JavaScript Interview Questions:-

1.How to use await outside of async function prior to ES2022?

This approach involves defining an **`async`** IIFE (Immediately Invoked Function Expression) and using **`await`** inside it. This allows you to simulate the behavior of **`await`** outside of a traditional **`async`** function by immediately executing the asynchronous code inside an async context.

Remember, this is a workaround and not a direct usage of **`await`** outside of an **`async`** function, as supported from ES2022 onwards.

function delay(ms) { return new Promise(resolve => { setTimeout(resolve, ms); }); } (async function() { console.log('Before'); await delay(2000); // Using await inside an async IIFE console.log('After 2 seconds'); })();

2.What is Function Composition in Javascript?

Function composition in JavaScript refers to combining multiple functions to create a new function. It's a technique where the output of one function becomes the input of another function, allowing you to create complex operations by chaining smaller functions together

// Example functions const add = x => x + 5; const multiply = x => x * 2; // Function composition const composedFunction = x => multiply(add(x)); // Usage const result = composedFunction(3); // Result: (3 + 5) * 2 = 16

3.What is module pattern in js?

The Module Pattern in JavaScript is a design pattern used to encapsulate private and public members within a module, providing a way to organize and structure code while keeping certain variables and functions private (accessible only within the module) and exposing selected functionalities or data as public (accessible from outside the module).

(function () { // Private variables or functions goes here. return { // Return public variables or functions here. }; })();

4.What are compose and pipe functions?

Both **`compose`** and **`pipe`** are useful for creating a pipeline of functions, allowing for more readable and maintainable code by breaking down complex operations into smaller, composable functions. The key difference lies in the order in which they execute functions: **`compose`** processes functions from right to left, while **`pipe`** processes functions from left to right.

5.What are the possible side-effects in javascript?

A side effect is the modification of the state through the invocation of a function or expression. These side effects make our function impure by default. Below are some side effects which make function impure,

- Making an HTTP request. Asynchronous functions such as fetch and promise are impure.
- DOM manipulations
- Mutating the input data
- Printing to a screen or console: For example, console.log() and alert()
- Using **`setTimeout`** or **`setInterval`**
- Fetching the current time
- Exception Throwing d

6)What is referential transparency?

An expression in javascript that can be replaced by its value without affecting the behaviour of the program is called referential transparency. Pure functions are referentially transparent.

```
const add = (x, y) => x + y;
const multiplyBy2 = (x) => x * 2;

//Now add (2, 3) can be replaced by 5.

multiplyBy2(add(2, 3));
```

7)What are the differences between pure and impure functions?

Pure Functions	Impure Functions
Always returns the same output for the same input	Can produce different results for the same input

Pure Functions	Impure Functions
No side effects (no changes to external state)	May cause side effects by modifying external state
Relies only on arguments passed to the function	May rely on external variables or global state
Doesn't modify its arguments or any external data	Can modify its arguments or external data
Easy to test and reason about (deterministic)	Difficult to test and reason about (non-deterministic)

8)What are the differences between primitives and non-primitives?

Primitives	Non-Primitives (Reference Type
Immutable	Mutable
Stored directly in memory	Stored by reference in memory
Copied by value	Copied by reference
Examples:	Examples:
- String	- Objects
- Number	- Arrays
- Boolean	- Functions
- Undefined	- Objects created with **`new`** keyword
- Null	- Dates

9)What is pass by value and pass by reference?

Pass-by-value creates a new space in memory and makes a copy of a value. Primitives such as string, number, boolean etc will actually create a new copy. Hence, updating one value doesn't impact the other value. i.e, The values are independent of each other.

EX:let a = 5;

let b = a;

b++;

console.log(a, b); //5, 6

Pass by reference doesn't create a new space in memory but the new variable adopts a memory address of an initial variable. Non-primitives such as objects, arrays and functions gets the reference of the initiable variable. i.e, updating one value will impact the other variable.

EX:let user1 = {

  name: "John",

  age: 27,

};

let user2 = user1;

user2.age = 30;

console.log(user1.age, user2.age); // 30, 30

10)How to verify if a variable is an array?

