What is JAVASCRIPT..?

JavaScript is a **programming language used for creating dynamic content on websites**. It is a **lightweight**, **cross-platform** and **single-threaded** programming language. JavaScript is an **interpreted**language that executes code line by line providing more flexibility. It is a commonly used programming language to**create dynamic and interactive elements in web applications**.

**\*\*NOTE:** Everything in JavaScript happens inside an “**Execution Context”.**

JavaScript is a “**Synchronous single-threaded”** language .

Whenever JavaScript program runs a **Global Execution Context** get Created.

**Memory Execution Code Execution Variable Enviroment Thread of Execution**

All Variables and Thread of

Fuctions are store in Execution is like a **“KEY : VALUE” Pair.** thread in which the

whole code is

Executed

**Line by Line.**

The JavaScript code is being executed to see a more clear picture. Everything in JavaScript happens inside the execution context. Execution context is the little separate section where code is being executed and variables get their memory space.  
The JavaScript code is being executed in Two Phases:

1. The first one is the memory allocation phase during this all the variables and function definitions get stored inside the memory heap. The JavaScript assigns undefined to each variable in this phase.
2. The second one is a thread of the execution phase, during this the code written inside the JavaScript file is being executed.  
   Each variable holds the value *undefined*till the program reaches the line where we have assigned that variable. After that line, the variable’s undefined value gets replaced by the original value.

* **NOTE: Call Stack** maintains the Order of Execution of “**Execution Context”.**

**Call Stack** is sometimes Known As **:**

1. Execution Context Stack.
2. Program Stack.
3. Control Stack.
4. Runtime Stack.
5. Machine Stack.

## JavaScript Output Display Properties:

There are primarily four distinct methods for displaying output in JavaScript.

* **innerHTML –** Display the HTML output on an element.
* **Using console.log() Method –** Display the content on the browser console.
* **document.write() method –** Displays the content on an HTML element.
* **window.alert() Method –** Display the HTML element to the output.

**\*\*\* Hoisting in JavaScript: JavaScript hoisting** is a mechanism where variable and function declarations are moved to the top of their containing scope during the compilation phase. JavaScript hoisting is a important concept for understanding how variables and functions are processed during code execution

**Note:**  JavaScript only hoists declarations, not initializations.

**Features of Hoisting:**

* Declarations are hoisted, not initializations.
* Allows calling functions before their declarations.
* All variable and function declarations are processed before any code execution.
* Undeclared variables are implicitly created as global variables when assigned a value.

## ****Sequence of variable declaration****:

***Declaration –> Initialisation/Assignment –> Usage***

* Synchronous Execution:

In synchronous execution, code is executed sequentially, one line at a time. Each operation must complete before the next one starts. This can lead to blocking behavior, where time-consuming operations (e.g., network requests, file I/O) can block the execution of subsequent code until they complete.

* ASynchronous Execution:

Asynchronous execution allows code to run without blocking the execution of subsequent operations. Time-consuming tasks can be executed in the background, allowing other code to run concurrently. This is particularly useful for operations like network requests, timers, and file I/O.

JavaScript uses mechanisms like callbacks, promises, and async/await to handle asynchronous operations.

* DataType in JAVASCRIPT:

**JavaScript has seven data types are Primitive (predefined)  data types:**

**1. Primitive Data Types:**

o String: Represents textual data. Enclosed within single (''), double ("") or backticks (``) quotes.

o Number: Represents numeric values, including integers and floating-point numbers.

o Boolean: Represents a logical value, either true or false.

o Null: Represents the intentional absence of any object value.

o Undefined: Represents the uninitialized state of a variable.

o Symbol: Represents a unique identifier.

o BigInt: BigInt is a built-in object in JavaScript that provides a way to represent whole numbers larger than 2^53-1.

**2. Non-primitive Data Types (or Reference Types):**

o Object: Represents a collection of key-value pairs (properties and methods).

o Array: Represents a list-like collection of elements, accessible by numerical indices.

o Function: Represents a reusable block of code that can be called with different arguments.

JavaScript is a **dynamically typed** (also called loosely typed) scripting language, meaning you don't have to specify the data type of a variable when declaring it; the interpreter determines the data type based on the assigned value.

* Variables in JAVASCRIPT:

Variables are used to store data values. These values can vary over time, hence the term "variable." Variables in JavaScript can hold different types of data, and these data types determine what kind of operations can be performed on them and how they are stored in memory.

**there are three keywords used for declaring variables:** let, var, and const.

## ****Block Scope:****

The variables which are declared inside the { } block are known as block-scoped variables. variables declared by the var keyword cannot be block-scoped.

**1. Let:**

* Variables declared with let have block scope, meaning they are only accessible within the block in which they are defined (enclosed within curly braces {}).
* They can be reassigned values.
* Variables declared with let are not hoisted to the top of their containing block. This means they are not accessible before the line they are declared on.

**2. Var:**

* Variables declared with var have function scope or global scope if declared outside of any function.
* They can be reassigned values.
* Variables declared with var are hoisted to the top of their containing function or script, which means they can be accessed before the line they are declared on, but their value will be undefined until they are initialized.

**3. Const:**

* Variables declared with const have block scope like let.
* The value of a variable declared with const cannot be reassigned once it has been initialized.
* Variables declared with const must be assigned a value at the time of declaration. Trying to declare a const variable without initializing it will result in a syntax error.
* Functions in JAVASCRIPT:

JavaScript functions are essential for organizing code and executing specific tasks. They contain sets of instructions that run when triggered by events or actions. Functions are fundamental building blocks in JavaScript, enabling code reuse, modularity, and abstraction.

"A function in JavaScript is a named block of code that can be defined and invoked to perform a specific task or calculation. Functions can take input parameters, execute a sequence of statements, and optionally return a value."

Functions in JavaScript can be defined using the **function** keyword followed by a name, a list of parameters (optional), and a block of statements enclosed within curly braces {}. They can be invoked (called) using their name followed by parentheses ().

// Function definition

function greet(name) {

console.log("Hello, " + name + "!");

}

// Function invocation

greet("Alice"); // Output: Hello, Alice!

* Undefined or not defined in JAVASCRIPT:

"undefined" and "not defined" are two different values in JavaScript. "undefined" indicates that a variable has been declared but not given a value, while "not defined" indicates that a variable does not exist.

**Undefined:**It is a JavaScript keyword that has a special meaning. Everything which gets a space in memory will contain undefined until we assign a value to that memory space.

**Not defined:**In JavaScript, it is one of the reference errors that JavaScript will throw when someone accesses the variable which is not inside the memory heap.

Lexical Environment : A lexical environment refers to the context in which a piece of JavaScript code is executed. It consists of two main components: the environment record and a reference to the outer lexical environment.

*“A lexical environment in JavaScript is a data structure that stores the variables and functions that are defined in the current scope and all of the outer scopes. It is also known as the lexical scope or the lexical closure.”*

1. **Environment Record**: This is where the variables and functions defined in the current scope are stored. It essentially acts as a map of identifiers to their corresponding values.
2. **Outer Lexical Environment Reference**: This is a reference to the lexical environment in which the current code is nested. This reference allows JavaScript to follow the scope chain, looking up variables and functions in outer scopes if they're not found in the current one.

**Conclusion:**In JavaScript, the lexical scope is used to determine the accessibility of variables and functions within a program. Understanding lexical scope is crucial for writing clean, organized, and maintainable code. By properly scoping variables and functions, we can avoid naming conflicts, improve code readability, and reduce the risk of unintended side effects.

It’s important to note that JavaScript also has dynamic scope, which is determined at runtime rather than at compile time. This means that the scope of a variable can change depending on how it is called or invoked.

