


CODE EXERCISE

Deep Dive into JavaScript's Event Loop and Concurrency Model!

CODING EXERCISES TEST YOUR SKILLS



Deep Dive into JavaScript's Event Loop and Concurrency Model! 	1
Exercise 1: Understanding the Event Loop	2
Exercise 2: Exploring Microtasks	3
Exercise 3: Macrotasks and Microtasks	3
Exercise 4: Blocking the Event Loop	4
Exercise 5: Async/Await and the Event Loop	4
Exercise 6: Event Loop with SetImmediate	5
Exercise 7: Process.nextTick in Node.js	6
Exercise 8: Combining Async Operations	6
Exercise 9: Async Function in a Loop	7
Exercise 10: Event Loop with I/O Operations	8

Deep Dive into JavaScript's Event Loop and Concurrency Model!

From exploring the nuances of microtasks and macrotasks to dissecting async/await and event loop intricacies, these exercises are tailor-made for developers eager to master JavaScript's concurrency model and event-driven architecture.

Learn more about JavaScript with Examples and Source Code Laurence Svekis
Courses <https://basescripts.com/>

👉 Embark on this journey to demystify asynchronous JavaScript:

#JavaScript #EventLoop #Concurrency #AsyncProgramming #WebDevelopment
#CodingExercises #TechCommunity

Dive into these exercises, share your experiences, and let's unravel the wonders of the JavaScript event loop together! 🖥️🌐

The JavaScript Event Loop and Concurrency Model are fundamental concepts for understanding how JavaScript handles asynchronous operations, despite being a single-threaded language. These concepts are crucial for developing efficient, non-blocking web applications. Below are 10 coding exercises designed to illustrate various aspects of the event loop, concurrency, and how JavaScript manages asynchronous and synchronous code execution.

Exercise 1: Understanding the Event Loop

Objective: Observe how JavaScript's event loop handles asynchronous and synchronous operations.

```
console.log('Start');
setTimeout(() => {
  console.log('Timeout 1');
}, 0);
Promise.resolve().then(() => console.log('Promise'));
console.log('End');
```

Learn more about JavaScript with Examples and Source Code Laurence Svekis
Courses <https://basescripts.com/>

Explanation: This exercise demonstrates the order of execution in the JavaScript event loop. Despite the `setTimeout` delay being 0, the promise is executed first due to the microtask queue's priority over the callback queue.

Exercise 2: Exploring Microtasks

Objective: Understand the execution order of microtasks.

```
console.log('Start');  
Promise.resolve().then(() => console.log('Promise 1'));  
Promise.resolve().then(() => {  
  console.log('Promise 2');  
  Promise.resolve().then(() => console.log('Promise 3'));  
});  
console.log('End');
```

Explanation: Demonstrates how promises (microtasks) are handled in the JavaScript event loop. Microtasks execute after the currently executing script and before any other macrotasks, such as I/O or timers.

Exercise 3: Macrotasks and Microtasks

Objective: Differentiate between macrotasks and microtasks in the event loop.

```
setTimeout(() => console.log('Timeout'), 0);  
Promise.resolve().then(() => console.log('Promise'));  
// Expected output:  
// Promise  
// Timeout
```

Learn more about JavaScript with Examples and Source Code Laurence Svekis
Courses <https://basescripts.com/>

Explanation: This illustrates how the event loop prioritizes microtasks (promises) over macrotasks (setTimeout). The microtasks queue is processed completely before any macrotasks are considered.

Exercise 4: Blocking the Event Loop

Objective: Observe how blocking the event loop affects asynchronous execution.

```
console.log('Start');
setTimeout(() => console.log('Timeout'), 1000);
const startTime = new Date().getTime();
while (new Date().getTime() - startTime < 2000);
console.log('While loop ended');
```

Explanation: This code blocks the event loop with a synchronous while loop, delaying the execution of the setTimeout callback. It highlights the impact of blocking operations on asynchronous callbacks.

Exercise 5: Async/Await and the Event Loop

Objective: Explore how async/await influences the execution order in the event loop.

```
async function asyncFunction() {
  console.log('Async function start');
  await Promise.resolve();
  console.log('After await');
}
console.log('Global start');
```

```
asyncFunction();
console.log('Global end');
// Expected output:
// Global start
// Async function start
// Global end
// After await
```

Explanation: Demonstrates how `async/await` works within the event loop, with the `await` keyword causing the rest of the `async` function to be executed as a microtask.

Exercise 6: Event Loop with `setImmediate`

Objective: Compare `setImmediate` and `setTimeout` in the context of the event loop.

```
console.log('Start');
setImmediate(() => console.log('SetImmediate'));
setTimeout(() => console.log('Timeout'), 0);
console.log('End');
// Note: `setImmediate` is available in Node.js environment, not in all JavaScript environments.
```

Explanation: This exercise is designed to illustrate the differences between `setImmediate` and `setTimeout` in Node.js, showing how tasks are scheduled in the event loop. The actual output can vary depending on the environment and current phase of the event loop.

Learn more about JavaScript with Examples and Source Code Laurence Svekis
Courses <https://basescripts.com/>

Exercise 7: Process.nextTick in Node.js

Objective: Understand the role of process.nextTick in the event loop.

```
console.log('Start');  
process.nextTick(() => console.log('NextTick'));  
Promise.resolve().then(() => console.log('Promise'));  
console.log('End');  
  
// Note: `process.nextTick` is specific to Node.js.
```

Explanation: Shows how process.nextTick allows developers to schedule callbacks to be invoked at the beginning of the next iteration of the event loop, before any other I/O events, including before promises.

Exercise 8: Combining Async Operations

Objective: Visualize the execution order with a mix of asynchronous operations.

```
setTimeout(() => console.log('Timeout 1'), 0);  
Promise.resolve().then(() => console.log('Promise 1'));  
setTimeout(() => {  
  Promise.resolve().then(() => console.log('Promise 2'));  
  console.log('Timeout 2');  
}, 0);  
  
// Expected output:  
// Promise 1  
// Timeout 1  
// Timeout 2  
// Promise 2
```

Learn more about JavaScript with Examples and Source Code Laurence Svekis
Courses <https://basescripts.com/>

Explanation: This exercise demonstrates how JavaScript handles a combination of asynchronous operations, illustrating the interleaving of microtasks and macrotasks.

Exercise 9: Async Function in a Loop

Objective: Investigate how asynchronous functions behave inside a synchronous loop.

```
for (let i = 0; i < 3; i++) {  
  setTimeout(() => console.log(i), 0);  
}
```

```
for (let i = 0; i < 3; i++) {  
  (async () => console.log(i))();  
}
```

// Expected output:

// 0

// 0

// 0

// 1

// 1

// 1

// 2

// 2

// 2

Explanation: Demonstrates the behavior of asynchronous functions (setTimeout and async functions) inside a synchronous for loop, highlighting the closure and scope handling in JavaScript.

Exercise 10: Event Loop with I/O Operations

Objective: Explore how I/O operations are handled in the event loop.

```
const fs = require('fs');
console.log('Start');
fs.readFile(__filename, () => {
  setTimeout(() => console.log('Timeout'), 0);
  setImmediate(() => console.log('SetImmediate'));
});
console.log('End');
```

// Note: This exercise is specific to Node.js and demonstrates the interaction between I/O callbacks and timers in the event loop.

Explanation: This code aims to show the order of execution for I/O callbacks, setTimeout, and setImmediate in Node.js, illustrating the complexities of the event loop in handling different types of operations.