It is possible to check if a variable is an array instance using 3 different ways,

i. Array.isArray() method:

   The `Array.isArray(value)` utility function is used to determine whether value is an array or not. This function returns a true boolean value if the variable is an array and a false value if it is not.

```
const numbers = [1, 2, 3];
const user = { name: "John" };
Array.isArray(numbers); // true
Array.isArray(user); //false
```

ii. instanceof operator:

   The instanceof operator is used to check the type of an array at run time. It returns true if the type of a variable is an Array other false for other type.

```
const numbers = [1, 2, 3];
const user = { name: "John" };
console.log(numbers instanceof Array); // true
console.log(user instanceof Array); // false
```

iii. Checking constructor type:

   The constructor property of the variable is used to determine whether the variable Array type or not.

```
const numbers = [1, 2, 3];
const user = { name: "John" };
console.log(numbers.constructor === Array); // true
console.log(user.constructor === Array); // false
```

11. What is an environment record?

Manage Identifiers: Environment records handle the connections between identifiers (like variable names) and their values within execution contexts.
Types of Records: Three main types include Object Environment Record, Declarative Environment Record, and Function Environment Record.
Purposeful Assignments: They serve different purposes, like mapping identifiers to object properties and managing variable declarations within lexical scopes.
Variable Resolution: Crucial for variable lookup and resolution in JavaScript code during execution.
Scope Management: Essential for defining the accessibility and lifetime of variables, aiding in scoping and closure mechanisms within the language.
12.What is optional chaining in js?

Optional chaining in JavaScript is a feature that allows you to access deeply nested properties of an object without explicitly checking if each level exists. It helps to avoid errors like **`TypeError: Cannot read property 'x' of undefined`** or **`Cannot read property 'y' of null`** when trying to access properties of objects that may be undefined or null at some level of nesting.

The optional chaining operator **`?.`** can be used to safely access nested properties or methods without causing an error if an intermediate property is null or undefined. If the property or method exists, it will be accessed; otherwise, it returns **`undefined`**.

// Object with nested properties const user = { id: 1, name: 'John', address: { street: '123 Main St', city: 'Example City', zipCode: '12345' }, // comment: 'This is a comment' // Try uncommenting this line to see the effect }; // Accessing nested properties with optional chaining const zipCode = user.address?.zipCode; // Safely access 'zipCode' property console.log(zipCode); // Output: '12345' if 'address' and 'zipCode' exist, otherwise 'undefined'

In this example:

`user.address?.zipCode` tries to access the **`zipCode`** property within the **`address`** object of the **`user`**. If **`address`** is **`undefined`** or **`null`**, it short-circuits the access and returns **`undefined`** without throwing an error.

13)How do you reverse an array without modifying original array?

There are few solutions that won't mutate the original array. Let's take a look.

i. Using slice and reverse methods: In this case, just invoke the `slice()` method on the array to create a shallow copy followed by `reverse()` method call on the copy.

```
const originalArray = [1, 2, 3, 4, 5];
const newArray = originalArray.slice().reverse(); //Slice an array gives a
new copy

console.log(originalArray); // [1, 2, 3, 4, 5]
console.log(newArray); // [ 5, 4, 3, 2, 1]
```

ii. Using spread and reverse methods: In this case, let's use the spread syntax (...) to create a copy of the array followed by `reverse()` method call on the copy.

```
const originalArray = [1, 2, 3, 4, 5];
const newArray = [...originalArray].reverse();

console.log(originalArray); // [1, 2, 3, 4, 5]
console.log(newArray); // [ 5, 4, 3, 2, 1]
```

iii. Using reduce and spread methods: Here execute a reducer function on an array elements and append the accumulated array on right side using spread syntax

```
const originalArray = [1, 2, 3, 4, 5];
const newArray = originalArray.reduce((accumulator, value) => {
  return [value, ...accumulator];
}, []);

console.log(originalArray); // [1, 2, 3, 4, 5]
console.log(newArray); // [ 5, 4, 3, 2, 1]
```

iv. Using reduceRight and spread methods: Here execute a right reducer function(i.e. opposite direction of reduce method) on an array elements and append the accumulated array on left side using spread syntax

```
const originalArray = [1, 2, 3, 4, 5];
const newArray = originalArray.reduceRight((accumulator, value) => {
  return [...accumulator, value];
}, []);

console.log(originalArray); // [1, 2, 3, 4, 5]
console.log(newArray); // [ 5, 4, 3, 2, 1]
```

14.What is the difference between setTimeout, setImmediate and process.nextTick?