* Scope and Scope Chain in JavaScript:

**Scope**

* Scope in JavaScript determines the accessibility of variables and functions at various parts of one’s code or program.
* In other words, Scope will help us to determine a given part of a code or a program, what variables or functions one can access, and what variables or functions one cannot access.
* Within a scope itself, a variable a function, or a method could be accessed. Outside the specified scope of a variable or function, the data cannot be accessed.
* There are three types of scopes available in JavaScript: **Global Scope, Local / Function Scope,**and **Block Scope.**

**Scope Chain**

* JavaScript engine uses scopes to find out the exact location or accessibility of variables and that particular process is known as Scope Chain.
* Scope Chain means that one variable has a scope (it may be global or local/function or block scope) is used by another variable or function having another scope (may be global or local/function or block scope).
* This complete chain formation goes on and stops when the user wishes to stop it according to the requirement.
* Block Scope and shadowing in JS:

To understand shadowing in JavaScript, we need to be clear with the scope first. In computer programming languages, **Scope**is a certain section/region of the program where a defined variable can have its existence and can be recognized, beyond that it can’t be accessed.

* **Block** is also Known as **compound statement** that is defined by curly braces { } and used to combine multiple statements into one statement where JavaScript expects only one statement. And all the variables and functions that can be accessed inside a block are said to be inside that block scope, hence called **Block scoped**.
* For example, **let and const** variables are stored in separate memory space, so it is called **block-scoped** but var variables can be accessed outside the block as it is stored in the Global object memory space, hence it is called **Global scoped**.
* **Shadowing**: Now, when a variable is declared in a certain scope having the same name defined on its outer scope and when we call the variable from the inner scope, the value assigned to the variable in the inner scope is the value that will be stored in the variable in the memory space. This is known as **Shadowing or Variable Shadowing**. In JavaScript, the introduction of let and const in ECMAScript 6 along with block scoping allows variable shadowing.
* Block Scope has a lexical behavior and also follows rules of the Lexical Scope chain.
* Illegal shadowing: while shadowing a variable, it should not cross the boundary of the scope, i.e. we can shadow *var* variable by *let* variable but cannot do the opposite. Same as the const variable So, if we try to shadow *let*variable by *var* variable, it is known as **Illegal Shadowing** and it will give the error as*”variable is already defined.”*
* Clouser in JS:

CLOUSER is the combination of functions and the lexical scope in which they are defined. A closure allows a function to access variables from its scope, outer [function scope](https://www.geeksforgeeks.org/explore-the-concept-of-javascript-function-scope-and-different-types-of-javascript-functions/), and the global scope. This creates a “closed-over” environment, preserving the state of the outer function even after it has finished executing.

1. **Scope Chain:** In JavaScript, each function has access to its variables, as well as variables from all its ancestor scopes, creating a chain of scopes known as the “[scope chain.](https://www.geeksforgeeks.org/explain-scope-and-scope-chain-in-javascript/)“
2. **Lexical Scoping:** Closures are based on [lexical scoping](https://www.geeksforgeeks.org/lexical-scope-in-javascript/), which means that a function’s scope is determined by its position in the source code.
3. **Function Inside Function:** When a function is defined inside another function, it forms a closure. The inner function has access to the outer function’s variables, even after the outer function has completed its execution.

Closures are powerful because they allow functions to maintain access to variables from their containing scope even after the containing function has finished executing. This allows for encapsulation, data privacy, and the creation of modular and reusable code.

“Each and every Function in JavaScript has access to it’s Outer Lexical Environment(Means Functions and Variables present in environment to it’s parents)”

EXAMPLE:

innerFunction is a closure because it's defined within the scope of outerFunction. When outerFunction is called, it defines innerFunction, which captures the lexical environment of outerFunction, including the outerVariable. Even after outerFunction finishes executing, the innerFunction still has access to outerVariable because it's part of the closure.

function outerFunction() {

let outerVariable = 'I am from the outer function';

function innerFunction() {

console.log(outerVariable);

}

return innerFunction;

}

let myClosure = outerFunction();

myClosure(); // "I am from the outer function"

In below code we have ParentFunction which has one variable parentVariable and return ChildFunction only. We have one more variable executeChild refference to ChildFunction only not to ParentFunction. So when code get executed ChildFunction still have access to parentVariable which is defined outside scope in ParentFunction.

**This is called Closures.**

function ParentFunction() {

    var parentVariable = 60;

    function ChildFunction() {

        console.log(parentVariable )

    }

    return ChildFunction;

}

var executeChild = ParentFunction();

    executeChild(); //60

***#Advantages of Clouser are:***

* Closures provide a way to encapsulate variables and functions, preventing them from being accessed by other parts of the program. This helps to avoid naming conflicts and improves the overall maintainability of the code.
* Closures are very helpful to hide the implementation details in javascript. Closures can be useful to create private variables and functions.
* Closures allow us to create higher-order functions, which are functions that take other functions as arguments or return functions as values. Higher-order functions are an important feature of functional programming and can be used to write more concise and expressive code.

***#Disadvantages of Clouser are:***

* Closures consume a lot of memory. Creating a function within a function causes memory duplication, which slows down the application.
* As long as the closure are active , the memory can’t be garbage collected.
* Closures are not garbage collected when functions are internally connected.
* Garbage collection(gc):

Let us now see the general life cycle of memory management :

* On the first step whenever a variable, object or function is created, a memory space is allocated to it.
* In the next stage, the allocated memory is used by means of Read/Write operations.
* When the memory is no longer needed, release the memory space.

The last step is called the Garbage Collection mechanism. This is an automated process in JavaScript and the process is done by an entity called Garbage Collector, though we can’t physically find this agent in the browser engine.

**An overview behind garbage collection:** The majority of memory management issues occur when we try to release the allocated memory. The main concern that arises is the determination of unused memory resources. In case of the low-level languages where the developer has to manually decide when the memory is no longer needed, high-level languages such as **JavaScript** use an automated form of memory management known as **Garbage Collection(GC).**

* SETTIMEOUT FUNCTION():

Syntax: setTimeout(function, time);

Parameters:

function: The function or code snippet to be executed after the specified delay.

delay: The time, in milliseconds, to wait before executing the function.

setTimeout() method allows you to schedule the execution of a function or the evaluation of a code after a specified delay. The setTimeout() method calls a function after several milliseconds. setTimeout() is for executing a function once after a specified delay.

The setTimeout function is often used in scenarios like handling animations, managing asynchronous operations, or simulating delays. Additionally, it can be utilized in combination with closures for passing parameters to the callback function.

function sayHello() {

console.log("Hello, world!");

}

setTimeout(sayHello, 2000); // "Hello, world!" will be logged after 2 seconds.

**You can also pass an anonymous function directly to setTimeout.**

setTimeout(function() {

console.log("This message is delayed by 3 seconds.");

}, 3000);

**Canceling a Timeout:**

You can cancel a timeout before it executes using clearTimeout. The setTimeout function returns a timeout ID that you can use to cancel the timeout.

const timeoutID = setTimeout(function() {

console.log("This will not be logged.");

}, 5000);

clearTimeout(timeoutID); // Timeout is canceled, and the function will not execute

**Passing Arguments to the Function:**

You can pass additional arguments to the function to be executed.

function greet(name) {

console.log("Hello, " + name);

}

setTimeout(greet, 1000, "Alice"); // "Hello, Alice" will be logged after 1 second.

* **Function Statement (Function Declaration)**

A function statement defines a function with a specified name and can be used before it is defined in the code due to hoisting.

#### **Characteristics:**

* **Hoisting**: Function declarations are hoisted to the top of their scope. This means the function can be called before it is defined in the code.
* **Naming**: Function declarations must have a name.
* **Global Scope**: If defined in the global scope, it becomes a global function. If defined within another function or block, it is scoped to that function or block.

function first() {

    console.log("First Function Called!");

}

first();

* **Function Expression:**

A function expression defines a function as part of a larger expression syntax (typically as a variable assignment). These are not hoisted and are only available after the assignment.

