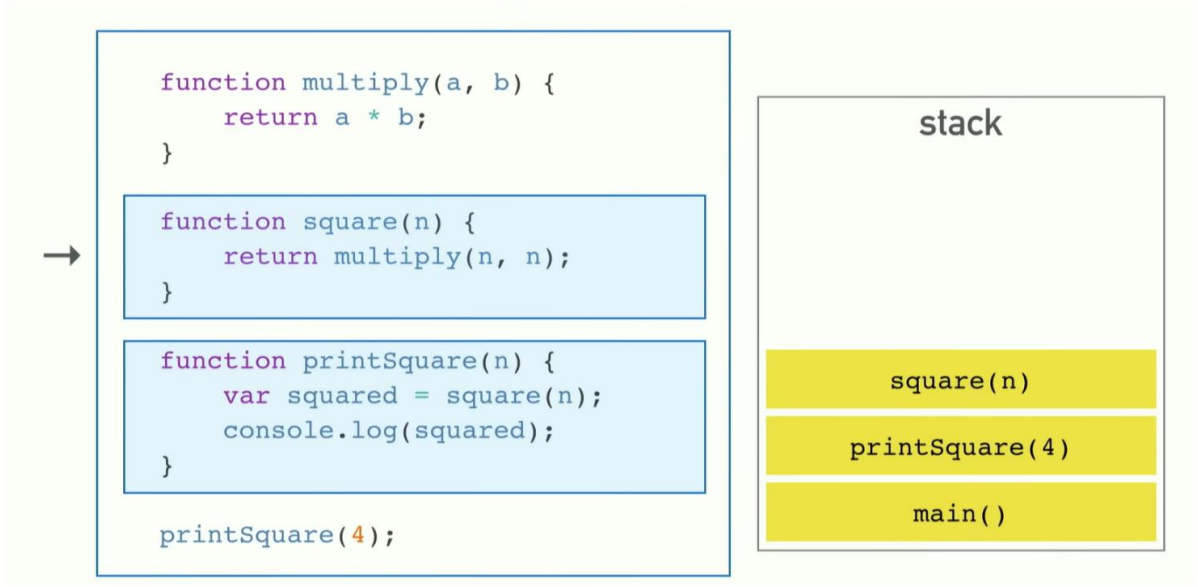


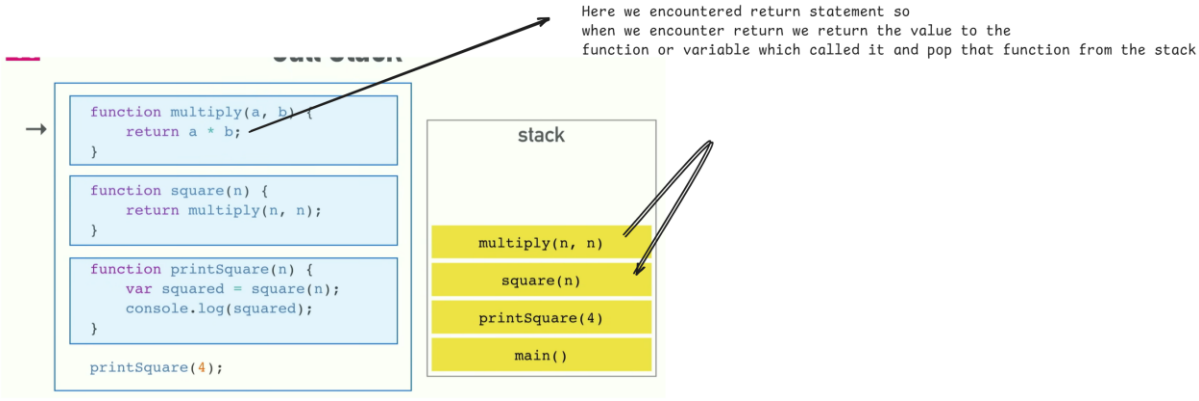
main()--area is called in which we write js code basically js -file is loaded into stack



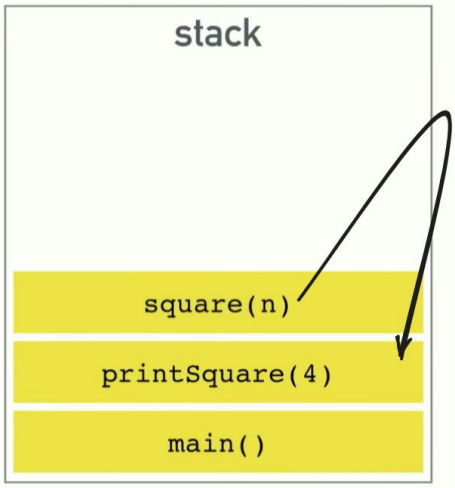
printSquare(4) is called it will get pushed onto the stack


function multiply(a, b) {
return a * b;
}
function square(n) {
return multiply(n, n);
}
function printSquare(n) {
var squared = square(n);
console.log(squared);
}
printSquare(4);
stack
here this
square() function
is called so it will get pushed to stack
printSquare(4)
main()

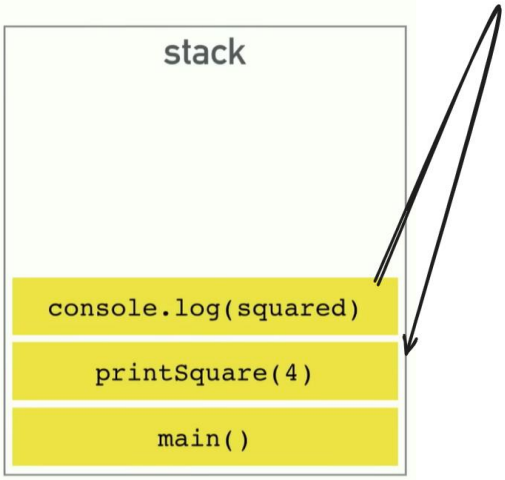

function multiply(a, b) {
return a * b;
}
function square(n) {
return multiply(n, n);
}
function printSquare(n) {
var squared = square(n);
console.log(squared);
}
printSquare(4);
stack
square(n)
printSquare(4)
main()

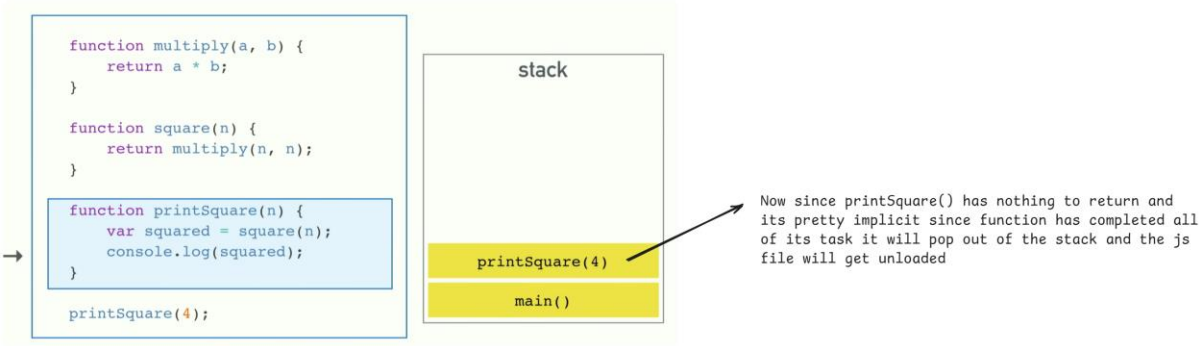

Here we encountered return statement so
when we encounter return we return the value to the
function or variable which called it and pop that function from the stack
function multiply(a, b) {
return a * b;
}
function square(n) {
return multiply(n, n);
}
function printSquare(n) {
var squared = square(n);
console.log(squared);
}
printSquare(4);
stack
multiply(n, n)
square(n)
printSquare(4)
main()

```
function multiply(a, b) {
    return a * b;
}

function square(n) {
    return multiply(n, n);
}

function printSquare(n) {
    var squared = square(n);
    console.log(squared);
}

printSquare(4);
```




```
function multiply(a, b) {
    return a * b;
}

function square(n) {
    return multiply(n, n);
}

function printSquare(n) {
    var squared = square(n);
    console.log(squared);
}

printSquare(4);
```

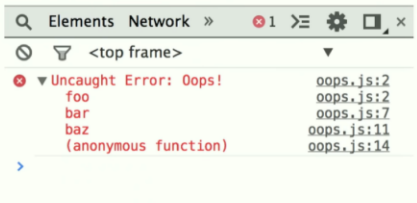





```
function multiply(a, b) {
    return a * b;
}

function square(n) {
    return multiply(n, n);
}

function printSquare(n) {
    var squared = square(n);
    console.log(squared);
}

printSquare(4);
```

We can even see this stack or call stack-calling function stack on our console
if we try to throw an error to make the call stack stuck purposly

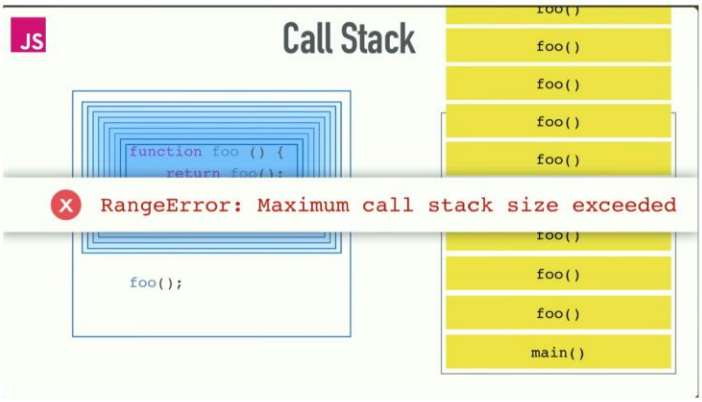
```
function foo() {
  throw new Error('Oops!');
}

function bar() {
  foo();
}

function baz() {
  bar();
}

baz();
```



Call stack play a very important role especially when we deal with recursive function

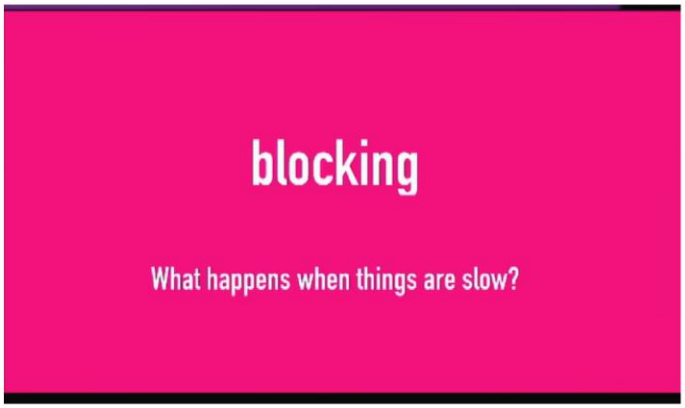
```
function foo () {
    return foo();
}

foo();
```

