Introduction to JavaScript Execution

10000 C{}DERS
Right path for a Bright Career.

JavaScript is Single-threaded

- JavaScript operates in a single-threaded environment, which means it has one call stack and can only execute one piece of code at a time. This simplifies the programming model, avoiding issues like deadlocks and race conditions that can occur in multi-threaded environments.

Executes Code in a Synchronous Manner

- JavaScript executes code synchronously, meaning each line of code is executed in sequence from top to bottom. This synchronous behavior ensures that each task must complete before the next one begins. However, this can lead to blocking if a task takes a long time to complete, potentially causing the application to become unresponsive.

Needs Mechanisms to Handle Asynchronous Tasks

- To prevent blocking and keep the application responsive, JavaScript needs mechanisms to handle asynchronous tasks. Asynchronous operations, such as I/O tasks, network requests, or timers, are handled using mechanisms like callbacks, promises, and async/await. These mechanisms allow JavaScript to offload tasks and continue executing other code while waiting for the asynchronous operations to complete.

What is an Event Loop ?

Definition

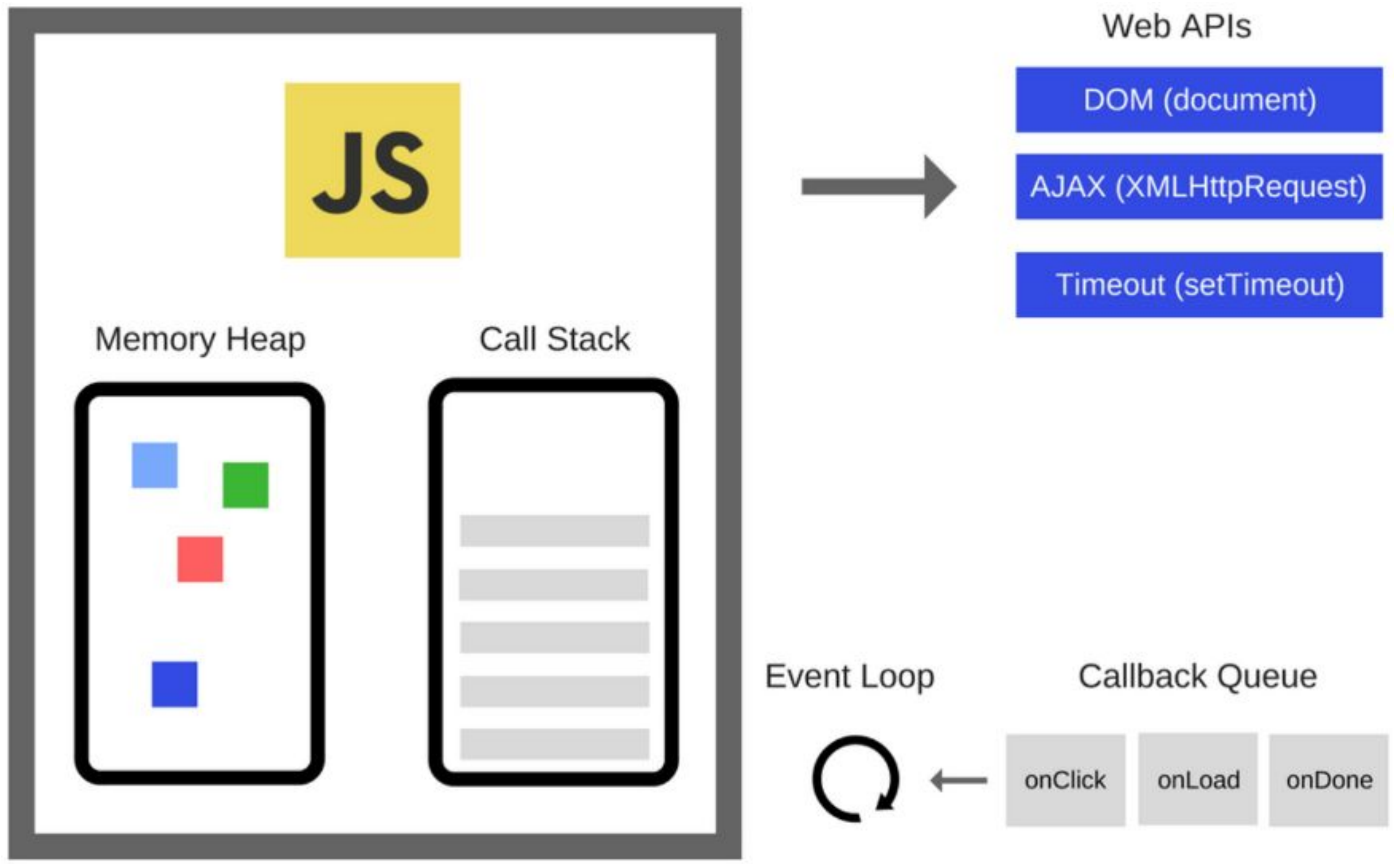
- The event loop is a mechanism that allows JavaScript to perform non-blocking operations by offloading operations to the system kernel whenever possible. When an asynchronous operation is executed, like an I/O task or a network request, JavaScript doesn't wait for the operation to complete. Instead, it continues to execute other code while the system kernel handles the asynchronous operation. Once the operation is completed, the system kernel notifies JavaScript, and the corresponding callback is added to the callback queue to be executed by the event loop.