Feature	**`setTimeout`**	**`setImmediate`**	**`process.nextTick`**
Execution Timing	Executes after a specified delay (minimum)	Executes in the next iteration of the event loop	Executes before the next iteration of the event loop

15What are the different ways to create sparse arrays?

There are 4 different ways to create sparse arrays in JavaScript

i. Array literal: Omit a value when using the array literal

```
const justiceLeague = ["Superman", "Aquaman", , "Batman"];
console.log(justiceLeague); // ['Superman', 'Aquaman', empty
,'Batman']
```

ii. Array() constructor: Invoking Array(length) or new Array(length)

```
const array = Array(3);
console.log(array); // [empty, empty ,empty]
```

iii. Delete operator: Using delete array[index] operator on the array

```
const justiceLeague = ["Superman", "Aquaman", "Batman"];
delete justiceLeague[1];
console.log(justiceLeague); // ['Superman', empty, ,'Batman']
```

iv. Increase length property: Increasing length property of an array

```
const justiceLeague = ["Superman", "Aquaman", "Batman"];
justiceLeague.length = 5;
console.log(justiceLeague); // ['Superman', 'Aquaman', 'Batman',
empty, empty]
```

16.What is the difference between dense and sparse arrays?

An array contains items at each index starting from first(0) to last(array.length - 1) is called as Dense array. Whereas if at least one item is missing at any index, the array is called as sparse.

Let's see the below two kind of arrays,

```
const avengers = ["Ironman", "Hulk", "CaptainAmerica"];
console.log(avengers[0]); // 'Ironman'
console.log(avengers[1]); // 'Hulk'
console.log(avengers[2]); // 'CaptainAmerica'
console.log(avengers.length); // 3

const justiceLeague = ["Superman", "Aquaman", , "Batman"];
console.log(justiceLeague[0]); // 'Superman'
console.log(justiceLeague[1]); // 'Aquaman'
console.log(justiceLeague[2]); // undefined
console.log(justiceLeague[3]); // 'Batman'
console.log(justiceLeague.length); // 4
```

17.How do you group and nest console output?

The `console.group()` can be used to group related log messages to be able to easily read the logs and use console.groupEnd()to close the group. Along with this, you can also nest groups which allows to output message in hierarchical manner.

For example, if you’re logging a user’s details:

```
console.group("User Details");
console.log("name: Sudheer Jonna");
console.log("job: Software Developer");

// Nested Group
console.group("Address");
console.log("Street: Commonwealth");
console.log("City: Los Angeles");
console.log("State: California");

// Close nested group
console.groupEnd();

// Close outer group
console.groupEnd();
```

18.What is nullish coalescing operator (??)?

It is a logical operator that returns its right-hand side operand when its left-hand side operand is null or undefined, and otherwise returns its left-hand side operand. This can be contrasted with the logical OR (||) operator, which returns the right-hand side operand if the left operand is any falsy value, not only null or undefined.

```
console.log(null ?? true); // true
console.log(false ?? true); // false
console.log(undefined ?? true); // true
```

19.How do style the console output using CSS?

You can add CSS styling to the console output using the CSS format content specifier %c. The console string message can be appended after the specifier and CSS style in another argument. Let's print the red the color text using console.log and CSS specifier as below,

```
console.log("%cThis is a red text", "color:red");
```

It is also possible to add more styles for the content. For example, the font-size can be modified for the above text

```
console.log(
  "%cThis is a red text with bigger font",
  "color:red; font-size:20px"
);
```

20.What is the easiest way to ignore promise errors?

The easiest and safest way to ignore promise errors is void that error. This approach is ESLint friendly too.

```
await promise.catch((e) => void e);
```

```
21.Is that possible to use expressions in switch cases?
```

You might have seen expressions used in switch condition but it is also possible to use for switch cases by assigning true value for the switch condition. Let's see the weather condition based on temparature as an example,

```
const weather = (function getWeather(temp) {
  switch (true) {
    case temp < 0:
      return "freezing";
    case temp < 10:
      return "cold";
    case temp < 24:
      return "cool";
    default:
      return "unknown";
  }
})(10);
```

22.How to invoke an IIFE without any extra brackets?

An IIFE (Immediately Invoked Function Expression) in JavaScript is typically invoked by wrapping the function declaration in parentheses and immediately calling it with an additional set of parentheses.