This basically a recursive function without base condition
that means there is no condition to terminate this recursive call
so foo() will keep calling itself and js will keep pushing it
to stack

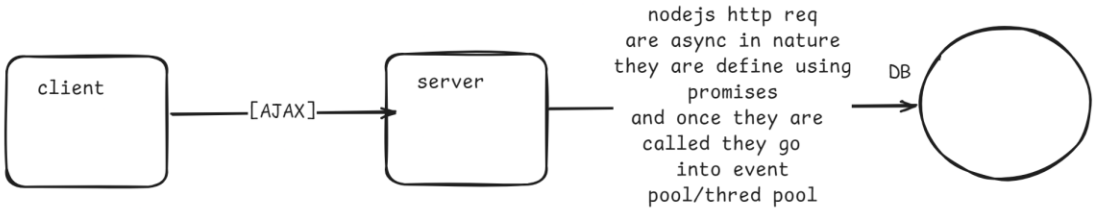


Q.So the big question is why being synchronous is a bad thing
-well cause some process are slow

Q.Then another question is what happens when things are slow
->well since js is single threaded so if a task or a thread is slow then blocking happens



So basically in real case senerio the request which we sent from client to server or server to DB
those request are async



this is asyn req to fetch
data so once the req is evoked
client will communicate with client but since its
async in nature it wont hung the executation thread
it will send the request and that request will go through
AJAX engine and then that ajax engine will forward the
http request and while that http request is
being resolved our function which envoked the request
will wait for http response from server in ajax engine queue
while our js thread is executing the other code

->and since our request and fetching of data is asynchronous our code can render site without getting stucked on a single thread

But lets assume for our understanding purpose that our request are not async instead they were synchronous code now statement like console.log() but performing a loop from 1 to maybe million or 10 million is slow performing image processing is slow and network request are slow

You may ask why network request are slow ,well there are many reason lets see what they are

1. Network Latency

- **Distance**: The physical distance between the client and server affects the time it takes for data to travel back and forth. Longer distances result in higher latency.
- **Routing**: Data packets may take multiple hops through various routers, each adding processing time.

2. Bandwidth Limitations

- **Network Capacity**: The maximum amount of data that can be transmitted over a network in a given time can limit performance. If the bandwidth is low or saturated, requests will take longer to complete.

3. Server Response Time

- **Processing Delays**: The server might take time to process the request, especially if it involves complex computations or database queries.
- **High Load**: If the server is handling too many requests at once, it may take longer to respond.

4. Round Trip Time (RTT)

- The time it takes for a request to go to the server and for the response to return. This includes all the time taken for routing and processing.

5. Network Congestion

- High traffic on the network can cause delays as packets may experience queuing at routers or switches.

6. Protocol Overhead

- Different network protocols (like HTTP) have varying levels of overhead. For example, HTTP/1.1 has more overhead compared to HTTP/2, which can lead to slower performance.

7. Data Size

- Larger payloads take longer to transmit. If the data being sent or received is large, it can contribute to slow requests.

8. Client-Side Factors

- **Browser Performance**: The client's device and browser can also affect how quickly requests are made and how responses are processed.
- **JavaScript Execution**: If the client is doing heavy computations or running blocking scripts, it may delay handling the network response.

9. Connection Quality

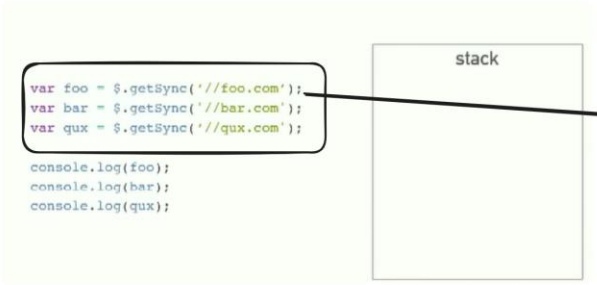
- **Wi-Fi vs. Wired**: Wireless connections can be less stable and slower compared to wired connections, leading to potential delays.
- **Interference**: Physical obstructions or electronic interference can impact wireless signal quality.

10. Error Handling and Retries

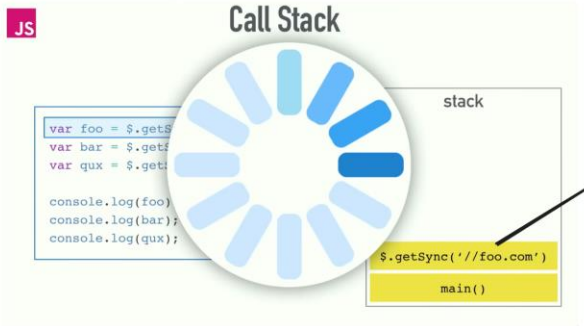
- If a request fails and needs to be retried, it will take longer to complete. This can happen due to transient network issues or server errors.

So taking into consideration every factor of why network request are slow if they "were" synchronous then this will create a fallout for our code probably in system crash also

but for learning process lets try to assume they are sync in nature

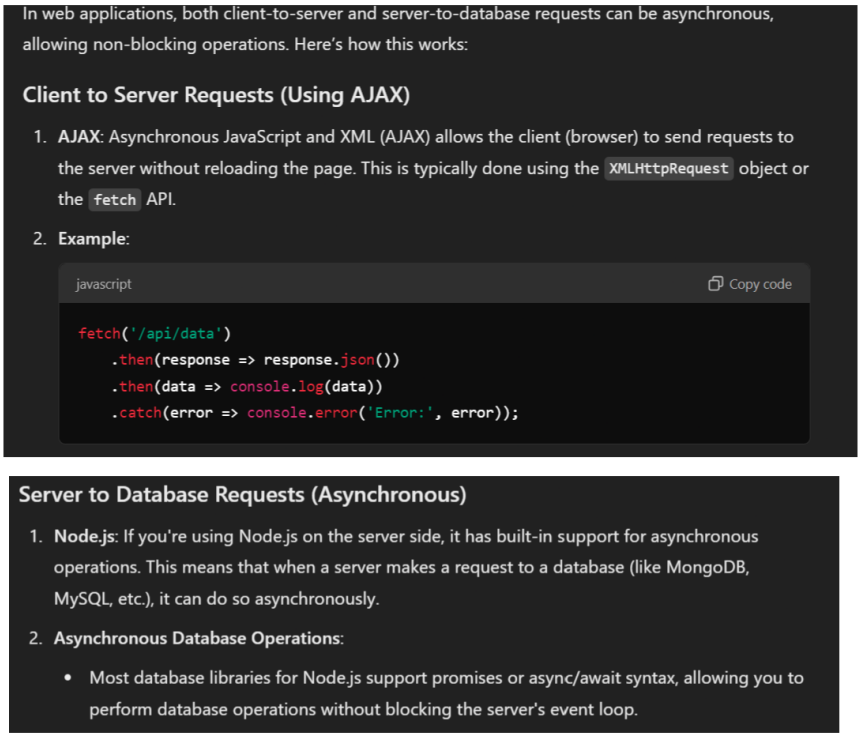


so lets assume these 3 here are sync process
of requesting from foo.com bar.com and quz.com
each will get called on stack one by one
but since they are sync if the data which
is requested is huge than it will take some time
to iterate over that data and get back response



$.getSync("//foo.com") request is evoked and now
my thread is hunged it's processing the request but
since it's single thread plus request is sync in nature
code below this function wont get executed at all
and if there is a code to render the site then that site
wont get rendered

AND that is why request are async in nature especially network request are :



In web applications, both client-to-server and server-to-database requests can be asynchronous, allowing non-blocking operations. Here's how this works:

Client to Server Requests (Using AJAX)

1. **AJAX**: Asynchronous JavaScript and XML (AJAX) allows the client (browser) to send requests to the server without reloading the page. This is typically done using the `XMLHttpRequest` object or the `fetch` API.
2. **Example**:

javascript Copy code

```
fetch('/api/data')
    .then(response => response.json())
    .then(data => console.log(data))
    .catch(error => console.error('Error:', error));
```

Server to Database Requests (Asynchronous)

1. **Node.js**: If you're using Node.js on the server side, it has built-in support for asynchronous operations. This means that when a server makes a request to a database (like MongoDB, MySQL, etc.), it can do so asynchronously.
2. **Asynchronous Database Operations**:
   - Most database libraries for Node.js support promises or async/await syntax, allowing you to perform database operations without blocking the server's event loop.

3. **Example Using MongoDB with Mongoose:**

javascript Copy code

```javascript
const mongoose = require('mongoose');

// Connect to MongoDB
mongoose.connect('mongodb://localhost/mydatabase', { useNewUrlParser: true, useUnified

const User = mongoose.model('User', new mongoose.Schema({ name: String }));

// Asynchronous function to get users
async function getUsers(req, res) {
    try {
        const users = await User.find(); // Asynchronous request to the database
        res.json(users);
    } catch (error) {
        res.status(500).send('Error fetching users');
    }
}

// Example route
app.get('/users', getUsers);
```

- **Non-blocking**: Asynchronous operations allow other code to run while waiting for a database response, improving performance and responsiveness.
- **Promises and Async/Await**: Modern JavaScript features make it easier to work with asynchronous code. Using `async` functions and `await` expressions allows for writing asynchronous code that looks synchronous.

What is AJAX:

What Is AJAX?

AJAX stands for Asynchronous JavaScript and XML. It is a set of web development to build more responsive websites and applications. AJAX allows web pages to update their content without users having to reload the page.

AJAX is derived from JavaScript's function to allow for a more interactive experience. **JavaScript** creates, adds, and manages dynamic structure by monitoring which content requires real-time updates while a visitor is accessing a website.

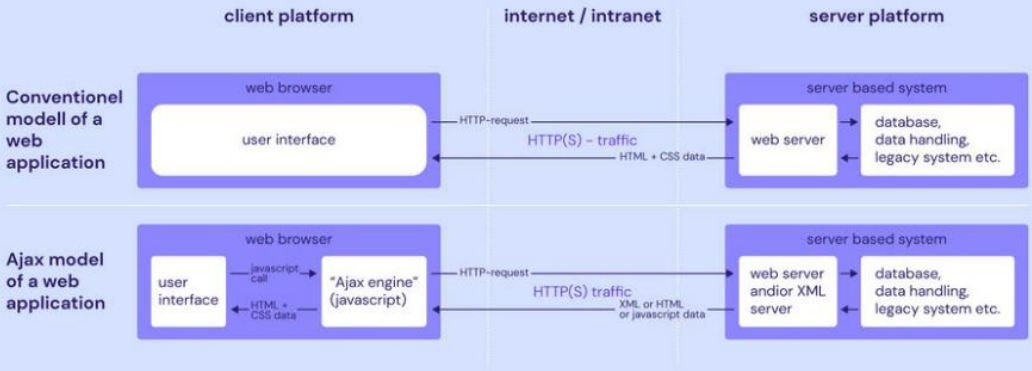
How Does AJAX Work?