Importance

- The event loop is crucial for managing asynchronous operations in JavaScript. It allows JavaScript to handle multiple operations without creating new threads, maintaining its single-threaded nature. This mechanism ensures that the application remains responsive and efficient by preventing blocking and managing the execution of asynchronous tasks. By using the event loop, JavaScript can execute callbacks and promises in a controlled manner, providing a smooth user experience even when performing complex or time-consuming tasks.

Event Loop Architecture

Layout :



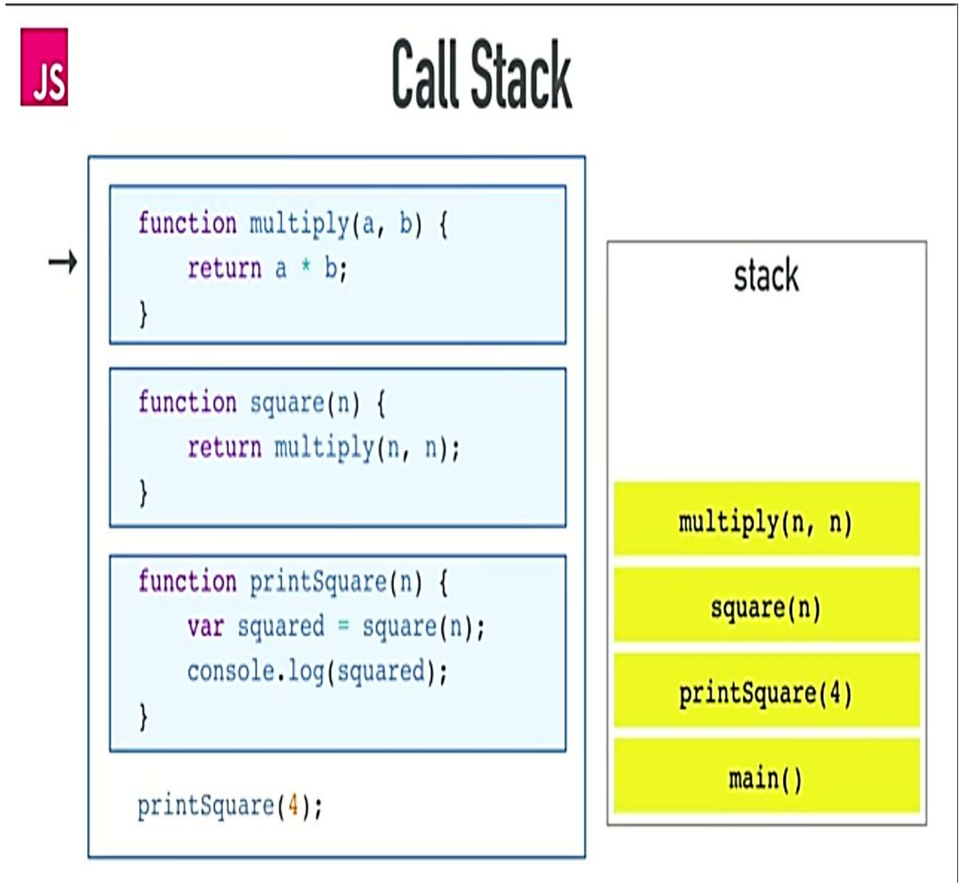
Call Stack

- A call stack is a mechanism for an interpreter (like the JavaScript interpreter in a web browser) to keep track of its place in a script that calls multiple functions . **LIFO (Last In, First Out)**

  - When a script calls a function, the interpreter adds it to the call stack and then starts carrying out the function.