#### **Characteristics:**

1. **No Hoisting**: Function expressions are not hoisted, so they cannot be called before they are defined.
2. **Anonymous or Named**: Function expressions can be anonymous (no name) or named.
3. **Assignment to Variables**: Function expressions are often assigned to variables, which can be constants (const) or variables (let or var).

var second = function () {

    console.log("Second Function Called!");

}

second();

**Difference Between Function Statement and Function Expression:**

**Hoisting**: Function statements are hoisted, allowing them to be called before their definition, whereas function expressions are not hoisted and can only be used after their assignment.

**Syntax:** Function statements have the form function name() { ... },while function expressions are often assigned to variables with the form const name = function() {...};

first();      // First Function Called!

second();    //  ReferenceError: Cannot access 'second' before initialization

//Satement:

function first() {

    console.log("First Function Called!");

}

// Expression

var second = function () {

    console.log("Second Function Called!");

}

### **Summary**

* **Function Statement (Declaration)**:
  + Syntax: function name() { ... }
  + Hoisted: Yes
  + Must be named: Yes
* **Function Expression**:
  + Syntax: const name = function() { ... };
  + Hoisted: No
  + Can be named or anonymous: Yes
* **Anonymous Function:**

An anonymous function in JavaScript is a function that does not have a name. These functions are typically used in places where functions are passed as arguments, assigned to variables, or used as immediate functions.

1. **Assigned to a Variable**:

const greet = function() {

    console.log("Hello, Beautiful");

};

greet();

1. **Passed as an Arguments:**

setTimeout(function() {

    console.log("This message is delayed.");

}, 1000);

### **Characteristics:**

* **No Name**: Unlike named functions, anonymous functions do not have an identifier.
* **Common Usage**: Frequently used in callbacks, event handlers, and as IIFEs.
* **Scope**: They inherit the scope in which they are defined, just like any other function.

Anonymous functions are a powerful feature in JavaScript, allowing for more concise and flexible code, especially in functional programming patterns.

* **Named Function:**

A named function expression in JavaScript is a function expression where the function has a name. This name is only accessible within the function's own scope, which can be useful for recursion or for more descriptive debugging.

 **Useful for Recursion**: Named function expressions are particularly useful for recursive functions as they allow the function to call itself.

 **Not Hoisted**: Like all function expressions, named function expressions are not hoisted, so they cannot be called before they are defined.

.

const factorial = function fact(n) {

    if(n<=1)  return 1;

    return n\* fact(n-1);

};

console.log(factorial(5));      // output: 120

console.log(fact);              // ReferenceError: fact is not defined

* **Difference between Parameter And Argument:**

 **Definition vs. Invocation**: Parameters are defined in the function declaration, while arguments are provided during the function call.

 **Placeholders vs. Actual Values**: Parameters are placeholders for the data the function will process, and arguments are the actual data passed to the function.

.

function greet(name) { // 'name' is a parameter

    console.log("Hello, " + name);

}

greet("Alice"); // "Alice" is an argument

* **First class function:**

The ability to use a function as a value and it can be passed as an arguments to another function and can be return as a functions this ability of function known as **FIRST CLASS FUNCTIONS.**

**Characteristics of First-Class Functions:**

1. **Assignment to Variables**: Functions can be stored in variables, allowing them to be passed around and used like any other value.

const add = function(a, b) {

    return a + b;

};

1. **Pass as Arguments**: Functions can be passed as arguments to other functions, allowing for higher-order functions.

function operate(fn, a, b) {

    return fn(a, b);

}

operate(add, 5, 3); // 8

1. **Return from Functions**: Functions can return other functions, enabling the creation of function factories or closures.

function multiplier(factor) {

    return function(number) {

        return number \* factor;

    };

}

const double = multiplier(2);

console.log(double(5)); // 10

* **Callback Functions:**

A callback function in JavaScript is a function that is passed as an argument to another function and is executed after some operation has been completed. Callbacks are a way to ensure that certain code does not execute until another code has finished executing. This is especially useful for asynchronous operations, such as reading files, making network requests, or handling user interactions.

***Characteristics of Callback Functions:***

1. **Passed as an Argument**: A callback function is passed as an argument to another function.
2. **Executed Later**: The callback is executed after the completion of a certain task or event.

***Examples:***

function greet(name, callback) {

    console.log("Hello," + name);

    callback();

}

function saygoodbye() {

    console.log("Goodbye!!");

}

greet("Beautifull", saygoodbye);

// Hello, Beautifull   Goodbye!!

***Asynchronous Callback function:***

*JavaScript is a* ***“Synchronous Single-Threaded Language”.*** *that means it can do one thing at a time in a specific order, But due to* ***“CallBack Functions”*** *we can do Async things in javascript*

*Callbacks are often used to handle asynchronous operations,  such as reading a file or making an HTTP request.*

function fetchdata(callback) {

    setTimeout(function () {

        const data = {name: "Shifra", identity: "Robot", age: 200};

        callback();

    }, 2000);

 }

function processData(data) {

    console.log("Received Data:", data);

 }

 fetchdata(processData);

// Output (after 2 seconds delay):

// Received data: { name: "Alice", age: 25 }

* **Main Thread Functions:**

The **main thread** **(often referred to as the "event loop" or "single thread")**  in JavaScript is the single thread of execution where all the synchronous code runs, including rendering the user interface, handling user events, and executing JavaScript code. Since JavaScript is single-threaded by nature, the main thread processes one task at a time, and any long-running task can block the execution of subsequent tasks, impacting the responsiveness of the web page.

***Blocking the Main Thread*** refers to the situation in JavaScript where a long-running, synchronous task occupies the main execution thread, preventing it from performing other tasks such as updating the user interface, responding to user interactions, or executing asynchronous callbacks. This can lead to a sluggish or unresponsive user experience as the browser is unable to process any other operations until the blocking task is completed.

console.log("Start");

// Long-running task

for (let i = 0; i < 1e9; i++) {

    // Simulating heavy computation

}

console.log("End");

// This event handler will not be executed until the long-running task completes

document.getElementById("myButton").addEventListener("click", function() {

    console.log("Button clicked");

});

In this example, while the long-running task (the for loop) is executing, the browser cannot respond to user interactions, such as clicking the button, until the loop finishes. This illustrates the concept of the main thread and Blocking of Main Thread and the importance of avoiding blocking it to ensure a responsive user experience.

***Avoiding Blocking the Main Thread*** involves using asynchronous programming techniques and APIs to perform time-consuming operations in a non-blocking manner. This ensures that the main thread remains free to handle user interactions and UI updates, providing a smooth and responsive user experience.

***Techniques to Avoid Blocking the Main Thread***

1. **Asynchronous Functions**: Use setTimeout, setInterval, Promises, and async/await to handle tasks asynchronously.

function heavyTask() {

    for (let i = 0; i < 1e9; i++) {

        if (i % 1e6 === 0) {

            setTimeout(heavyTask, 0); // Schedule the next chunk

            break;

        }

    }

}

heavyTask();

2.**RequestAnimationFrame:** Use requestAnimationFrame for animations and UI updates to ensure they run smoothly and synchronously with the browser's repaint cycles.

function animate() {

    // Perform animation step

    requestAnimationFrame(animate);

}

requestAnimationFrame(animate);

**Summary**

* **Main Thread**: The primary execution context where JavaScript code runs, handles user interactions, and updates the UI.
* **Blocking the Main Thread**: Occurs when long-running tasks prevent the main thread from performing other tasks, leading to an unresponsive user experience.
* **Avoiding Blocking**: Utilize asynchronous operations, Web Workers, and requestAnimationFrame to keep the main thread responsive and ensure a smooth user experience.
* **EventListeners in js:**

**Syntax:**

element.addEventListener(event, function, useCapture);

***Adding Event Listeners*** (**Parameters):**

* **event:** A string representing the event type to listen for (e.g., 'click', 'mouseover', 'keydown').
* **function:** The callback function to execute when the event occurs.
* **useCapture (optional):** A boolean indicating whether to use event capturing or bubbling. The default is false (bubbling).