AJAX comprises the following technologies:

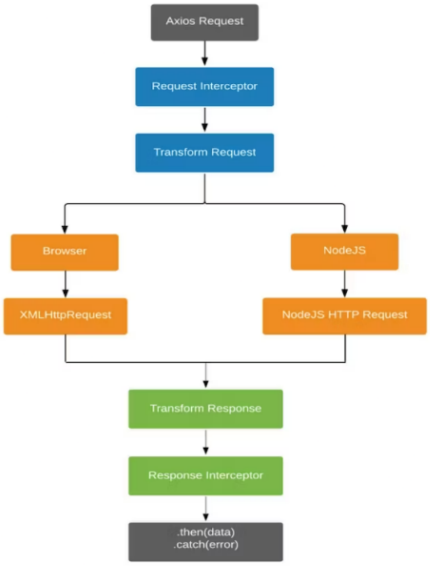
- **XHTML and CSS** – for presenting the information.
- **The Document Object Model (DOM)** – for the dynamic display data and its interaction.
- **XML, HTML, and XSLT** – for data interchange and manipulation. However, many developers have replaced XML with **JSON** since it originated from JavaScript.
- **XMLHttpRequest object** – allows asynchronous communication with the web server.
- **JavaScript** – the **programming language** that links all these web technologies.

HOSTINGER

Three. Two. Online

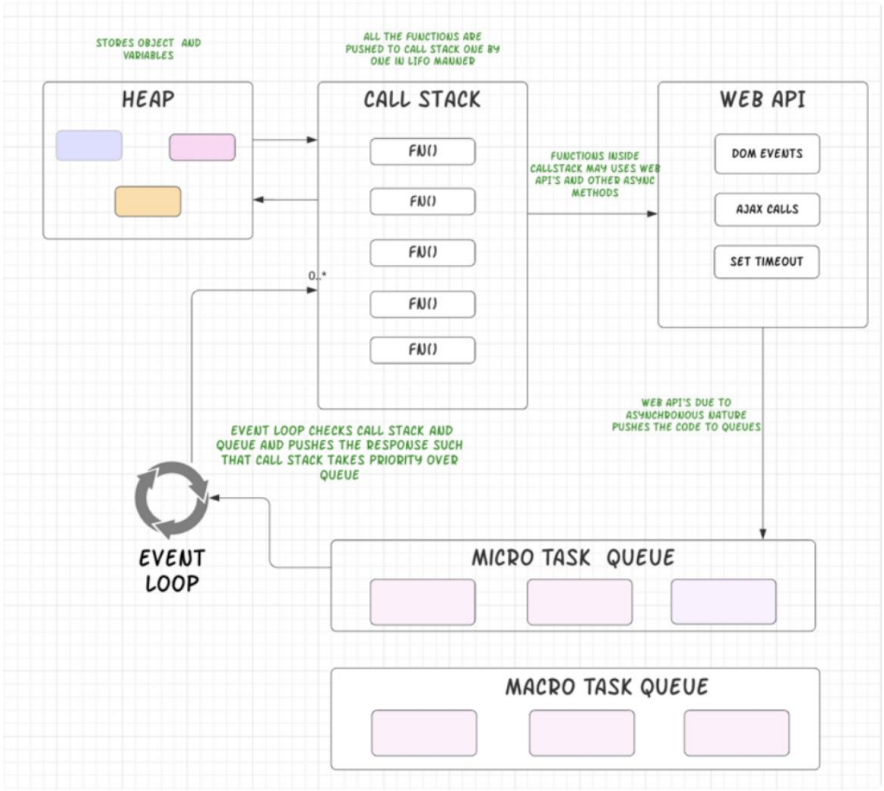


→One of the popular asyn request function handler is axios



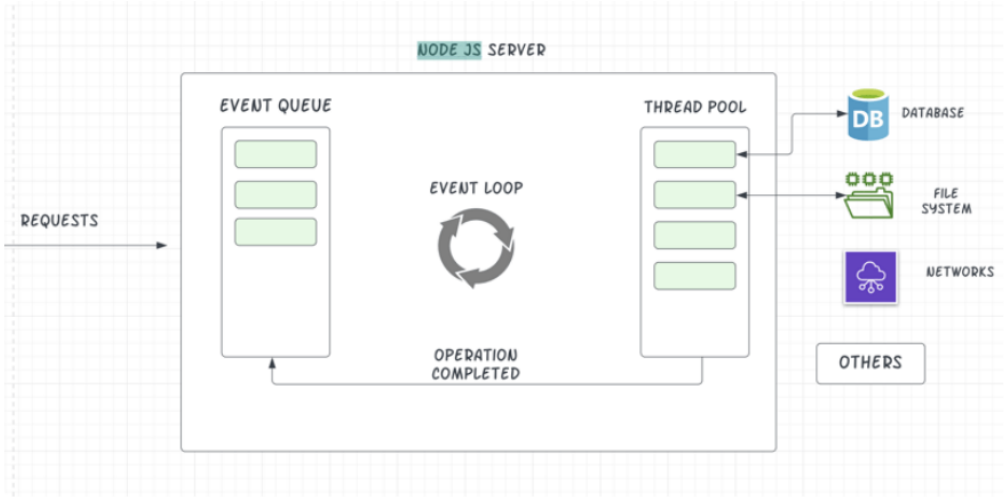
Okay lets continue with Browser architecture and nodejs architecture to run js

Event Loop in Browser



1. **Heap** - It stores all the object reference and variables that we define in our function.
2. **Call Stack** - All the function that we use in our code is stacked here in LIFO manner such that the last function is at the top and first function is at the bottom.
3. **Web API's** - These API's are provided by browser which provides additional functionality over V8 engine. The functions which uses these API's are pushed to this container which on completion of the response of Web API is popped out of this container.
4. **Queues** - The queues are used to compute the asynchronous code response such that it doesn't block engine to execute further.
    - **Macro Task Queue** - This queue executes async functions like DOM events, Ajax calls and setTimeout and has lower priority than Job queue.
    - **Micro Task Queue** - This queue executes async functions which uses promises and has higher precedence over Message Queue.

Event Loop in Node Js

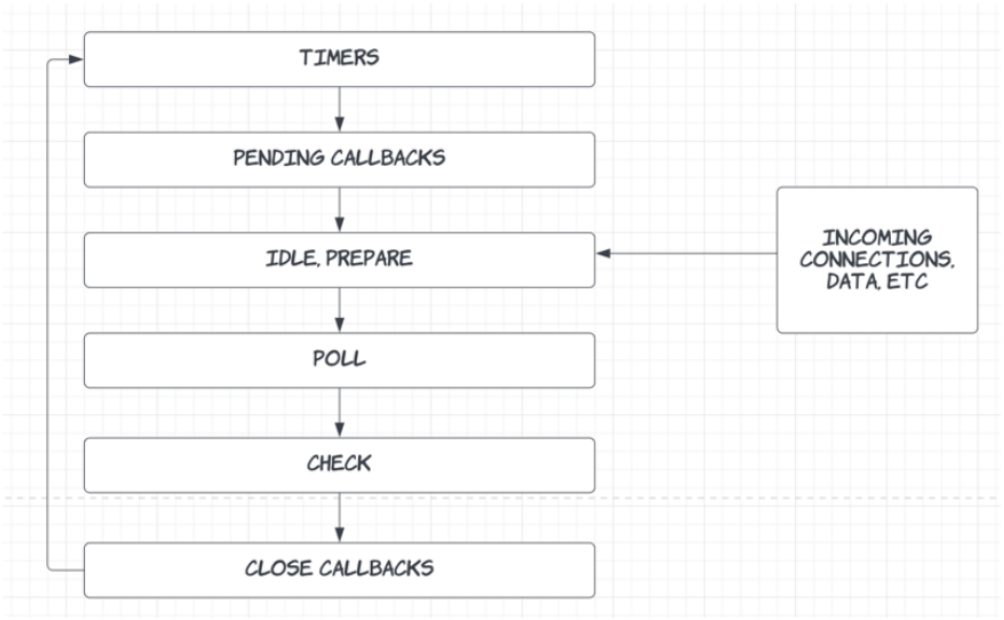


The Node Server consist of following parts:

1. **Event Queue** - On completion of the Thread Pool a callback function is issued and sent to the event queue. When call stack is empty the event goes through the event queue and sends callback to the call stack.

2. **Thread Pool** - The thread pool is composed of 4 threads which delegates operations that are too heavy for the event loop. I/O operations, Opening and closing connections, setTimeouts are the example of such operations.

3. Event loop in Node Js has different phases which has **FIFO** queue of callbacks to execute. When event loop enters a given phase it operates callbacks in that phase queue until the queue has been exhausted and maximum number of callbacks has executed and then moves to next phase.



The Event Loop is an endless loop which waits for the tasks, executes them and then sleeps until it receives more tasks. The event loop executes tasks from queue only when stack is empty. It processes the oldest task first and allows us to use callbacks and promises.

Difference between both the event loops?

1. First difference is node uses a thread pool to manage disk I/O. It executes the I/O and other timer API's asynchronously.

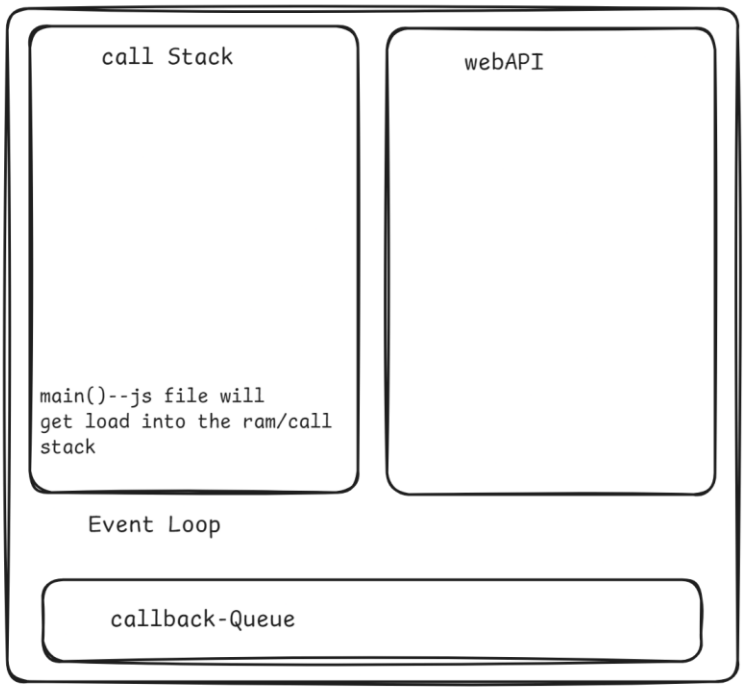
2. Browser does not have **setImmediate()** function. This function execute once the I/O operation is done, if particular code is inside this it will be executed first. Whereas in **setTimeout()** callback function is executed after given minimum threshold value in milliseconds.