However, there's a way to invoke an IIFE without using any extra brackets by making it part of a larger expression or statement.

One common approach is to use the grouping operator **`( )`** to surround the entire function expression, which allows its immediate invocation without additional parentheses.

```
const result = function() {
  return 'IIFE invoked without extra brackets';
}();
```

23.What is the difference between isNaN and Number.isNaN?

i. isNaN: The global function `isNaN` converts the argument to a Number and returns true if the resulting value is NaN.
ii. Number.isNaN: This method does not convert the argument. But it returns true when the type is a Number and value is NaN.

Let's see the difference with an example,

```
isNaN(‘hello’);   // true
Number.isNaN('hello'); // false
```

24.What are the differences between for...of and for...in statements

Both for...in and for...of statements iterate over js data structures. The only difference is over what they iterate:

i. for..in iterates over all enumerable property keys of an object
ii. for..of iterates over the values of an iterable object.

Let's explain this difference with an example,

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

arr.newProp = "newVlue";

// key are the property keys
for (let key in arr) {
  console.log(key); // 0, 1, 2 & newValue
}

// value are the property values
for (let value of arr) {
  console.log(value); // a, b, c
}
```

Since for..in loop iterates over the keys of the object, the first loop logs 0, 1, 2 and newProp while iterating over the array object. The for..of loop iterates over the values of a arr data structure and logs a, b, c in the console.

25.What are the built-in iterables

Below are the list of built-in iterables in javascript,

i. Arrays and TypedArrays
ii. Strings: Iterate over each character or Unicode code-points
iii. Maps: iterate over its key-value pairs
iv. Sets: iterates over their elements

v. arguments: An array-like special variable in functions
vi. DOM collection such as NodeList
vii. Generators

26.What are the differences between spread operator and rest parameter

Rest parameter collects all remaining elements into an array. Whereas Spread operator allows iterables(arrays / objects / strings) to be expanded into single arguments/elements. i.e, Rest parameter is opposite to the spread operator.

27.What are the differences between arguments object and rest parameter

There are three main differences between arguments object and rest parameters

i. The arguments object is an array-like but not an array. Whereas the rest parameters are array instances.
ii. The arguments object does not support methods such as sort, map, forEach, or pop. Whereas these methods can be used in rest parameters.
iii. The rest parameters are only the ones that haven't been given a separate name, while the arguments object contains all arguments passed to the function

28.**What is deno**

Deno is a simple, modern and secure runtime for JavaScript and TypeScript that uses V8 JavaScript engine and the Rust programming language.

29.How do you prevent promises swallowing errors in js?

To prevent promises from silently swallowing errors in JavaScript, it's essential to handle promise rejections explicitly by utilizing **`catch`** blocks or **`try...catch`** statements. This ensures that errors within promises are caught and appropriately dealt with, preventing them from being unnoticed.

30.What is an async function

An async function is a function declared with the `async` keyword which enables asynchronous, promise-based behavior to be written in a cleaner style by avoiding promise chains. These functions can contain zero or more `await` expressions.

Let's take a below async function example,

```
async function logger() {
  let data = await fetch("http://someapi.com/users"); // pause
until fetch returns
  console.log(data);
}
logger();
```

31.What is the difference between function and class declarations

The main difference between function declarations and class declarations is `hoisting`. The function declarations are hoisted but not class declarations.

Classes:

```
const user = new User(); // ReferenceError

class User {}
```

Constructor Function:

```
const user = new User(); // No error

function User() {}
```

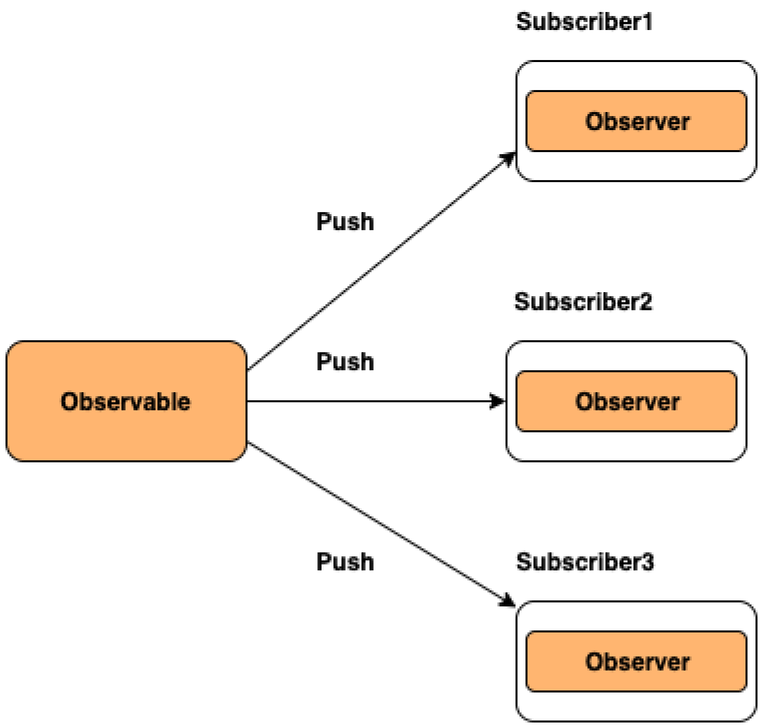
32.What is an observable

An Observable is basically a function that can return a stream of values either synchronously or asynchronously to an observer over time. The consumer can get the value by calling `subscribe()` method. Let's look at a simple example of an Observable

```
import { Observable } from "rxjs";

const observable = new Observable((observer) => {
  setTimeout(() => {
    observer.next("Message from a Observable!");
  }, 3000);
});

observable.subscribe((value) => console.log(value));
```



33.What is the easiest way to resize an array

The length property of an array is useful to resize or empty an array quickly. Let's apply length property on number array to resize the number of elements from 5 to 2,

```
var array = [1, 2, 3, 4, 5];
console.log(array.length); // 5

array.length = 2;
console.log(array.length); // 2
console.log(array); // [1,2]
```

and the array can be emptied too

```
var array = [1, 2, 3, 4, 5];
```

```
array.length = 0;
console.log(array.length); // 0
console.log(array); // []
```

34.What is a Short circuit condition

Short circuit conditions are meant for condensed way of writing simple if statements. Let's demonstrate the scenario using an example. If you would like to login to a portal with an authentication condition, the expression would be as below,

```
if (authenticate) {
  loginToPorta();
}
```

Since the javascript logical operators evaluated from left to right, the above expression can be simplified using && logical operator

```
authenticate && loginToPorta();
```

35.What is RxJS

RxJS (Reactive Extensions for JavaScript) is a library for implementing reactive programming using observables that makes it easier to compose asynchronous or callback-based code. It also provides utility functions for creating and working with observables.

36.What are the common use cases of observables

Some of the most common use cases of observables are web sockets with push notifications, user input changes, repeating intervals, etc

37.Is Node.js completely single threaded

Node is a single thread, but some of the functions included in the Node.js standard library(e.g, fs module functions) are not single threaded. i.e, Their logic runs outside of the Node.js single thread to improve the speed and performance of a program.

38.What is babel

Babel is a popular JavaScript compiler that allows developers to write code using the latest ECMAScript (ES) syntax and features, and then transpile (convert) it into older versions of JavaScript that can run in older browsers or environments that do not support the latest features.

39.How do you detect primitive or non primitive value type

In JavaScript, primitive types include boolean, string, number, BigInt, null, Symbol and undefined. Whereas non-primitive types include the Objects. But you can easily identify them with the below function,

```
var myPrimitive = 30;
var myNonPrimitive = {};
function isPrimitive(val) {
  return Object(val) !== val;
}