***Event Listeners in JavaScript:***

**Event listeners** in JavaScript are functions that wait for specific events to occur on a particular element or object. When the specified event occurs, the event listener executes the associated callback function. This mechanism is fundamental for creating interactive and dynamic web applications.

The addEventListener() method of the EventTarget interface sets up a function that will be called whenever the specified event is delivered to the target. This method allows multiple event handlers on an element, enabling dynamic and flexible interaction management within web applications.

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <title>Event Listener Example</title>

</head>

<body>

    <h2>Welcome To JavaScript World!!</h2>

    <button id="clickMe">Click Me</button>

    <script>

        // Select the button element

        const button = document.getElementById('clickMe');

        // Define the event listener function

        function handleClick() {

            alert('Button was clicked!');

        }

        button.addEventListener('click', handleClick);

    </script>

**\*why we need to remove EventListener**

Event Listener are heavy that means it takes memory, When an event listener is attached to an element, it creates a closure that retains references to variables and functions in its lexical scope. This closure remains in memory as long as the event listener is active. If the event listener is not removed when it's no longer needed, these closures continue to exist, potentially leading to memory leaks.

By removing the event listener, you effectively release the closure and its associated resources, allowing them to be garbage collected. This ensures that memory is efficiently managed, reducing the risk of memory leaks and improving the overall performance and stability of your application.

    <button id="clickMe">Click Me</button>

    <button id="removeListener">Remove Listener</button>

    <script>

        const clickButton = document.getElementById('clickMe');

        const removeButton = document.getElementById('removeListener');

        // Define the callback function

        function handleClick() {

            alert('Button was clicked!');

        }

        // Attach the event listener

        clickButton.addEventListener('click', handleClick);

        // Remove the event listener when the second button is clicked

        removeButton.addEventListener('click', function() {

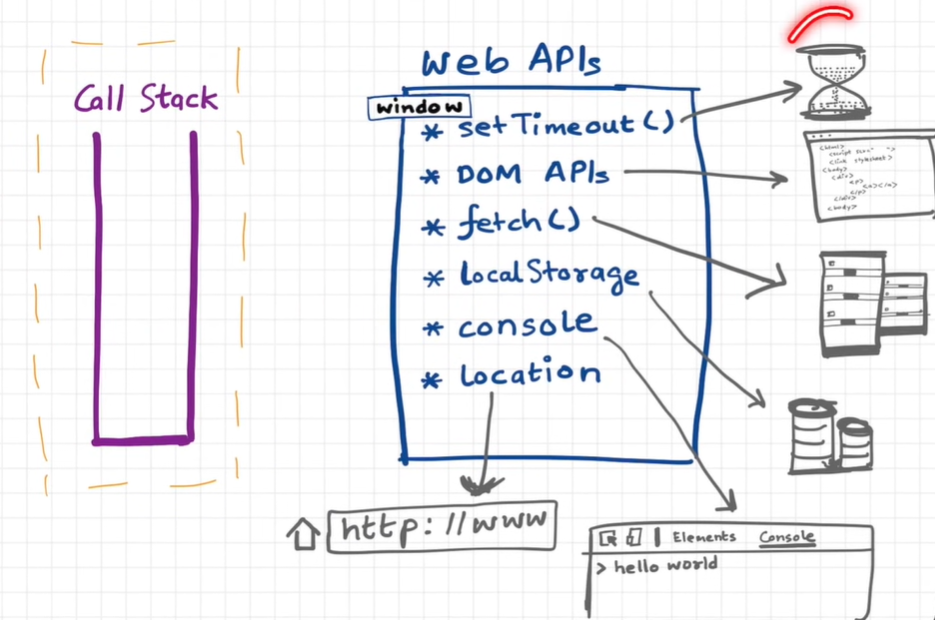
            clickButton.removeEventListener('click', handleClick);

            alert('Click listener removed');

        });

    </script>

In JavaScript, the event loop, callback queue, and microtask queue are key concepts that help manage asynchronous operations and ensure that JavaScript, which is single-threaded, can handle multiple tasks without blocking the main execution thread. Here's an explanation of each:



* Event Loop:

The event loop is a mechanism that continuously checks the call stack and the task queues (macro and microtask queues) to determine what to execute next. Its primary job is to pick up tasks from these queues and push them onto the call stack for execution. The event loop ensures that asynchronous operations like I/O, timers, and promises are handled efficiently.

* Call Stack:

The call stack is a data structure that keeps track of function calls in the program. When a function is called, it is pushed onto the call stack. When the function returns, it is popped off the call stack.

* Callback Queue (Task Queue)

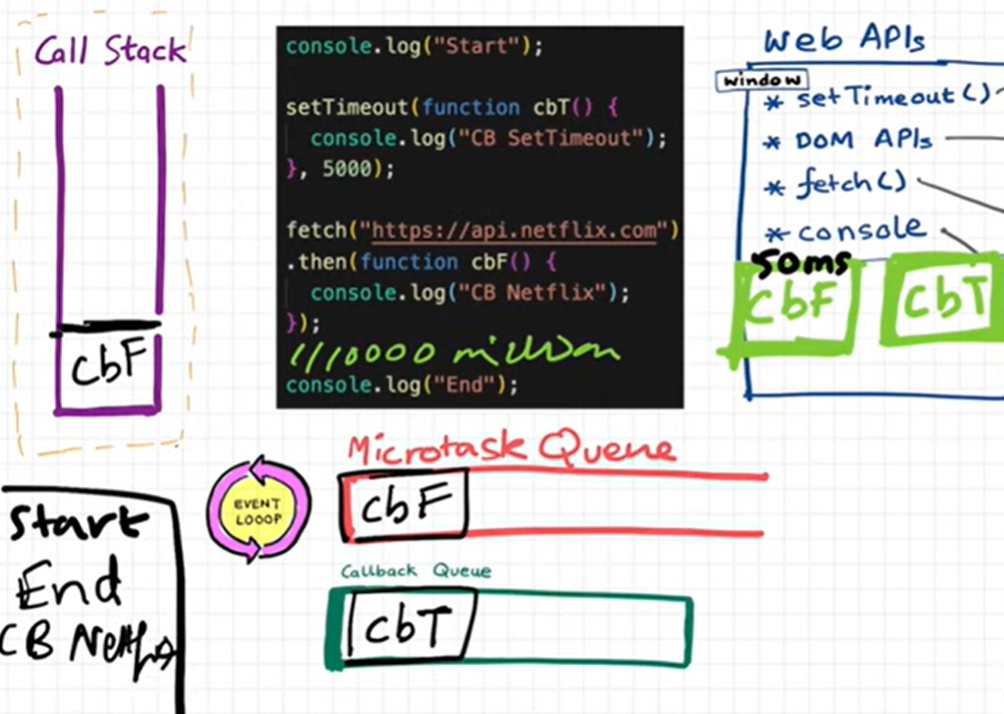
The callback queue, or task queue, is where callback functions from asynchronous operations (like setTimeout, setInterval, and I/O tasks) are placed when they are ready to be executed. These tasks are typically referred to as "macro tasks". The event loop processes tasks in the callback queue after the call stack is empty.

* Microtask Queue

The microtask queue is a separate queue used for tasks that need to be processed after the currently executing script and before any tasks from the callback queue. Examples of microtasks include Promise callbacks (.then, .catch, .finally) and MutationObserver callbacks. Microtasks are processed after the current task but before the next task in the callback queue, giving them higher priority.