3. Node Js event loop has multiple phases and each phase handle specific type of tasks whereas browser has micro task and macro task queue within which all the tasks are processed in order they were placed into the queue.

4. In a browser when you open a page in a tab, you actually create a process in which there can be multiple threads, such as JS engine, page rendering, HTTP request threads and many more. Whereas in Node JS when you initiate a Client Request which performs Blocking I/O operations, event loop picks up a thread and assign the client request to that thread as Event loop is single threaded.

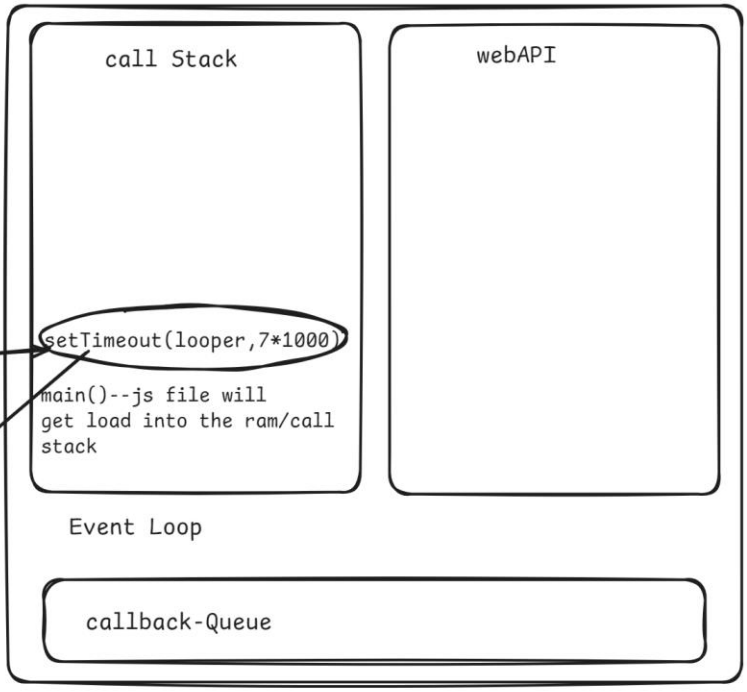
Let's understand how the program will now execute in call stack,webAPI and callback queue

```
function looper()
{
    let sum=0
    for(let i=0;i<=10;i++)
    {
        sum+=i;
    }
    console.log(sum);
}
setTimeout(looper,7*1000);
console.log("Hello World");
```



will
read this
line
first and
put it
in call stack

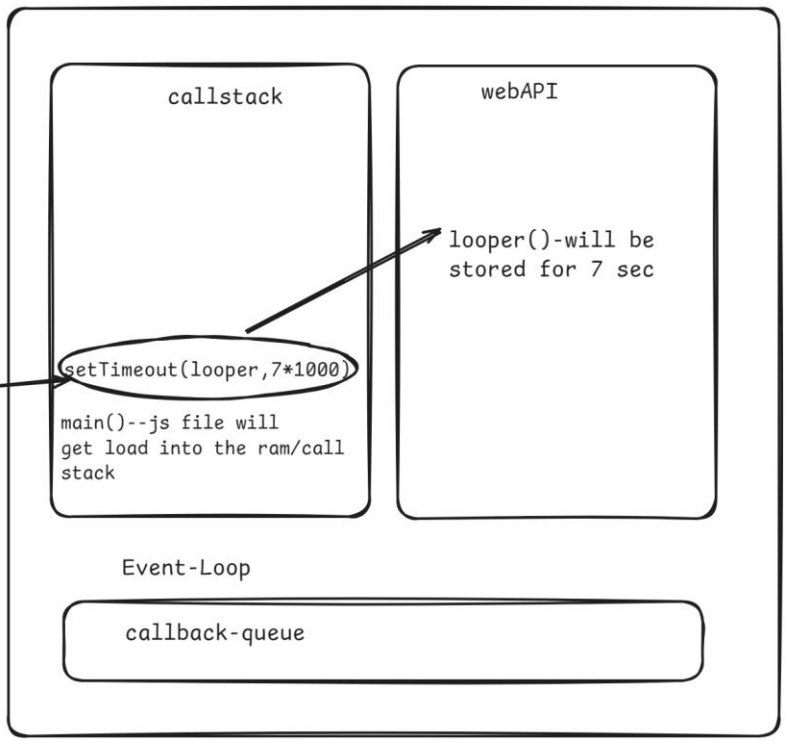
```
function looper()
{
    let sum=0
    for(let i=0;i<=10;i++)
    {
        sum+=i;
    }
    console.log(sum);
}
setTimeout(looper,7*1000);
console.log("Hello World");
```



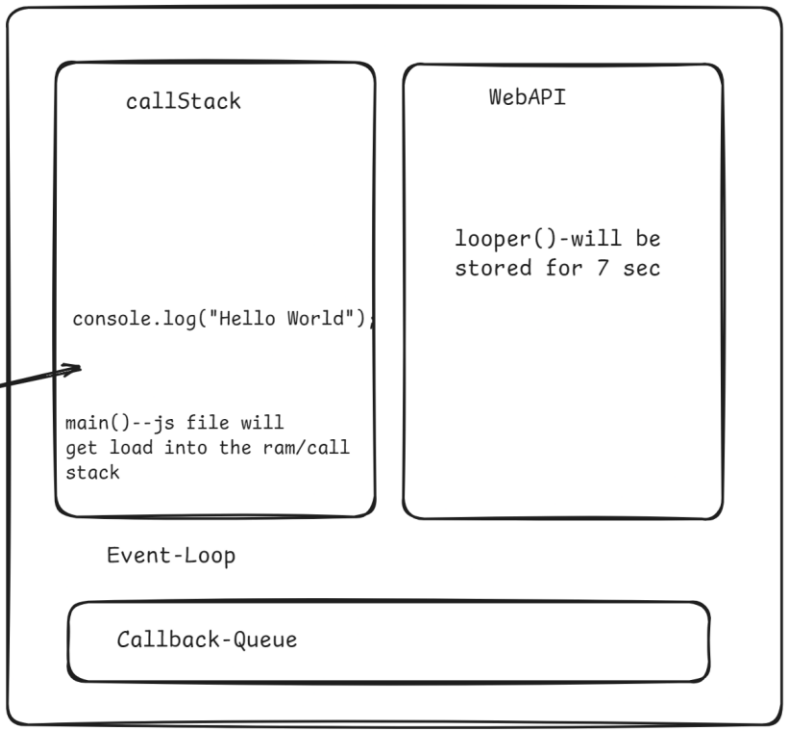
Now here once
the setTimeout()
high order function
is loaded
in to call stack
browser or nodejs
will automatically
figure it out its
a webAPI/c++ api

so the call back function reference inside
setTimeout() will get stored in webAPI stack
a browser stack which isolated from the actual
ram/call stack executor basically it will
get stored inside browser / nodejs thread pool
storage for that perticular time

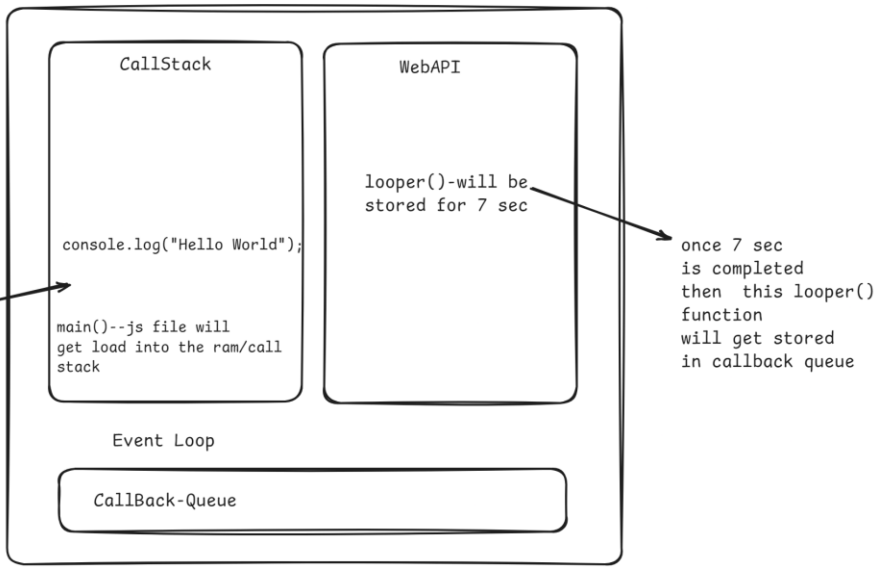
```
function looper()
{
    let sum=0
    for(let i=0;i<=10;i++)
    {
        sum+=i;
    }
    console.log(sum);
}
setTimeout(looper,7*1000);
console.log("Hello World");
```



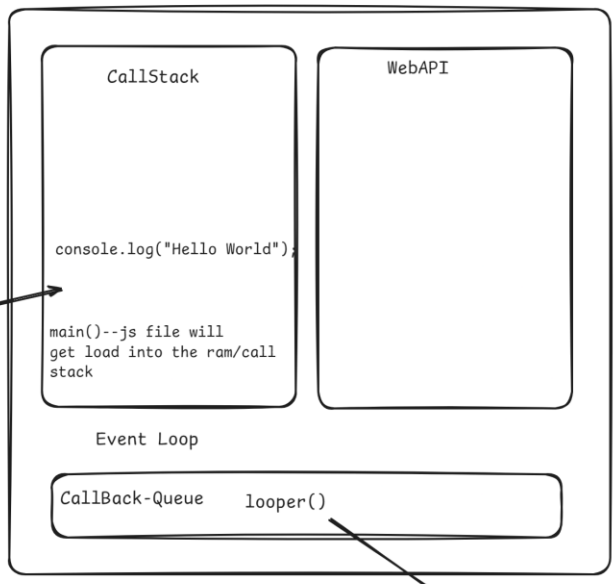

```
function looper()
{
    let sum=0
    for(let i=0;i<=10;i++)
    {
        sum+=i;
    }
    console.log(sum);
}
setTimeout(looper,7*1000);
console.log("Hello World");
```