isPrimitive(myPrimitive);
isPrimitive(myNonPrimitive);
```

If the value is a primitive data type, the Object constructor creates a new wrapper object for the value. But If the value is a non-primitive data type (an object), the Object constructor will give the same object.

40.What is the difference between shim and polyfill?

Feature	Shim	Polyfill
Purpose	Fills gaps in functionality	Provides entire missing functionality

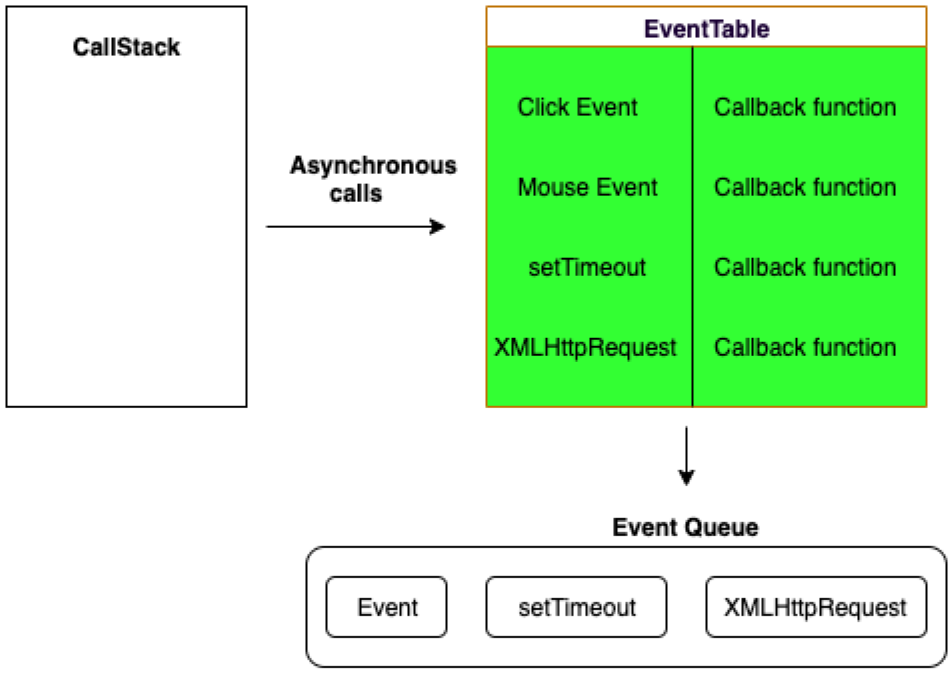
Feature	Shim	Polyfill
Functionality	Provides partial implementation	Offers complete implementation of missing features
Targeted Use	Specific to a particular feature or API	Targeted for a broader range of missing functionalities
Compatibility	Addresses specific issues or APIs	Focuses on overall browser compatibility
Usage	Primarily for newer browser inconsistencies	Mainly for supporting newer features in older browsers

41.What is a microTask queue

In JavaScript, the microtask queue (also known as the microtask queue) is a task queue that handles the execution of microtasks. Microtasks are tasks that need to be executed asynchronously and are scheduled to run after the current task has finished but before the browser renders the UI. They have higher priority than macrotasks (such as callbacks from I/O operations, timeouts, or rendering) and are guaranteed to execute before the next rendering. `But if these microtasks are running for a long time then it leads to visual degradation.`

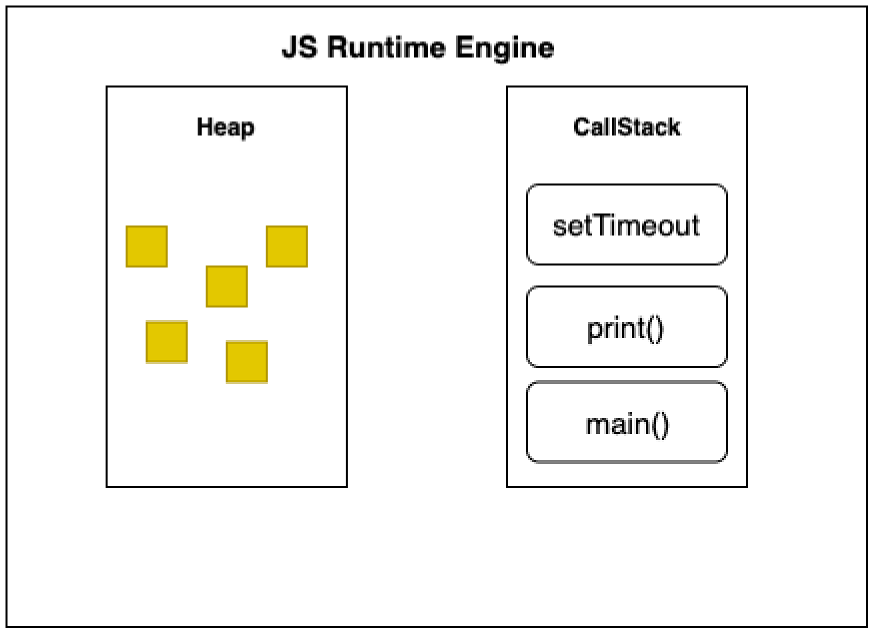
42.What is an event table

Event Table is a data structure that stores and keeps track of all the events which will be executed asynchronously like after some time interval or after the resolution of some API requests. i.e Whenever you call a setTimeout function or invoke async operation, it is added to the Event Table. It doesn't not execute functions on it’s own. The main purpose of the event table is to keep track of events and send them to the Event Queue as shown in the below diagram.



43.What is heap

Heap(Or memory heap) is the memory location where objects are stored when we define variables. i.e, This is the place where all the memory allocations and de-allocation take place. Both heap and call-stack are two containers of JS runtime. Whenever runtime comes across variables and function declarations in the code it stores them in the Heap.



44.What are the differences between promises and observables

Some of the major difference in a tabular form

Promises	Observables
Emits only a single value at a time	Emits multiple values over a period of time(stream of values ranging from 0 to multiple)
Eager in nature; they are going to be called immediately	Lazy in nature; they require subscription to be invoked
Promise is always asynchronous even though it resolved immediately	Observable can be either synchronous or asynchronous
Doesn't provide any operators	Provides operators such as map, forEach, filter, reduce, retry, and retryWhen etc
Cannot be canceled	Canceled by using unsubscribe() method

45.How do you use javascript libraries in typescript file

It is known that not all JavaScript libraries or frameworks have TypeScript declaration files. But if you still want to use libraries or frameworks in our TypeScript files without getting compilation errors, the only solution is `declare` keyword along with a variable declaration. For example, let's imagine you have a library called `customLibrary` that doesn’t have a TypeScript declaration and have a namespace called `customLibrary` in the global namespace. You can use this library in typescript code as below,

