**JavaScript**

**JavaScript execution context**

Execution context is the environment where code is executed and determines what variables, objects and functions are accessible. There are execution context are 3 types

1. **Global execution context**
2. **Function execution context**
3. **Eval execution context**

**Global Execution context**

The global execution context is the default environment where JavaScript code starts executing. It is created automatically when the JavaScript engine runs script for the first Tim, even before any code is execution. One global execution context exists per program. It is created before any code executed. It contains Global variables, Global functions., Global objects and this keyword which refers to the Global object.

Key characteristics of the global execution context

1. **Global scope**
2. **Global object**
3. **The this keyword**
4. **Hoisting in GEC**
5. **Single Instance**

Phase of Global Execution Context

1. **Creation phase**
2. **Execution phase**

**Function Execution context**

The function execution context is created every time a function is called in JavaScript. It manages the execution of code inside the function and contains details about the functions scope, variables, and behavior during its execution.

Key characteristics of function execution context

1. Created per function call
2. Contains
3. Lifecycle
4. Stack behavior
5. Hoisting

Phases of function execution context

1. **Creation phase**
2. **Execution phase**

**Eval execution context**

The eval execution context is created when the eval() function is used to execute a string of code. It is a special execution context that allows JavaScript to evaluate and run code at runtime

Key characteristics of eval execution context

1. **Dynamic code execution**
2. **Contains**
3. **Security risks**
4. **Lifecycle**

**Event Loop**

The event loop is a core mechanism in JavaScript that enables **non-blocking, asynchronous programming**. It ensures that JavaScript, despite being single threaded, can perform tasks like handling user inputs, executing asynchronous operation, and managing timers without stopping everything.

It enables JavaScript code execute smoothly even if there are tasks that take time, like waiting for a response from the internet / a timer.

Key characteristics of the Event loop

**Single threaded**

* JavaScript runs on a single thread, meaning it can execute only one task at a time in the call stack.

**Manages asynchronous operations**

* The event loop handles tasks from asynchronous APIs like setTimeout, promises, and user events (clicks, keypresses).

**Non blocking**

JavaScript delegates long-running tasks (like network request / timers) to the browser / node.js runtime, allowing the main thread to continue running other code.

**Interaction with call stack and queue**

The event loop coordinates the execution of tasks from the call stack, callback queue, and microtask queue.

1. Call stack  
   The call stack is a data structured used to track the function execution order in JavaScript.

Key features

Stack behavior

Function execution

Booking nature

1. Callback queue

The callback queue holds asynchronous callback that are waiting to be executed.

Key features

FIFO order

Interaction with the Event loop

Includes

console.log("callback queue");

setTimeout(() => {

  console.log("callback");

}, 1000);

console.log("End")

Execution order

Start is logged

setTimeout schedules the callback and moves it to the callback queue.

End is logged

After 1 second, the callback is moved from the callback queue to the call stack and executed.

1. Microtask queue

The microtask queue is a special queue for microtasks, which are high priority tasks scheduled for execution before tasks in the callback queue.

Key features

1. Higher priority
2. Includes
3. Executed After current task

console.log("microtask start");

setTimeout(() => {

  console.log("callback");

}, 1000);

Promise.resolve().then(() => {

  console.log("microtask promise");

});

console.log("End")

Execution order

Start is logged

setTimeout schedules the callback in the callback queue.

Promise.resolve() schedules a microtask in the microtask queue

“End” is logged

The microtask is executed

The callback is executed.