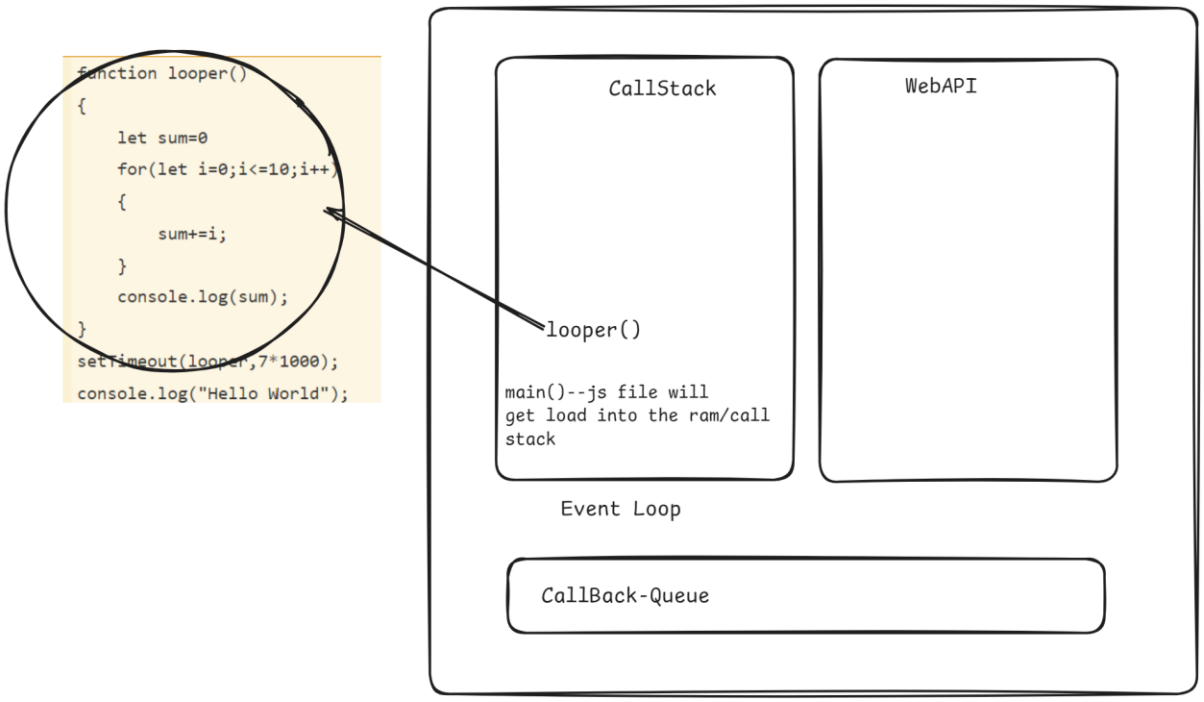
```
function looper()
{
    let sum=0
    for(let i=0;i<=10;i++)
    {
        sum+=i;
    }
    console.log(sum);
}
setTimeout(looper,7*1000);
console.log("Hello World");
```



```
function looper()
{
    let sum=0
    for(let i=0;i<=10;i++)
    {
        sum+=i;
    }
    console.log(sum);
}
setTimeout(looper,7*1000);
console.log("Hello World");
```



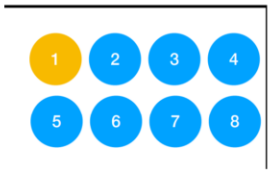
Now this looper()
function wont
get executed till our call
stack is not empyt once the call stack
stop execution all the code
of the file then -event loop
will push it back to call stac




```
function looper()
{
    let sum=0
    for(let i=0;i<=10;i++)
    {
        sum+=i;
    }
    console.log(sum);
}
setTimeout(looper,7*1000);
console.log("Hello World");
```

Just because we have used a async function does not make our
js program multi-threaded it's still single threaded it just
that when it encounters a aync function it pushed
it into webapi array/linked-list stack or a thread pool stack
and store it there for a perticular time

once the time limit is comleted it pops out the callback
in async function and then that callback gets stored in
a call back queue and once the call-stack which is
actually beinng ran on our ram logical memory is empty
then [event loop] dequeue the callback and pushed it
into actual call stack and runs on that core

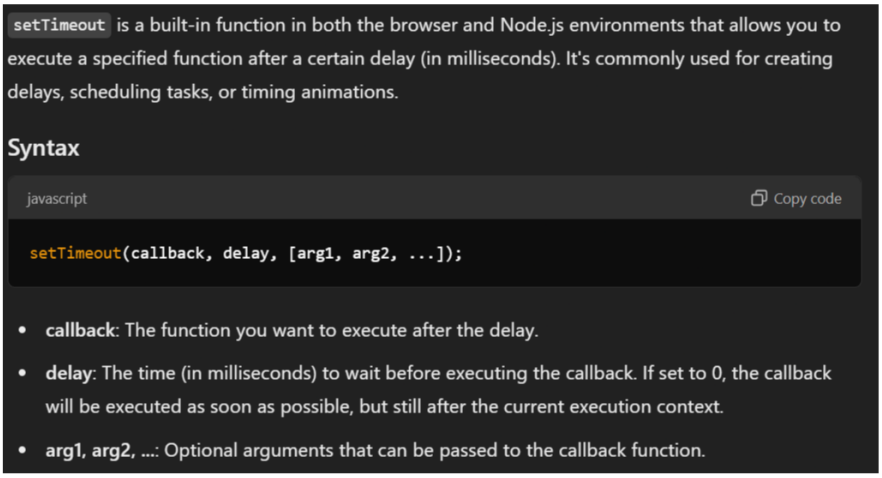


so still running on one core but as
soon as it encounters async function
puts it on hold in a webAPI or thread pool stack

Chapter:18-callback hell ,Promises and async plus await

So this is how basically async-js work behind the scene but lets now look some of the async function and lets try to find a way to make our own async function

1.setTimeout(cb,secondsTo_Delay);--the very first async function build using c++ api to make our js replicate multi threaded behaviour



Example:

```
1 console.log("start");
2
3 //a very easy setTimeout function example will be using a simple anonymous function:
4
5 setTimeout(()=>{
6     console.log("Hello this is a callback inside setTimeout")
7 },1000)
8
9
10 console.log("end")
```

Output

```
Output

start
end
Hello this is a callback inside setTimeout

[Execution complete with exit code 0]
```

Got executed once the event loop stack or the main stack finished their execution and got empty then our event loop in runtime enviorment pushed it to main stack

When you use setTimeout, you're telling Node.js to execute the provided callback function as soon as the current event loop cycle is completed and before moving on to the next phase.

Note :Remember while using setTimeout or any webAPI/async function if we are using a anonymous function /arrow function to pass it as callback then we wont be having any problem

but if we are using a named function and we are trying to pass that named function as call back to a high order function remember not to give or pass it with ()

```
function sumlogger(n1,n2)
{
    return n1+n2;
}

setTimeout(sumlogger(2,4),2000);
```

We cant pass it like this it will give us a error saying

```
node:internal/validators:455
    throw new ERR_INVALID_ARG_TYPE(name, 'Function', value);
    ^

TypeError [ERR_INVALID_ARG_TYPE]: The "callback" argument must be of type function. Received type number (6)
```

```
function sumlogger(n1,n2)
{
    return n1+n2;
}

setTimeout(sumlogger(2,4),3000);
```

Basically as soon as our js interpreter
will scan this line and see that we have
passed a function as input it will immediatly
call it and return value which is syntactically
wrong

plus if we try to replicate a setTimeout function
the code or whoever wrote it may look something like this:

```
function setTimeout(callback,time)
{
    let timeSum=0;
    for(let i=0;i<=time;i++)
    {
        timeSum+=0;
    }
    sleep(timeSum);
    callback();
}--->this is actually pseudo code real code may not look like this
```

so lets see basically when we tries to call a function with bracket what happens actually

```
function sumlogger(n1,n2)
{
    return n1+n2;
}

setTimeout(sumlogger(2,4),2000);
```

```
function setTimeout(callback,time)
{
    let timeSum=0;
    for(let i=0;i<=time;i++)
    {
        timeSum+=0;
    }
    sleep(timeSum);
    callback();
}
```

when setTimeout(callback,time) will be called it will map sumlogger(2,4) with callback parameter

```
function setTimeout(callback:sumLogger(2,4),time)
{
    let timeSum=0;
    for(let i=0;i<=time;i++)
    {
        timeSum+=0;
    }
    sleep(timeSum);
    //callback(); here callback will get replaced with
//sumLogger(2,4)
    sumLogger(2,4)();
}
```

This will be get called immediately and then [()] this bracket will create a error cause of syntax and semantic issue and that's why we cant pass it with bracket cause the way highorder function are define

Similar things happen even when we use callback using anonymous function:
if we envoke the anonymous function in the setTimeout it will give us error

->u may ask yaar shivansh so do u mean when we define a arrow function or anonymous function
they dont get envoked immediately yes in order to invoke a function we need to give it a set ()

```
setTimeout(()=>{
    console.log("Hello this is a callback inside setTimeout")
},1000)
```

Here this function doesnt get envoked immediately in order to envoke them immediately
we use iife

```
setTimeout(
    (
    ()=>{console.log("Hello this is a callback inside setTimeout");}
    )(),
1000)
```

So this is how we envoke a function immediately using
iife that is we wrap anonymous function with opining
bracket "(" and close it with ")" and attack ()
to make a call

```
console.log("start");

//a very easy setTimeout function example will be using a simple anonymous function:

setTimeout(()=>{
    console.log("Hello this is a callback inside setTimeout")
},1000)


console.log("end")
```

will call

```
function setTimeout(callback,time)
{
    let timeSum=0;
    for(let i=0;i<=time;i++)
    {
        timeSum+=0;
    }
    sleep(timeSum);
    callback();
}
```

```
function setTimeout(callback:(()=>console.log("hello")),time)
{
    let timeSum=0
    for(let i=0;i<=time;i++){...}
    sleep(timeSum);
    //callbackk ();
    //replace callback with mapped data
    (()=>console.log("Hello"))()
}//so now here everything is okay it got called using iife
```

But if for some reason accidently we will pass the anonymous function as iife in setTimeout
it will create error:

```
setTimeout(
   (
   ()=>{console.log("Hello this is a callback inside setTimeout");}
   )(),
1000)
```

will call

```
function setTimeout(callback,time)
{
    let timeSum=0;
    for(let i=0;i<=time;i++)
    {
        timeSum+=0;
    }
    sleep(timeSum);
    callback();
}
```

```
function setTimeout(callback:(()=>console.log("hello"))(),time)
{
    let timeSum=0
    for(let i=0;i<=time;i++){...}
    sleep(timeSum);
    //callbackk ();
    //replace callback with mapped data
    (()=>console.log("Hello"))()()//here this two bracket will create error
}
```