```
declare var customLibrary;
```

In the runtime, typescript will provide the type to the `customLibrary` variable as `any` type. The another alternative without using declare keyword is below

```
var customLibrary: any;
```

46.What is the purpose of queueMicrotask

The `queueMicrotask` function is used to schedule a microtask, which is a function that will be executed asynchronously in the microtask queue. The purpose of `queueMicrotask` is to ensure that a function is executed after the current task has finished, but before the browser performs any rendering or handles user events.

Example:

```
console.log("Start"); //1

queueMicrotask(() => {
  console.log("Inside microtask"); // 3
});

console.log("End"); //2
```

By using queueMicrotask, you can ensure that certain tasks or callbacks are executed at the earliest opportunity during the JavaScript event loop, making it useful for performing work that needs to be done asynchronously but with higher priority than regular `setTimeout` or `setInterval` callbacks.

47.What are different event loops

```
In JavaScript, there are multiple event loops that can be used
depending on the context of your application. The most common
event loops are:
The Browser Event Loop
The Node.js Event Loop


    - Browser Event Loop: The Browser Event Loop is used in client-
    side JavaScript applications and is responsible for handling
    events that occur within the browser environment, such as user
    interactions (clicks, keypresses, etc.), HTTP requests, and
    other asynchronous actions.


    - The Node.js Event Loop is used in server-side JavaScript
    applications and is responsible for handling events that occur
    within the Node.js runtime environment, such as file I/O,
    network I/O, and other asynchronous actions.
```

48.What is microtask

Microtask is the javascript code which needs to be executed immediately after the currently executing task/microtask is completed. They are kind of blocking in nature. i.e, The main thread will be blocked until the microtask queue is empty. The main sources of microtasks are Promise.resolve, Promise.reject, MutationObservers, IntersectionObservers etc

Note: All of these microtasks are processed in the same turn of the event loop

49.What are tasks in event loop

A task is any javascript code/program which is scheduled to be run by the standard mechanisms such as initially starting to run a program, run an event callback, or an interval or timeout being fired. All these tasks are scheduled on a task queue. Below are the list of use cases to add tasks to the task queue,

i. When a new javascript program is executed directly from console or running by the `<script>` element, the task will be added to the task queue.
ii. When an event fires, the event callback added to task queue
iii. When a setTimeout or setInterval is reached, the corresponding callback added to task queue

50.How do you implement zero timeout in modern browsers

You can't use setTimeout(fn, 0) to execute the code immediately due to minimum delay of greater than 0ms. But you can use window.postMessage() to achieve this behavior.

20.What is the easiest way to ignore promise errors?

The easiest and safest way to ignore promise errors is void that error. This approach is ESLint friendly too.

```
await promise.catch((e) => void e);
```