***How It All Works Together***

1. **Synchronous Code Execution**: The call stack starts with synchronous code. Each function call is pushed onto the call stack and popped off once it completes.
2. **Asynchronous Task Handling**:
   * When an asynchronous function (like setTimeout) is called, the callback is placed in the callback queue (task queue) after the specified time.
   * Promise resolutions and other microtasks are placed in the microtask queue.
3. **Event Loop Processing**:
   * The event loop checks if the call stack is empty.
   * If the call stack is empty, it first processes all tasks in the microtask queue, executing them one by one until the microtask queue is empty.
   * After the microtask queue is empty, the event loop then picks up tasks from the callback queue (task queue) and pushes them onto the call stack for execution.

****

We can develop asynchronous code using JavaScript because of the **Event Loop** and **Call-back Queue**, provided by the runtime environment These features provided by JavaScript runtimes enable use to write asynchronous code.

JavaScript by itself is a blocking synchronous language.

We can use a function from Web APIs called setTimeout to create asynchronous behaviour.

console.log('Start');

setTimeout(() => {

  console.log('Timeout callback');

}, 0);

Promise.resolve()

  .then(() => {

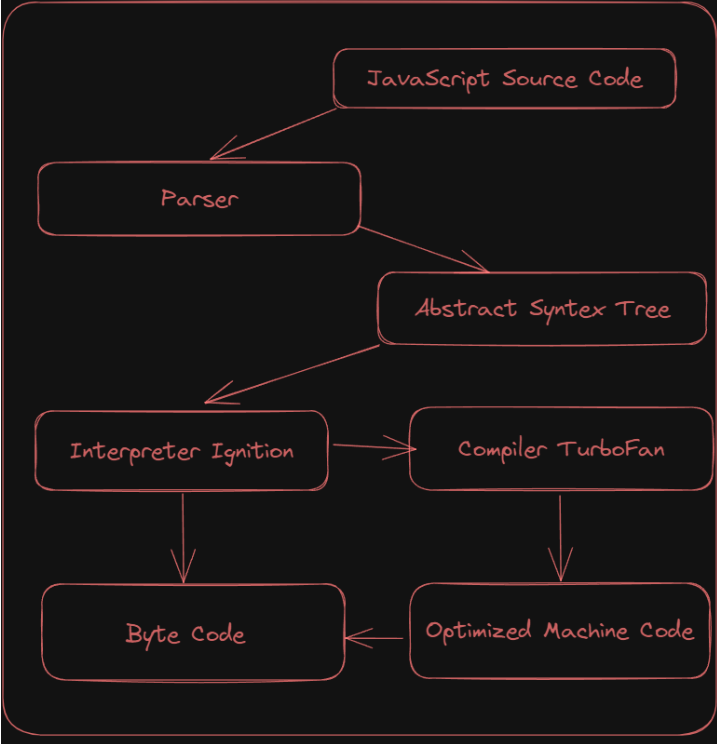
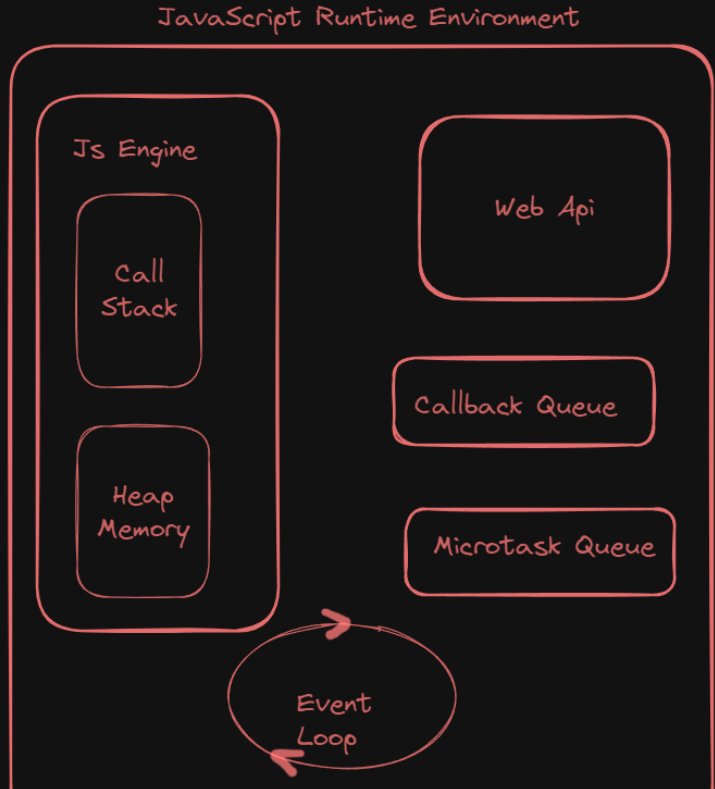
    console.log('Promise callback');

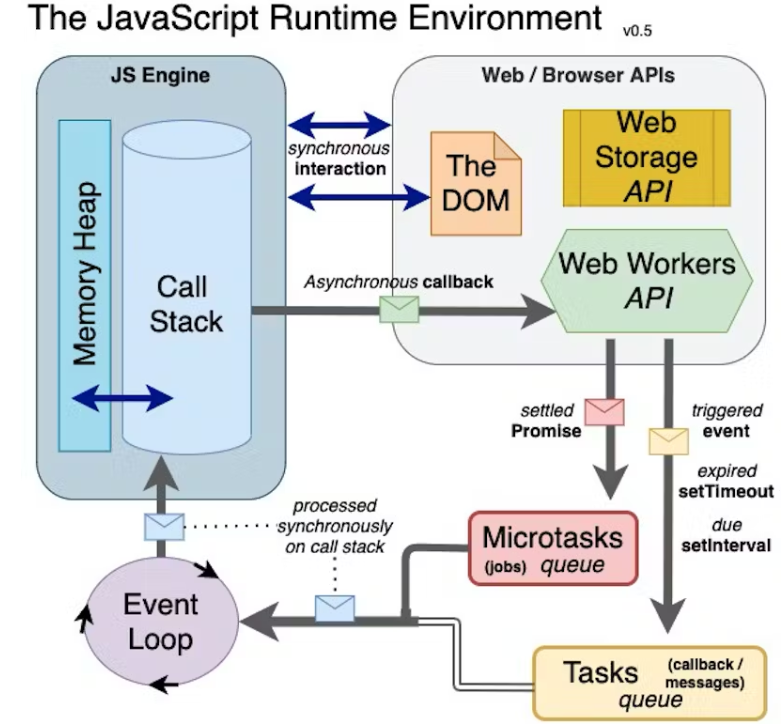
  });

console.log('End');

* The console.log('Start') is executed first and "Start" is printed.
* The setTimeout function sets up a timer and its callback is placed in the callback queue to be executed after 0 milliseconds.
* The Promise.resolve().then(...) sets up a promise resolution and its callback is placed in the microtask queue.
* The console.log('End') is executed next and "End" is printed.
* The call stack is now empty, so the event loop first processes the microtask queue and executes the promise callback, printing "Promise callback".
* After the microtask queue is empty, the event loop processes the callback queue and executes the setTimeout callback, printing "Timeout callback".

***JS Runtime Environment JS (V8 GOOGLE) Engine***

****



* **JavaScript Runtime Environment**

JS Runtime environment is like a big container that has everything present that is required to run JS code. It has a JS engine, Web API, Callback Queue, MicrotaksQueue and EventLoop.

First JS Engine: Spider Monkey developed by Brendon Eich (Father Of JS). Developed by Mozilla, used in Firefox.