That's why in high order function we pass function refernce if it's async

Now we have seen how can we pass callback into setTimeout and we tried to make a pseudo code to understand how setTimeout works but that not the real code there is no sleep() function in js like java so lets try to write our own js code for setTimeout from scratch

```
function mySetTimeout(callback, delay) {
    const start = Date.now(); // Get the current time

    // Create a recursive function to check the elapsed time
    function checkTime() {
        const currentTime = Date.now(); // Get the current time
        if (currentTime - start >= delay) {
            callback(); // Execute the callback if the delay has passed
        } else {
            // Use setImmediate to continue checking without blocking the event loop
            setImmediate(checkTime); //a asynchronous function which will call checkTime again and again and it's set
            //to be excuted immediate after the complete execution of code in our call-stack
        }
    }

    checkTime(); // Start the checking process
}

// Example usage
console.log("Start");
mySetTimeout(() => {
    console.log("Executed after 2 seconds!");
}, 3000);
console.log("End");
console.log(Date.now());

```

```
function mySetTimeout(callback, delay) {
    const start = Date.now(); // Get the current time

    // Create a recursive function to check the elapsed time
    function checkTime() {
        const currentTime = Date.now(); // Get the current time
        if (currentTime - start >= delay) {
            callback(); //and as we can see setTimeout has already inside its function defination has
            //define callback(); so when ever we try to pass a function as call back we only have to pass
            //it's name if it is named function without () and if anonymous function pass it without ()

            //casue setTimeout will automatically attach it .

            // Execute the callback if the delay has passed
        } else {
            // Use setImmediate to continue checking without blocking the event loop
            setImmediate(checkTime); //a asynchronous function which will call checkTime again and again and it's set
            //to be excuted immediate after the complete execution of code in our call-stack
        }
    }

    checkTime(); // Start the checking process
}

// Example usage
console.log("Start");
mySetTimeout(() => {
    console.log("Executed after 2 seconds!");
}, 3000);
console.log("End");
console.log(Date.now());
```

The use of `setImmediate` in Node.js allows you to schedule a function to run immediately after the current event loop cycle, effectively enabling non-blocking behavior. Here's a breakdown of what this means:

Event Loop in Node.js

1. **Single Threaded**: Node.js operates on a single-threaded event loop, meaning it can handle many operations (like I/O tasks) without blocking the main thread.
2. **Phases of the Event Loop**: The event loop consists of several phases, where it processes different types of callbacks. After executing all the current tasks (like I/O or timers), it moves to the next phase.

Using `setImmediate`

- **Immediate Execution**: When you use `setImmediate`, you're telling Node.js to execute the provided callback function as soon as the current event loop cycle is completed and before moving on to the next phase.
- **Non-Blocking**: This allows you to check conditions or perform tasks without blocking other operations that might be happening concurrently. It lets other events and callbacks run in the meantime, which keeps the application responsive.

Example Context

In the context of your `mySetTimeout` function:

1. **Checking Time**: The `checkTime` function checks if the desired delay has passed. If not, it schedules itself to run again using `setImmediate`.
2. **Continuous Loop**: By using `setImmediate`, you're creating a loop that checks the elapsed time without holding up the event loop, allowing other tasks (like handling incoming requests or executing other timers) to run in the meantime.

Summary

Using `setImmediate`:

- Schedules the `checkTime` function to run right after the current execution context is completed.
- Ensures the application remains responsive by allowing other operations to execute in between checks.
- Mimics the behavior of a timer without blocking the event loop, providing an efficient way to wait for a specific duration before executing a callback.

`setImmediate` is **not** a web API; it is a function provided by Node.js. It is specifically designed to execute a single callback function after the current event loop cycle, making it part of Node.js's asynchronous programming model.

Key Points About `setImmediate`

1. **Node.js Environment**:
   - `setImmediate` is available only in Node.js. It allows developers to schedule code execution for the next iteration of the event loop after the current operation completes.
2. **Non-Blocking**:
   - It helps to keep the application responsive by allowing other I/O operations to complete while waiting for the next execution of the scheduled callback.
3. **Usage Context**:
   - Commonly used in scenarios where you want to yield control back to the event loop and allow other operations to execute before continuing with a long-running process.

Comparison to Web APIs

- **Web APIs** (like `setTimeout`, `setInterval`, and `requestAnimationFrame`) are available in the browser environment and are used for scheduling tasks that affect the rendering of the web page.
- In contrast, `setImmediate` is more focused on server-side tasks in Node.js.

Example of `setImmediate`

Here's a simple example of how `setImmediate` is used in Node.js:

javascript Copy code

```javascript
console.log("Start");

setImmediate(() => {
    console.log("This runs after the current event loop cycle");
});

console.log("End");
```

Output

When you run this code, the output will be:

sql Copy code

```sql
Start
End
This runs after the current event loop cycle
```

Summary

- **Not a Web API**: `setImmediate` is exclusive to Node.js, unlike the timing functions found in browsers.
- **Purpose**: It's designed to allow asynchronous programming in Node.js, helping maintain responsiveness and manage timing for callbacks effectively.

Date.now()--what it is and why are we using it?

`Date.now()` is a static method in JavaScript that returns the current timestamp in milliseconds since January 1, 1970, 00:00:00 UTC (the Unix epoch). This method is useful for measuring time intervals, creating timestamps, and performing time-related calculations.

Syntax

javascript Copy code

```javascript
Date.now();
```

Return Value

- Returns a number representing the milliseconds elapsed since the Unix epoch.

Example Usage

1. **Getting the Current Timestamp**:

javascript Copy code

```javascript
const currentTimestamp = Date.now();
console.log(currentTimestamp); // Example output: 1630402335634
```

You can convert the timestamp returned by `Date.now()` into a more readable date and time format using the `Date` object in JavaScript. Here's how you can do that:

Example of Converting `Date.now()` to a Readable Date

1. **Get the Current Timestamp**: Use `Date.now()` to get the current timestamp in milliseconds.
2. **Create a Date Object**: Pass the timestamp to the `Date` constructor to create a new date object.
3. **Format the Date**: Use methods like `toString()`, `toLocaleString()`, or `toISOString()` to get a readable string representation.

Code Example

javascript Copy code

```javascript
// Step 1: Get the current timestamp
const timestamp = Date.now();

// Step 2: Create a Date object from the timestamp
const date = new Date(timestamp);

// Step 3: Format the date
console.log("Default Format: ", date.toString()); // e.g., "Fri Sep 30 2022 12:34:56 GMT+0(
console.log("Locale String: ", date.toLocaleString()); // e.g., "9/30/2022, 12:34:56 PM"
console.log("ISO String: ", date.toISOString()); // e.g., "2022-09-30T12:34:56.789Z"
```

Formatting Options

- `toString()`: Converts the date to a string in the local timezone.
- `toLocaleString()`: Converts the date to a string based on the local timezone and formatting options (you can pass options to customize it).
- `toISOString()`: Converts the date to a string in ISO 8601 format (UTC).

Example with Custom Formatting

You can also create a custom formatted date string using the individual components of the date:

javascript Copy code

```javascript
const formatDate = (date) => {
    const options = { year: 'numeric', month: '2-digit', day: '2-digit', hour: '2-digit',
    return date.toLocaleDateString(undefined, options);
};

const formattedDate = formatDate(date);
console.log("Custom Formatted Date: ", formattedDate); // e.g., "09/30/2022, 12:34:56 PM G
```

Summary

- Use `Date.now()` to get the current timestamp in milliseconds.
- Convert it to a `Date` object for easier manipulation and formatting.
- Utilize methods like `toString()`, `toLocaleString()`, and `toISOString()` to convert it into readable formats.

Now after learning what is async architecture of js
and what are async function lets see how we can build
a sync function

Well the answer is you cannot
build a async function from scratch unitl and
unless you know about CPP and java or golang/rust
which perform multi-threading it's bit tricky to built
them from scratch

But we can make our own function and wrap it
around a pre-built async function,and we actually
have done this :

```
function mySetTimeout(callback, delay) {
    const start = Date.now();
    function checkTime() {
        const currentTime = Date.now(); // Get the current tim
        if (currentTime - start >= delay) {
            callback();
        } else {
            setImmediate(checkTime);

        }
    }

    checkTime(); // Start the checking process
}

// Example usage
console.log("Start");
mySetTimeout(() => {
    console.log("Executed after 2 seconds!");
}, 3000);
console.log("End");
```

This is async function built by us
which is actually wrapped around a
pre built nodejs async function written
in c++ and that is :

setImmediate(parameterFunction)

Now a problem with wrapping a function around pre built async function like
this :

```
function mySetTimeout(callback, delay) {
    const start = Date.now();
    function checkTime() {
        const currentTime = Date.now(); // Get the current time
        if (currentTime - start >= delay) {
            callback();
        } else {
            setImmediate(checkTime);

        }
    }

    checkTime(); // Start the checking process
}

// Example usage
console.log("Start");
mySetTimeout(() => {
    console.log("Executed after 2 seconds!");
}, 3000);
console.log("End");
```

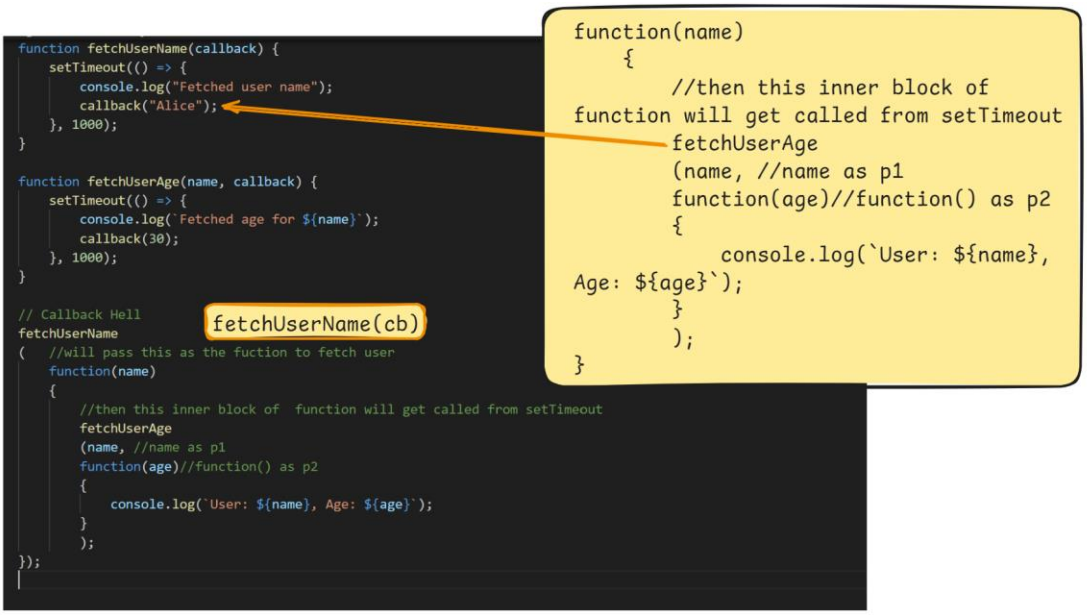
is that it uses call back and what happens is when there call backs chainned together
like asyncFunc(1)--takes a call back cb() and that cb contains calling asyncFunc2