  - Any functions that are called by that function are added to the call stack further up, and run where their calls are reached.

  - When the current function is finished, the interpreter takes it off the stack and resumes execution where it left off in the last code listing



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

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

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

printSquare(4);
```

Web Apis

- They are built into your web browser, and are able to expose data from the browser and surrounding computer environment and do useful complex things with it. They are not part of the JavaScript language itself, rather they are built on top of the core JavaScript language, providing you with extra superpowers to use in your JavaScript code.

- Web APIs (Application Programming Interfaces) are interfaces provided by browsers and web servers that allow developers to interact with the underlying system, hardware, or software features. These APIs enable web applications to perform tasks such as making HTTP requests, manipulating the DOM, accessing device hardware, and handling multimedia content.

Event Loop & Callback Queue

- The event loop is a fundamental concept in JavaScript that enables non-blocking, asynchronous programming. It is a mechanism that allows JavaScript to perform tasks like I/O operations and network requests without blocking the execution of other code. The event loop continuously checks the call stack and the callback queue, executing tasks in the call stack and moving callbacks from the callback queue to the call stack when the stack is empty.

- The callback queue, also known as the task queue, is a queue that holds callbacks and other functions that are waiting to be executed. These callbacks are typically the result of asynchronous operations like setTimeout, setInterval, or I/O tasks. When an asynchronous operation completes, its callback function is added to the callback queue.

Event Loop Execution

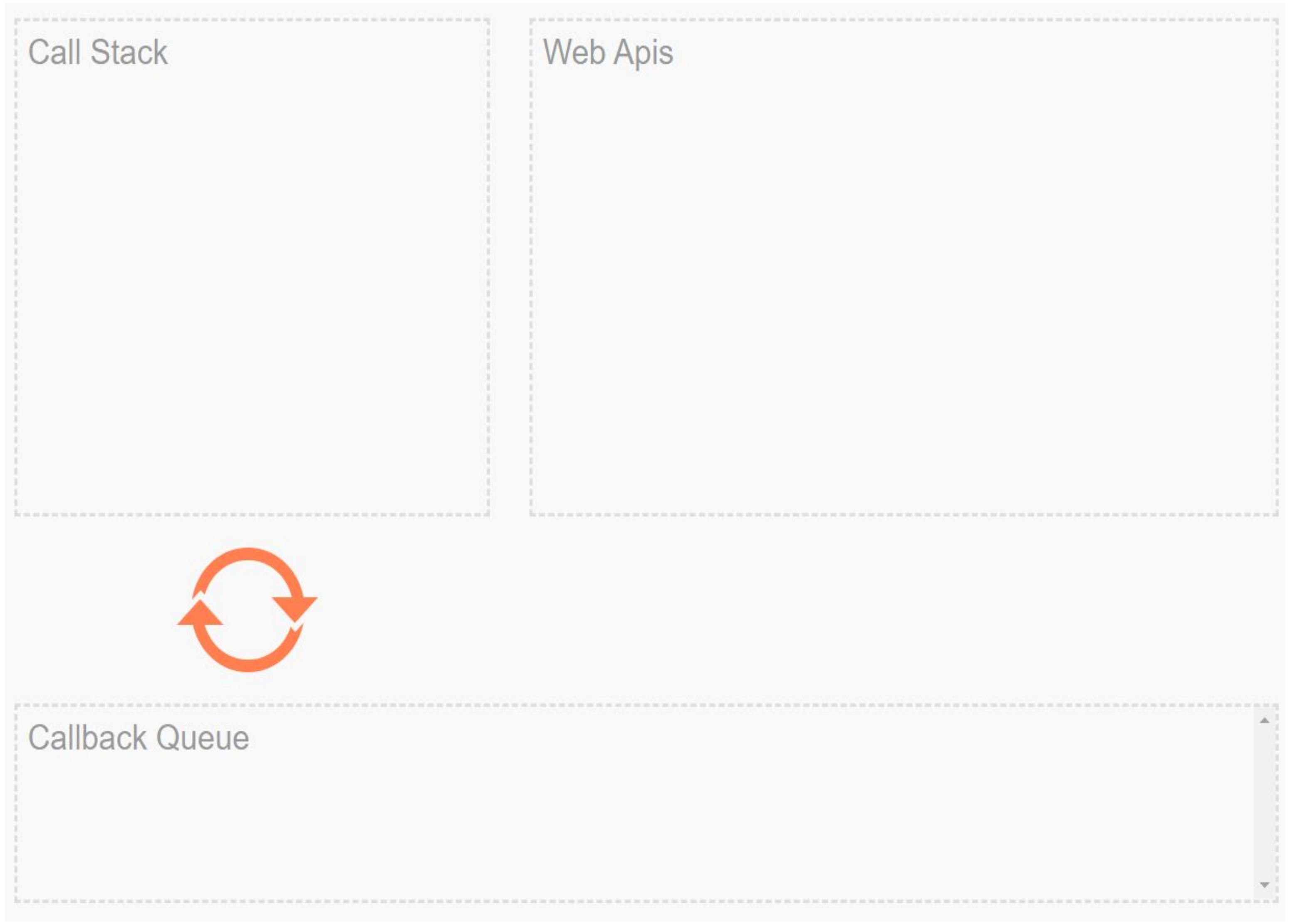
```
console.log(1);