**Runtime Environment Architecture**

1. **JavaScript Engine**: E.g., V8 in Chrome.
2. **Web APIs**:
   * DOM API: Document Object Model manipulation.
   * Fetch API: Network requests.
   * Timers: setTimeout, setInterval.
   * Geolocation API: Access to location data.
3. **Callback Queue (Task Queue)**: Queues macro tasks like I/O events and timer callbacks.
4. **Microtask Queue**: Queues microtasks like promise callbacks.
5. **Event Loop**: Manages execution of tasks from the callback and microtask queues.

* **JavaScript Engine Architecture**

The javascript engine is not a machine. It's nothing but normal code that is written in a low-level language. It takes a high-level code (javascript code) and converts it into machine-level code that can be executed by our machines For example, Google Chrome javascript engine V8 is written in C++.

**JS Engine uses three main steps to execute the code:**

1. PARSING
2. COMPILATION
3. EXECUTION

**Parsing** - It means reading your code line by line. It takes two major processes, First breaking down your code into tokens and then passing it to the Syntax Parser. The job of a syntax parser is to take the code and convert it into AST(Abstract Syntax Tree). Ast is a way of representing the code as a tree-like structure.

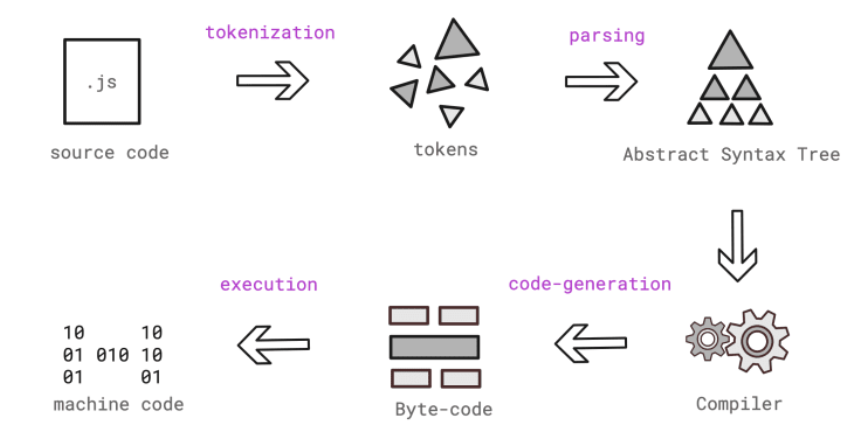
**Compilation** - JavaScript uses JIT ( just-in-time) compilation. It's a way of executing code that involves an interpreter along with the compiler and that mix it as a JIT. Just Think as the AST goes to the interpreter that converts high-level code to bytecodes and that moves to the execution step. While it is doing so it takes the help of the compiler to optimize the code on the run time.

**Execution** - The execution of code is not possible without the two main components of the JavaScript engine

* Memory Heap
* Call Stack

The memory Heap is the place where all the variables and functions are assigned to the memory. It also has a garbage collector that used to free up memory space whenever is possible. It uses a [Mark-and-sweep](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Memory_Management#:~:text=Mark%2Dand%2Dsweep%20algorithm&text=This%20algorithm%20assumes%20the%20knowledge,objects%20referenced%20from%20these%2C%20etc.) algorithm to free up the memory.

The Job of the call stack is to execute the code. It has Globel execution context that invokes in the call stack during run time and executes the code line by line.



***#Astexplorer.net:*** This is a site which generate AST(**Abstract Syntax Tree**)Code.

* **Higher Order Functions:**

A higher-order function in JavaScript is a function that can take other functions as arguments, return a function as its result, or both. And **The function which is passed in Higher Order Function are called as Callback Functions**. This capability allows for greater abstraction, modularity, and reuse of code. Higher-order functions enable powerful programming techniques such as function composition, currying, and the use of callbacks for asynchronous operations.

**Benefits**

* **Reusability**: Write functions that can be applied in various contexts.
* **Modularity**: Break down complex problems into smaller, manageable functions.
* **Abstraction**: Hide implementation details and focus on higher-level operations.

Higher-order functions are fundamental in JavaScript, enabling functional programming paradigms and enhancing code flexibility and maintainability.

/2. Higher-order function that returns a function

function createGreeter(greeting) {

    return function(name) {

      return greeting + ", " + name + "!";

    };

  }

  // Using the higher-order function to create specific greeter functions

  const helloGreeter = createGreeter("Hello");

  const goodMorningGreeter = createGreeter("Good Morning");

  console.log(helloGreeter("Alice")); // "Hello, Alice!"

  console.log(goodMorningGreeter("Bob")); // "Good Morning, Bob!"

* **Higher Order Functions: MAP(), FILTER(), REDUCE()**

**MAP():**The map() method iterates over each element in array and creates new array passing each element to specific function.

The map function is often used to transform data in arrays, such as formatting data, performing calculations, or extracting properties from objects within an array.

**Syntax:**  
**array.map(callback(currentValue, index, array), thisArg)**

**Parameters:** map() method accepts two parameters:

* **callback-** This function is called on each element of array and returns value which will be added in new array. It takes in 3 parameters:
  + **currentValue:** It is a required parameter and it holds the value of the current element.
  + **index:** It is an optional parameter and it holds the index of the current element.
  + **array:** It is an optional parameter and it holds the array.
* **thisArg (optional)-** It is used to hold the value passed to the function. By default it is undefined.

**Return value:** It returns a new array with elements as the return values from the callback function for each element.

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

// Using map to create a new array with each number squared

const squares = numbers.map(function(number) {

  return number \* number;

});

console.log(squares); // [1, 4, 9, 16, 25]

// Arrow Function Example:

// Using map with an arrow function

const doubles = numbers.map(number => number \* 2);

console.log(doubles); // [2, 4, 6, 8, 10]

**FILTER():** The `filter` function in JavaScript creates a new array with all elements that pass a test provided by a function. It keeps only the elements for which the function returns `true`, allowing you to filter out unwanted items from an array and only come up with desired output .

**Syntax:**  
**array.filter(callback(currentValue, index, array), thisArg)**

const array = [5, 7, 8, 9, 1, 3]

// Filter out Odd Values:

function isOdd(X) {

    return X % 2

}

const output = arr.filter(isOdd);

console.log(output);

// 1. Filter greater than three

const output1 = array.filter(function greater(X) {

    return X > 3;

});

console.log(output1);

// 2. Filter with arrow function

const output2 = array.filter((X) => X > 3);

console.log(output2);

**REDUCE():** The ‘ reduce ‘ function in JavaScript is an array method that applies a reducer function to each element of the array, resulting in a single output value. It is used to reduce an array to a single value by iteratively combining each element of the array using the provided function.

Reduce is Basically used at a place where you have to take all the elements of an Array and come up with a single value out of them.

**Syntax:**  
**array.reduce(callback(accumulator, currentValue, index, array), initialValue);**

**callback**: Function to execute on each element in the array, taking four arguments:

* **accumulator:** The accumulator accumulates the callback's return values. It is the accumulated value previously returned in the last invocation of the callback, or initialValue, if supplied.
* **currentValue:** The current element being processed in the array.
* **index (optional):** The index of the current element being processed in the array.
* **array (optional):** The array reduce was called upon.

**initialValue** (optional): A value to use as the first argument to the first call of the callback. If no initial value is supplied, the first element in the array will be used and skipped. Calling reduce on an empty array without an initial value will throw a TypeError.

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

// Using reduce to sum all the numbers in the array

const sum = numbers.reduce(function(accumulator, currentValue) {

  return accumulator + currentValue;

}, 0);

console.log(sum); // 15

In this example, the reduce function is used to sum all the numbers in the numbers array. The accumulator starts at 0 (the initialValue), and each currentValue is added to the accumulator.

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

// Using reduce to find the maximum value in the array

const max = numbers.reduce(function(accumulator, currentValue) {

  return Math.max(accumulator, currentValue);

}, numbers[0]);

console.log(max); // 5

In this example, the reduce function is used to find the maximum value in the numbers array. The accumulator starts at the first element of the array, and each currentValue is compared to the accumulator to find the maximum value.

