Event Loop Timeline Notes

▼ Execution Context & Call Stack

The JS engine parses the entire script and creates the **global execution context,** where the global code is executed.

Each execution context goes through **two phases**: the creation phase and the execution phase.

During the creation phase, the JS engine:

1. Allocates memory in the stack and heap for the variables and functions related to this context.
2. Initializes variables with **undefined** and stores references to function declarations.
3. Sets the value of the **this** keyword within that context.
4. Keeps a reference to the outer scope, which is used in closures to access outer variables.

During the execution phase, the engine executes the script line by line. When it encounters a function, a new **function execution context** is created and pushed on top of the global execution context.

To keep track of execution contexts, JavaScript uses a **call stack**.

When the engine starts executing the script, it creates the global execution context and pushes it onto the call stack. When a function is called, a new execution context is created and pushed onto the stack on top of the global execution context.

The function executes within this new context. Once the function finishes, its execution context is **popped** from the stack, and control returns to the previous context (usually the global context) to continue executing the remaining code.

The call stack follows the **LIFO** (Last In, First Out) principle.

The stack has a limited size, and exceeding this limit results in a stack overflow.

▼ Event Loop

JavaScript is **single-threaded and synchronous by nature**, since it runs one line of code at a time.
However, the browser or runtime environment provides Web APIs (like setTimeout, event listeners, or fetch) that allow long-running tasks to run in the background without blocking the call stack.

During execution, when the JS engine encounters asynchronous code (such as setTimeout), it delegates the work to the Web API, which is part of the runtime environment.

For example, with setTimeout, the Web API starts a timer. When the timer finishes, the Web API places the callback into the **macrotask queue**.

The **macrotask queue** is a data structure that holds callbacks from Web APIs such as:

- setTimeout
- setInterval
- UI events (click, scroll, etc.)
- Some other async APIs

Callbacks are queued in the order they become ready.

There is also a **microtask queue**, which holds promise reaction handlers such as:

- .then()
- .catch()

The microtask queue has **higher priority** than the macrotask queue, meaning its handlers must run **before** any macrotask is processed.

The **event loop**, which is part of the runtime environment, continuously checks the call stack.

If the call stack is empty (except for the global execution context, which remains idle while the script is still running), the event loop begins processing queued tasks.

First, it checks the **microtask queue**.

If it contains tasks, the event loop moves them **one by one** to the call stack and runs them **until the microtask queue becomes empty**.

Only when the microtask queue is fully cleared does the event loop take **one** task from the macrotask queue and push it into the call stack for execution.

After that macrotask finishes, the event loop again gives priority to the microtask queue, running all its tasks before processing the next macrotask.

So the rule is:

All microtasks must run before the event loop executes a single macrotask.

```
const name = prompt("Enter your name");

const promise = new Promise((resolve, reject)=>{
	if(name) resolve(name)
	else reject(new Error ("name is not provided by user"))
})

setTimeout(()=>{
	alert(`${name} from setTimeout`)
},0)

promise.then((data) => alert(`${data} from then`)).catch((err) => alert(err.message))

/*
1. get name from user
2. alert statement of then / catch is executed
3. alert statement of setTimeout is executed
*/
```