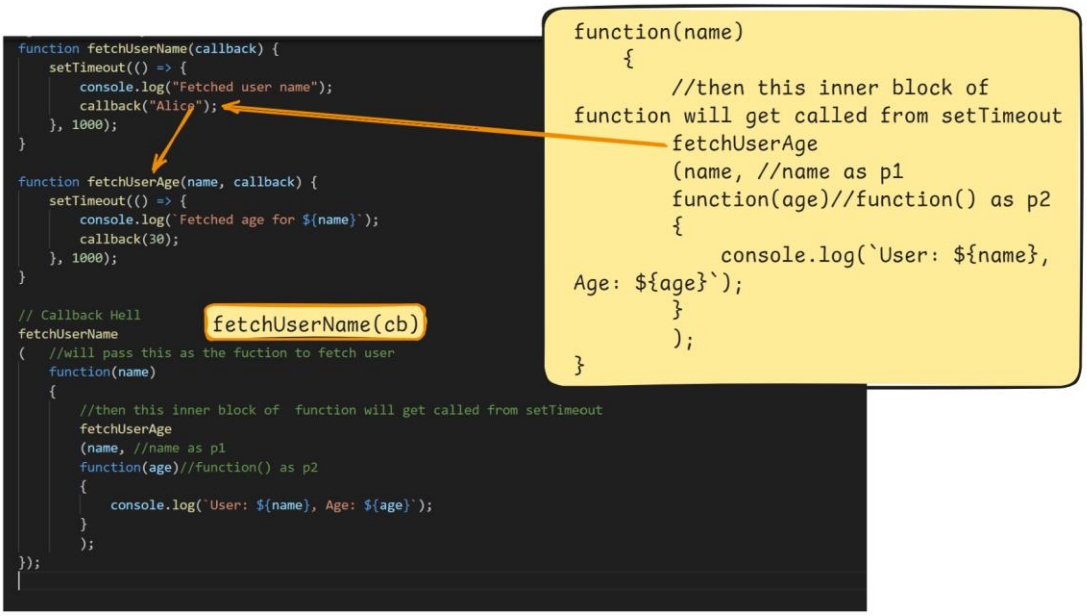
Lets try to understand this with an example :

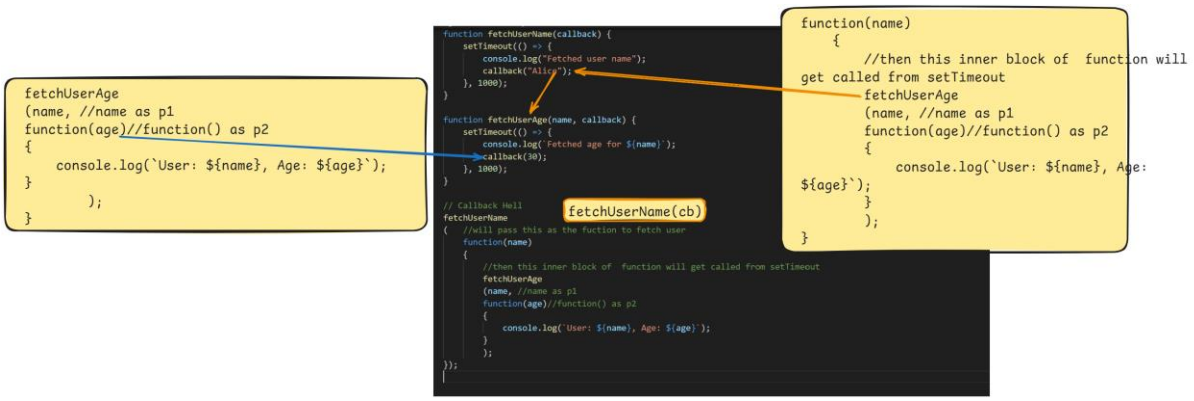
```
function fetchUserName(callback) {
    setTimeout(() => {
        console.log("Fetched user name");
        callback("Alice");
    }, 1000);
}

function fetchUserAge(name, callback) {
    setTimeout(() => {
        console.log(`Fetched age for ${name}`);
        callback(30);
    }, 1000);
}

// Callback Hell
fetchUserName
(   //will pass this as the fuction to fetch user
    function(name)
    {
        //then this inner block of  function will get called from setTimeout
        fetchUserAge
        (name, //name as p1
        function(age)//function() as p2
        {
            console.log(`User: ${name}, Age: ${age}`);
        }
        );
});
```

fetchUserName(cb) is user define async function wrapped around setTimeout


```
function(name)
    {
        //then this inner block of
function will get called from setTimeout
        fetchUserAge
        (name, //name as p1
        function(age)//function() as p2
        {
            console.log(`User: ${name},
Age: ${age}`);
        }
        );
}
```


function fetchUserName(callback) {
setTimeout(() => {
console.log("Fetched user name");
callback("Alice");
}, 1000);
}
function fetchUserAge(name, callback) {
setTimeout(() => {
console.log(`Fetched age for ${name}`);
callback(30);
}, 1000);
}
// Callback Hell
fetchUserName
fetchUserName(cb)
(//will pass this as the fuction to fetch user
function(name)
{
//then this inner block of function will get called from setTimeout
fetchUserAge
(name, //name as p1
function(age)//function() as p2
{
console.log(`User: ${name}, Age: ${age}`);
}
);
});
function(name)
{
//then this inner block of
function will get called from setTimeout
fetchUserAge
(name, //name as p1
function(age)//function() as p2
{
console.log(`User: ${name},
Age: ${age}`);
}
);
}


function fetchUserName(callback) {
setTimeout(() => {
console.log("Fetched user name");
callback("Alice");
}, 1000);
}
function fetchUserAge(name, callback) {
setTimeout(() => {
console.log(`Fetched age for ${name}`);
callback(30);
}, 1000);
}
// Callback Hell
fetchUserName
fetchUserName(cb)
(//will pass this as the fuction to fetch user
function(name)
{
//then this inner block of function will get called from setTimeout
fetchUserAge
(name, //name as p1
function(age)//function() as p2
{
console.log(`User: ${name}, Age: ${age}`);
}
);
});
function(name)
{
//then this inner block of
function will get called from setTimeout
fetchUserAge
(name, //name as p1
function(age)//function() as p2
{
console.log(`User: ${name},
Age: ${age}`);
}
);
}




```javascript
function fetchUserName(callback) {
    setTimeout(() => {
        console.log("Fetched user name");
        callback("Alice");
    }, 1000);
}

function fetchUserAge(name, callback) {
    setTimeout(() => {
        console.log(`Fetched age for ${name}`);
        callback(30);
    }, 1000);
}

// Callback Hell
fetchUserName((name) => {
    fetchUserAge(name, (age) => {
        console.log(`User: ${name}, Age: ${age}`);
    });
});
```

Breakdown of the Example

1. `fetchUserName`: Simulates fetching a user's name after a 1-second delay. It takes a callback that is invoked with the user's name.
2. `fetchUserAge`: Simulates fetching the age of the user after a 1-second delay, taking the name as an argument and invoking a callback with the age.
3. **Callback Hell**: The nested structure of callbacks makes it harder to read and maintain, especially as more operations are added.

Callback hell refers to a situation in programming, particularly in JavaScript, where multiple nested callback functions lead to code that is difficult to read, understand, and maintain. It often occurs when dealing with asynchronous operations, such as making API requests, reading files, or querying databases.

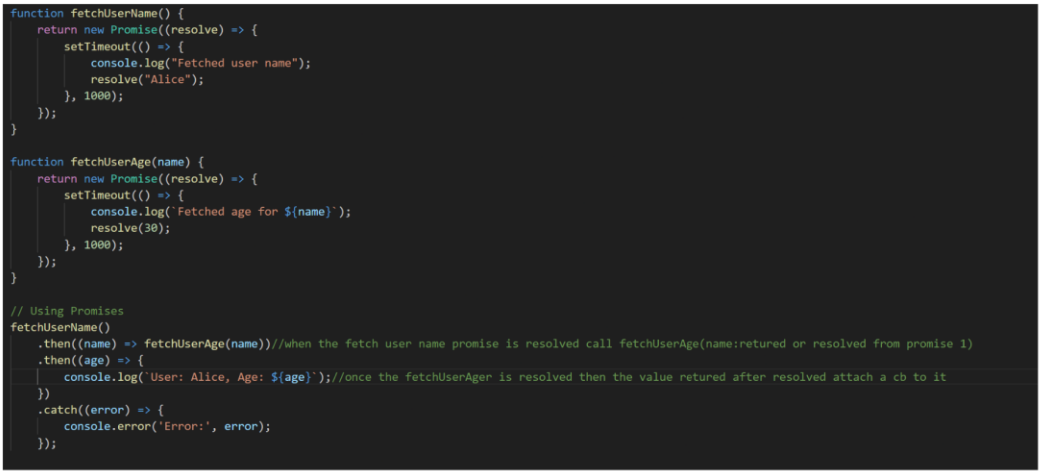
Characteristics of Callback Hell

1. **Deep Nesting**: Functions are nested inside one another, creating a pyramid-like structure.
2. **Hard to Read**: The indentation and structure make it challenging to follow the flow of the program.
3. **Error Handling**: Managing errors can become complicated because each callback may need its own error handling logic.
4. **Difficult to Maintain**: Adding more asynchronous operations leads to further nesting, exacerbating the issues.

Solutions

1. **Promises**: Promises provide a cleaner way to handle asynchronous operations without nesting.
2. **Async/Await**: This syntactic sugar over Promises allows you to write asynchronous code that looks synchronous, improving readability and maintainability.

Promises to the Rescue



```
function fetchUserName() {
    return new Promise((resolve) => {
        setTimeout(() => {
            console.log("Fetched user name");
            resolve("Alice");
        }, 1000);
    });
}

function fetchUserAge(name) {
    return new Promise((resolve) => {
        setTimeout(() => {
            console.log(`Fetched age for ${name}`);
            resolve(30);
        }, 1000);
    });
}

// Using Promises
fetchUserName()
    .then((name) => fetchUserAge(name))//when the fetch user name promise is resolved call fetchUserAge(name:retured or resolved from promise 1)
    .then((age) => {
        console.log(`User: Alice, Age: ${age}`);//once the fetchUserAger is resolved then the value retured after resolved attach a cb to it
    })
    .catch((error) => {
        console.error('Error:', error);
    });
```

Promises:Promises in js are syntactical sugar which makes the call back and making of async function much more readable and maintainale plus easy to grasp

but remember they are just syntactical sugar that is behind the scene they are still using callbacks

Promises in JavaScript are objects that represent the eventual completion (or failure) of an asynchronous operation and its resulting value. They provide a cleaner and more manageable way to handle asynchronous code compared to traditional callbacks.