* *\*Callback hell:*

Callbacks are an essential part of asynchronous programming in JavaScript, allowing functions to be passed as arguments to other functions to be executed later. They are crucial for handling tasks that take time to complete, such as reading files, making network requests, or interacting with databases. Here's a breakdown of their importance and associated issues:

***Importance of Callbacks:***

1. **Asynchronous Operations:** JavaScript is single-threaded, meaning it can only do one thing at a time. Asynchronous operations, facilitated by callbacks, allow the program to continue executing other tasks while waiting for slower operations to complete.
2. **Event Handling:** Callbacks are commonly used for handling events in web development, like button clicks or data loading events.
3. **Modularity:** Callbacks enable modular programming by allowing functions to be passed around as values, promoting code reuse and flexibility.

**Issues with Callbacks:**

1. **\*Callback Hell:** Callback hell, also known as **"pyramid of doom,"** occurs when multiple asynchronous operations are nested within each other, leading to deeply nested and hard-to-read code. This structure makes code difficult to maintain, understand, and debug.
2. **Error Handling:** Error handling in callback-based code can become cumbersome, especially when multiple callbacks are involved. Errors need to be propagated through each level of nesting, leading to verbose and error-prone code.
3. **\*Inversion of Control (IoC):** In callback-based programming, control flow is inverted compared to synchronous code. Instead of the programmer controlling the flow of execution, control is passed to the callback functions, which are executed later. This inversion can make code harder to reason about and debug.

To mitigate these issues, JavaScript developers have adopted various patterns and solutions such as Promises, async/await, and libraries like RxJS for reactive programming. These alternatives aim to provide cleaner and more readable code while still handling asynchronous operations effectively.

* *\*Promises in JavaScript:*

Promises in JavaScript are objects that represent the eventual completion or failure of an asynchronous operation. It provides a cleaner, more readable, and more manageable way to handle asynchronous code compared to traditional callbacks by allowing chaining of asynchronous operations and centralized error handling.

Promises have three states: pending, fulfilled, and rejected.

***Key Characteristics:***

* **Pending:** Initial state, neither fulfilled nor rejected.
* **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).

**Syntax:**

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

//”Producing code” (May take some time)

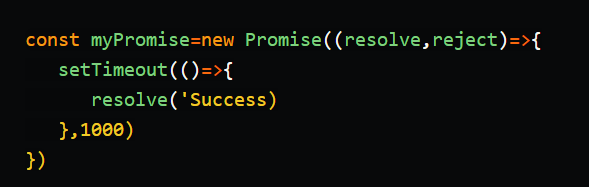
//do something

);

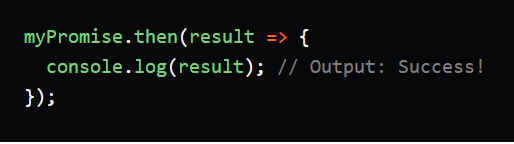
**Parameters:**

* The promise constructor takes only one argument which is a callback function
* The callback function takes two arguments, ***resolve*** and ***reject***
  + Perform operations inside the callback function and if everything went well then call resolve.
  + If desired operations do not go well then call reject

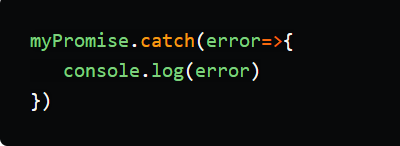
**resolve** and **reject** are functions that bring promise life cycle to completion. When the producing code obtains the result, it should be one of this two callbacks:



Once you have a promise, you can use the **then** method to handle the result when it's ready. The then method takes a single argument, a function that will be called with the result when the promise is fulfilled.



You can also use the catch method to handle errors when a promise is rejected.



***Pyramid of Doom:*** The "Pyramid of Doom" refers to a situation in JavaScript where multiple nested callbacks lead to code that is deeply indented and hard to read and maintain. This typically occurs when handling several asynchronous operations in sequence, resulting in a structure that resembles a pyramid. The code becomes difficult to manage due to the nested callbacks, making debugging and maintaining the code challenging.

**Problems with Pyramid of Doom:**

1. **Readability:** The nested structure makes the code hard to read and understand.
2. **Maintainability:** Adding, removing, or modifying the logic becomes cumbersome due to the deeply nested callbacks.
3. **Error Handling:** Propagating errors through multiple levels of callbacks is complex and error-prone.

**Solution: Promises** can help flatten the nested structure, making the code more readable and manageable.

const cart = ["Shoe", "Shirt", "Jeans", "Watch", "Top"];

function makeOrder(cart) {

    setTimeout(() => {

        const order\_Id = "12345";

        console.log('Order Created:' , order\_Id);

        cancelIdleCallback(order\_Id);

    }, 1000);

}

function proceedToPayment(order\_Id, callback) {

    setTimeout(() =>{

        const payment\_Info = {orderId: order\_Id,status: "Paid"};

        console.log('Payment Processed:', payment\_Info);

        callback(payment\_Info);

    },1000);

}

function showOrderSummary(payment\_info, callback) {

    setTimeout(() => {

        console.log('Order Summary show for:', payment\_info.order\_Id);

        callback();

    },1000);

}

function updateWalletBalance(callback) {

    setTimeout(() => {

        console.log('Wallet Balance Updated');

        callback();

    }, 1000);

}

// PYRAMID OF DOOM PROGRAM

makeOrder(cart, function (order\_Id) {

    proceedToPayment(order\_Id, function(payment\_Info) {

        showOrderSummary(payment\_Info, function() {

            updateWalletBalance(function(){

                console.log("All Operations are completed");

            });

        });

    });

});

// Using PROMISES to avoid Pyramid of Doom

makeOrder(cart)

    .then(order\_Id => proceedToPayment(order\_Id))

    .then(payment\_Info => showOrderSummary(payment\_Info))

    .then(() => updateWalletBalance())

    .then(() => {

        console.log('All operations complete');

    })

    .catch(error => {

        console.error('Error:', error);

    });

* *Promises chaining:*

Promise chaining in JavaScript refers to the process of executing multiple asynchronous operations in sequence using Promises, where each subsequent operation starts after the previous one completes. This is achieved by returning a new promise from the .then() method, allowing you to chain multiple .then() calls together. This approach provides a clean and readable way to handle a series of asynchronous tasks without falling into the "Pyramid of Doom."

The  promise.then() call always returns a promise. This promise will have the state as pending and result as undefined. It allows us to call the next .then method on the new promise.

When the first .then method returns a value, the next .then method can receive that. The second one can now pass to the third .then() and so on. This forms a chain of .then methods to pass the promises down. This phenomenon is called the Promise Chain.

promise

    .then(value1 => {

        // Handle resolved promise and return another promise

        return anotherPromise;

    })

    .then(value2 => {

        // Handle the next resolved promise

        return yetAnotherPromise;

    })

    .catch(error => {

        // Handle any error that occurs in the chain

    });

// Promise Chain with multiple then and catch

let promise = getPromise(ALL\_POKEMONS\_URL);

promise.then(result => {

    let onePokemon = JSON.parse(result).results[0].url;

    return onePokemon;

})

.then(onePokemonURL => {

    console.log(onePokemonURL);

    return getPromise(onePokemonURL);

})

.then(pokemon => {

    console.log(JSON.parse(pokemon));

})

.catch(error => {

    console.log('In the catch', error);

});

* *Promises constructor:*

The Promise constructor in JavaScript is used to create a new Promise object. It takes a single argument: a function called the executor. The executor function is executed immediately by the Promise implementation, and it receives two functions as arguments: resolve and reject. The executor function performs some asynchronous operation and then calls either resolve (if the operation is successful) or reject (if the operation fails).