setTimeout(function foo(){

console.log(2);

}, 0);

console.log(3);
```



Order of Execution

- **Execution of Functions on the Call Stack:**
  - When JavaScript code is executed, functions are pushed onto the call stack as they are called (invoked).
  - Each function executes in sequence according to the order they were pushed onto the stack.
  - As each function completes its execution, it is popped off the call stack.
- **Event Loop and Queue:**
  - JavaScript is single-threaded, meaning it can only execute one piece of code at a time.
  - Asynchronous operations like setTimeout, fetch, or handling events (e.g., click events) are managed differently.
  - When an asynchronous task completes (like a setTimeout timer firing or a network request finishing), it's pushed into a queue called the task queue or message queue.

Order of Execution

- **Event Loop Mechanism:**
  - The event loop continuously checks if the call stack is empty.
  - If the call stack is empty, the event loop pops tasks from the queue one by one and pushes them onto the call stack for execution.
  - This process ensures that asynchronous tasks are executed in the order they were placed into the queue.
- **Execution Order:**
  - Tasks in the queue are executed strictly in the order they were added, adhering to the event-driven nature of JavaScript.
  - While the event loop manages the flow of tasks from the queue to the call stack, it also ensures that the main thread remains responsive to user interactions.

Macro-Tasks (Task Queue) & Micro-Tasks

- JavaScript manages asynchronous tasks through two queues: the macro-task queue (or task queue) and the micro-task queue.

- The macro-task queue includes tasks like setTimeout, setInterval, and I/O operations, scheduled to execute after the current call stack clears.

- The micro-task queue contains higher-priority tasks such as Promise callbacks (then and catch), process.nextTick, and queueMicrotask, executed immediately after the current script and before the next macro-task.

- This dual-queue system allows JavaScript to efficiently handle asynchronous operations while ensuring responsive behavior for user interactions.

Order of Execution

- All functions that are currently in the call stack get executed and then they get popped off the call-stack.
- When the call stack is empty, all queued-up micro-tasks are popped onto the call-stack one by one and get executed, and then they get popped off the call-stack.
- When both the call-stack and micro-task queue are empty, all queued-up macro-tasks are popped onto the call-stack one by one and get executed, and then they get popped off the call-stack.

Conclusion

- In conclusion, the event loop is a crucial mechanism in JavaScript that ensures the execution of asynchronous tasks in an efficient and non-blocking manner. By overseeing the management of both micro-tasks, such as Promise callbacks and process.nextTick, and macro-tasks like setTimeout and I/O operations, the event loop maintains the responsiveness of JavaScript applications. This mechanism allows for concurrent handling of multiple tasks, prioritizing micro-tasks to be processed immediately after the current execution context clears. This ensures that JavaScript remains single-threaded yet capable of handling complex asynchronous operations seamlessly, contributing to smoother user interactions and overall application performance. Understanding the event loop is essential for developers to build responsive and efficient JavaScript applications.