Key Concepts of Promises

1. **States**: A Promise can be in one of three states:
   - **Pending**: The initial state; the operation is ongoing.
   - **Fulfilled**: The operation completed successfully, and the Promise has a resulting value.
   - **Rejected**: The operation failed, and the Promise has a reason for the failure (an error).
2. **Creating a Promise**: You can create a Promise using the `Promise` constructor. It takes a function called an executor that receives two parameters: `resolve` and `reject`.

javascript Copy code

```javascript
const myPromise = new Promise((resolve, reject) => {
    // Asynchronous operation
    const success = true; // Simulating success or failure

    if (success) {
        resolve("Operation succeeded!"); // Fulfilled
    } else {
        reject("Operation failed!"); // Rejected
    }
});
```

3. **Using Promises**:
   - You can handle the result of a Promise using the `.then()` method for fulfilled Promises and `.catch()` for rejected Promises.

javascript Copy code

```javascript
myPromise
    .then((result) => {
        console.log(result); // "Operation succeeded!"
    })
    .catch((error) => {
        console.error(error); // "Operation failed!"
    });
```

Benefits of Using Promises

- **Improved Readability**: Chaining Promises leads to cleaner code compared to nested callbacks (callback hell).
- **Error Handling**: You can handle errors in a more centralized manner using `.catch()`.
- **Composability**: Promises can be composed together to create complex asynchronous flows easily.

Lets try to see different state of promises :

1.pending

```
function PromsieExecutor()
{
    let catch_Promise=new Promise(function(resolve,reject)
    {
        //if we dont pass any resolve or reject method it will give me a pending status:

    })
    return catch_Promise;
}



let status_check=PromsieExecutor();
console.log(status_check);
```

```
D:\coding>node promises_pending.js
Promise { <pending> }
```

```
function PromsieExecutor2()
{
    let catch_Promise2=new Promise(function(resolve,reject)
    {
        //if we dont pass any resolve or reject method it will give me a pending status:
        //plus if we do pass resolve() or reject function in a callback wrapper it will still show pending status
        setTimeout(function(){resolve("hello")},3000);
        //cause once the function resolve returns me the data there should some sort of handler to catch that data


    })
    return catch_Promise2;
}

let status_check2=PromsieExecutor2();
console.log(status_check2);
```

2.Promise fullfiled -pending status changed to value:

```
let promiseHolder=new Promise(function(resolve){
    resolve("hi there")
  });//intiates a object pointing to class Promise;

  console.log(promiseHolder);
```

```
D:\coding>node promises_valueFilled.js
Promise { 'hi there' }
```

3.Undefined Promise status

```
let promiseHolder=new Promise(function(resolve){
    resolve()//this will return a undefined data
  });//intiates a object pointing to class Promise;

  console.log(promiseHolder);
```

```
D:\coding>node promises_undefined.js
Promise { undefined }
```

Accessing Promise Data :to access a fullfilled /resolved promise we use handler defined inside a promise class

Now we have used what promise are and how to use them and
how basically we can async function to be wrapped around Promise
and how there are 3 different promise state plus how we use there value
using access handler like .then();

but how does this resolve() ,reject() or .then() work behind the scene

so lets try to make our own promise :

```
class MyPromise {
    constructor(executor) {
        this.state = 'pending'; // Initial state
        this.value = undefined;  // Value when fulfilled
        this.reason = undefined; // Reason when rejected
        this.onFulfilledCallbacks = []; // Queue for success callbacks
        this.onRejectedCallbacks = [];  // Queue for error callbacks
        // The resolve function
        const resolve = (value) => {
            if (this.state === 'pending') {
                this.state = 'fulfilled';
                this.value = value;
                this.onFulfilledCallbacks.forEach(callback => callback(this.value));}
        };
        // The reject function
        const reject = (reason) => {
            if (this.state === 'pending') {
                this.state = 'rejected';
                this.reason = reason;
                this.onRejectedCallbacks.forEach(callback => callback(this.reason));
            }
        };

        // Execute the executor function
        try {
            executor(resolve, reject);
        } catch (error) {
            reject(error);
        }
    }



    // Method to handle fulfillment
    then(onFulfilled, onRejected) {
        return new MyPromise((resolve, reject) => {
            const handleFulfilled = () => {
                try {
                    const result = onFulfilled(this.value);
                    resolve(result);//why are we calling a resolve again on result

                    //well suppose if
                    /*
                    const promise1 = new MyPromise((resolve, reject) => {
                    setTimeout(() => resolve("First promise resolved!"), 1000);
                   });

                promise1
                 .then((result) => {
                    console.log(result); // Output: First promise resolved!
                    return "Second promise resolved!"; // Returning a value here we are
                    //returning a value which will be store in result:
                        [const result = onFulfilled(this.value);]//this will catch the return value:

                })

                .then((result) => {console.log(result); // Output: Second promise resolved!});

                //and this second function will handle the return promise returned from the first
                //.then()

                that 's why here:
                try {
                    const result = onFulfilled(this.value);
                    resolve(result);//we called a resolve on the return .then() function
                    //so we can chain it with other .then()
                */

                } catch (error) {
                    reject(error);
                }
            };

            const handleRejected = () => {
                try {
                    if (onRejected) {
                        const result = onRejected(this.reason);
                        resolve(result);
                    } else {
                        reject(this.reason);
                    }
                } catch (error) {
                    reject(error);
                }
            };


            if (this.state === 'fulfilled') {
                handleFulfilled();
            } else if (this.state === 'rejected') {
                console.log("Rejected");
                handleRejected();
            } else {
                this.onFulfilledCallbacks.push(handleFulfilled);
                this.onRejectedCallbacks.push("Rejected");
            }
        });
    }

    // Method to handle rejection
    catch(onRejected) {
        return this.then(null, onRejected);
    }
}


const promise1 = new MyPromise((resolve, reject) => {
    setTimeout(() => resolve("First promise resolved!"), 1000);
});

promise1
    .then((result) => {
        console.log(result); // Output: First promise resolved!
        return "Second promise resolved!"; // Returning a value
    })
    .then((result) => {
        console.log(result); // Output: Second promise resolved!
    });
console.log("hello world");
```

```
class MyPromise {
    constructor(executor) {
        this.state = 'pending'; // Initial state
        this.value = undefined;  // Value when fulfilled
        this.reason = undefined; // Reason when rejected
        this.onFulfilledCallbacks = []; // Queue for success callbacks
        this.onRejectedCallbacks = [];  // Queue for error callbacks

        // The resolve function
        const resolve = (value) => {
            if (this.state === 'pending') {
                this.state = 'fulfilled';
                this.value = value;
                this.onFulfilledCallbacks.forEach(callback => callback(this.value));
            }
        };

        // The reject function
        const reject = (reason) => {
            if (this.state === 'pending') {
                this.state = 'rejected';
                this.reason = reason;
                this.onRejectedCallbacks.forEach(callback => callback(this.reason));
            }
        };

        // Execute the executor function
        try {
            executor(resolve, reject);
        } catch (error) {
            reject(error);
        }
    }
```

This will call create the instance of Promise Class instance

```
const promise1 = new MyPromise((resolve, reject) => {
    setTimeout(() => resolve("First promise resolved!"), 1000);
});
```

```
(resolve, reject) => {
        setTimeout(() => {
            resolve("First promise resolved!")
        }, 1000);
    }
```

this annonymous function will map it with the executor function

```
executor:((resolve,reject)=>{
    setTimeout(()=>{resolve("First promise resolved!")},1000);
})(resolve,reject);//this is a iffe
```

```
    // Method to handle fulfillment
    then(onFulfilled, onRejected) {
        return new MyPromise((resolve, reject) => {
            const handleFulfilled = () => {
                try {
                    const result = onFulfilled(this.value);
                    resolve(result);
                } catch (error) {
                    reject(error);
                }
            };

            const handleRejected = () => {
                try {
                    if (onRejected) {
                        const result = onRejected(this.reason);
                        resolve(result);
                    } else {
                        reject(this.reason);
                    }
                } catch (error) {
                    reject(error);
                }
            };

            if (this.state === 'fulfilled') {
                handleFulfilled();
            } else if (this.state === 'rejected') {
                handleRejected();
            } else {
                this.onFulfilledCallbacks.push(handleFulfilled);
                this.onRejectedCallbacks.push(handleRejected);
            }
        });
    }

    // Method to handle rejection
    catch(onRejected) {
        return this.then(null, onRejected);
    }
}
```

will call
.then()
function

```
const promise1 = new MyPromise((resolve, reject) => {
    setTimeout(() => resolve("First promise resolved!"), 1000);
});

promise1
    .then((result) => {
        console.log(result); // Output: First promise resolved!
        return "Second promise resolved!"; // Returning a value
    })
    .then((result) => {
        console.log(result); // Output: Second promise resolved!
    });
```

```
    // Method to handle fulfillment
    then(onFulfilled, onRejected) {
        return new MyPromise((resolve, reject) => {
            const handleFulfilled = () => {
                try {
                    const result = onFulfilled(this.value);
                    resolve(result);
                } catch (error) {
                    reject(error);
                }
            };

            const handleRejected = () => {
                try {
                    if (onRejected) {
                        const result = onRejected(this.reason);
                        resolve(result);
                    } else {
                        reject(this.reason);
                    }
                } catch (error) {
                    reject(error);
                }
            };

            if (this.state === 'fulfilled') {
                handleFulfilled();
            } else if (this.state === 'rejected') {
                handleRejected();
            } else {
                this.onFulfilledCallbacks.push(handleFulfilled);
                this.onRejectedCallbacks.push(handleRejected);
            }
        });
    }

    // Method to handle rejection
    catch(onRejected) {
        return this.then(null, onRejected);
    }
}
```

```
const promise1 = new MyPromise((resolve, reject) => {
    setTimeout(() => resolve("First promise resolved!"), 1000);
});

promise1
    .then((result) => {
        console.log(result); // Output: First promise resolved!
        return "Second promise resolved!"; // Returning a value
    })
    .then((result) => {
        console.log(result); // Output: Second promise resolved!
    });
```

will return the data to be catch by result and apply the resolve function to it and data by processing it

why so that the next .then() function can handle that resolve value