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

    // Asynchronous operation

    if (/\* operation successful \*/) {

        resolve(value); // Operation successful

    } else {

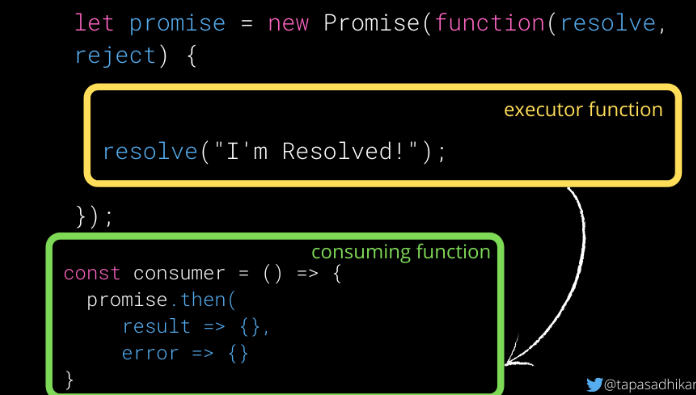
        reject(error); // Operation failed

    }

});

* *Promises Error Handling:*

A Promise uses an executor function to complete a task (mostly asynchronously). A consumer function (that uses an outcome of the promise) should get notified when the executor function is done with either resolving (success) or rejecting (error).



The .then() method should be called on the promise object to handle a result (resolve) or an error (reject).

It accepts two functions as parameters. Usually, the .then() method should be called from the consumer function where you would like to know the outcome of a promise's execution.

Error handling in Promises is done using the .catch() method. Any error that occurs during the execution of the promise or in any of the .then() callbacks can be caught and handled in the .catch() method.

let promise = getPromise(ALL\_POKEMONS\_URL);

const consumer = () => {

    promise.then((result) => {

            console.log({result}); // Log the result of 50 Pokemons

        })

        .catch(error => {

            // As the URL is a valid one, this will not be called.

            console.log('We have encountered an Error!', error); // Log an error

        });

};

consumer();

The .finally() handler performs cleanups like stopping a loader, closing a live connection, and so on. The finally() method will be called irrespective of whether a promise resolves or rejects. It passes through the result or error to the next handler which can call a .then() or .catch() again.

* The .finally() method makes loading false.
* If the promise resolves, the .then() method will be called. If the promise rejects with an error, the .catch() method will be called. The .finally() will be called irrespective of the resolve or reject.

let loading = true;

loading && console.log('Loading...');

// Gatting Promise

promise = getPromise(ALL\_POKEMONS\_URL);

promise.finally(() => {

    loading = false;

    console.log(`Promise Settled and loading is ${loading}`);

}).then((result) => {

    console.log({result});

}).catch((error) => {

    console.log(error)

});

* ***Promise API****(Application Programming Interface):*

Apart from the handler methods (.then, .catch, and .finally), there are six static methods available in the Promise API. The first four methods accept an array of promises and run them in parallel.

1. Promise.all
2. Promise.any
3. Promise.allSettled
4. Promise.race
5. Promise.resolve
6. Promise.reject

const promise1 = new Promise((resolve, reject) => {

    setTimeout(resolve, 1000, 'First promise resolved');

});

const promise2 = new Promise((resolve, reject) => {

    setTimeout(resolve, 2000, 'Second promise resolved');

});

const promise3 = new Promise((resolve, reject) => {

    setTimeout(reject, 1500, 'Third promise rejected');

});

const promise4 = new Promise((resolve, reject) => {

    setTimeout(resolve, 2500, 'Fourth promise resolved');

});

* **Promise.all( ): {Wait for All Promises to Success but when one promise get fail it give an error and other Promises result will be ignored}**

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

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

Promise.all([promise1, promise2, promise3, promise4])

.then(result => {

    console.log(result);

})

.catch((error) => {

    console.log("Error Found!");

});

* **Promise.allSettled( ): {Wait for All Promises to Success, but even if any one promise get fail it still wait for other Promises to be Finished and then give a result of Success or Failure in same order}**

**Promise.allSettled([promises])**- This method waits for all promises to settle(resolve/reject) and returns their results as an array of objects. The results will contain a state (fulfilled/rejected) and value, if fulfilled. In case of rejected status, it will return a reason for the error.

Promise.allSettled([promise1, promise2, promise3, promise4])

.then(result => {

    console.log(result);

})

.catch((error) => {

    console.log("Error Found!");

});

* **Promise.any( ): {Wait for First Settled Success and give that Result, but if all promises result will be failure it gives an Aggregated Error}**

**Promise.any([promises])** - Similar to the race () method, .any() also accepts an array of promises to execute them in parallel. This method doesn't wait for all the promises to resolve. It is done when any one of the promises is settled.{First Success}. If all of the promises are Fail then it gives us Aggregated Error.

**Aggregated Error ( Is an Array of all the Promises Errors )**

Promise.any([promise1, promise2, promise3, promise4])

.then(result => {

    console.log(result);

})

.catch((error) => {

    console.log("Error Found!",error);

    console.log(error.errors)

});

* **Promise.race( ): {Give First Result Irrespective of Success or Failure}**

**Promise.race([promises]) –** It waits for the first (quickest) promise to settle, and returns the result/error accordingly. The **Promise.race()** method returns a promise that fulfills or rejects as soon as one of the promises in an iterable fulfills or rejects, with the value or reason from that promise.

Promise.race([promise1, promise2, promise3, promise4])

.then(result => {

    console.log(result);

})

.catch((error) => {

    console.log("Error Found!");

});

* **Promise.resolve( ):**

Promise.resolve(value) – It resolves a promise with the value passed to it. It is the same as the following:

**let promise = new Promise(resolve => resolve(value));**

* **Promise.reject( ):**

Promise.reject(error) – It rejects a promise with the error passed to it. It is the same as the following:

**let promise = new Promise((resolve, reject) => reject(error));**

* *Async / Await Function:*

An async function in JavaScript is a function that operates asynchronously via the event loop, using an implicit Promise to return its result. These functions allow you to write asynchronous code that looks synchronous by using the await keyword to pause execution until a Promise is resolved or rejected. This makes it easier to work with Promises and handle asynchronous operations in a more readable and maintainable way.

Async functions always return a promise. If a value is returned that is not a promise, JavaScript automatically wraps it in a resolved promise.

// Syntax:

async function functionName() {

    // asynchronous code

}

*Await Function* - async/await is a syntactic feature in JavaScript that allows you to write asynchronous code in a more synchronous-looking manner. It is built on top of Promises and helps improve readability and maintainability of asynchronous code.

The await keyword can only be used inside an async function. It pauses the execution of the async function until the Promise is resolved or rejected.

Await makes the code wait until the promise returns a result, allowing for cleaner and more manageable asynchronous code.



*The****async****keyword transforms a regular JavaScript function into an asynchronous function, causing it to return a Promise.*

*The****await****keyword is used inside an async function to pause its execution and wait for a Promise to resolve before continuing.*

async function functionName() {

    const result = await promise;

    // Continue execution after the promise is resolved

}

async function functionName() {

    const result = await promise;

    // Continue execution after the promise is resolved

}

// Function to fetch data from a given URL using async/await

async function fetchData(url) {

    try {

        const response = await fetch(url);

        if (!response.ok) {

            throw new Error('Network response was not ok');

        }

        const data = await response.json();

        return data;

    } catch (error) {

        console.error('Fetch error:', error);

        throw error; // Rethrow the error to be handled by the caller

    }

}

// URL to fetch data from

const API\_URL = 'https://pokeapi.co/api/v2/pokemon?limit=10';

// Calling the async function and handling the result

(async () => {

    try {

        const data = await fetchData(API\_URL);

        console.log('Fetched data:', data);

    } catch (error) {

        console.error('Error fetching data:', error);

    }

})();