# Chapter 13 Concurrency Part 2

# 1 Agenda

- Convert a non-async function to an async one
- async and await keywords
- Event Loop concurrency model in JS runtime environment

## 2 Make a non-async function to an async function using the Promise object

Use the `Promise` object to wrap your non-async functions

Steps to create a Promise object:

1. Create a new Promise object by calling the `Promise` constructor.
2. Pass a executor function as the argument to the `Promise` constructor, either an arrow function or a named function.
3. Add your non-async function to the executor function.

```
function executor(resolve, reject) {
  // Execute your non-async code
  // call resolve(value) to resolve the promise object (change the state to fulfilled)
  // or call reject(error) to generate the rejected promises
}
```

Important:

- The executor function still runs in the main thread of the JS engine, not asynchronous code.
- Only the timer-based tasks, I/O tasks, and the Web API tasks run asynchronously in the browser.
- `resolve` and `reject` functions are micro async tasks and will be filed in the Micro-task Queue.

## Example 13-6: Make a non-async function do the async task and return a Promise object

Assume we have a function that take lots of time to complete the task:

```
/**
 *  This function is a long running task that takes a while to complete.
 * @returns
 */
function longtimeTask() {
  console.log('== Enter the long task');
  let i = 0;
  while (i < 10000) {
    i++;
  }
  console.log('== Exit the long task');
  // Return a random number between 0 and 100
  return Math.random() * 100;
}
```

Execute the `longtimeTask` function in an async way:

- Run the non-async function in the executor function of the Promise object.

```
console.log('Start the long running task');

// Use the Promise object to run the long running task
// When the promise is created, the long running task is started
const longtimeTaskPromise = new Promise((resolve, reject) => {
    const result = longtimeTask();
    if (result < 50) {
        reject('The result is less than 50');
    }
    // fulfilled task.
    // file the result to a queue (the Micro-task Queue)
    resolve(result);
});
```

- The function passed to the `then` method is called when the Promise object is resolved.

```
// get the result of the long running task
longtimeTaskPromise
    .then(result => console.log('The result is', result))
    .catch(error => console.error('An error occurred:', error));

console.log('Please wait for the long running task to complete...');
```

The output of the above code is:

```
Start the long running task (main thread)
== Enter the long task (executor function)
== Exit the long task (executor function)
Please wait for the long running task to complete... (main thread)
The result is 61.054982580283635 (then callback function)
```

You may interest in the following article about Promises:

JavaScript Visualized: Promises & Async/Await

## 3 Wrap the non-async function in an async function

You can also wrap your non-async function in an async function that returns a Promise object.

```
function myAsyncFunction() {
    return new Promise((resolve, reject) => {
        // code to execute the async task
        // call resolve(value) to resolve the promise object
        // call reject(reason) to reject the promise object
    });
}
```

## Example 13-7: Rewrite the `longtimeTask` function: Wrap it in an async function

This example rewrites the `longtimeTask` function in Example 13-6 to return a Promise object.

```
function async_longtimeTask(){
    return new Promise((resolve, reject) => {
        const result = longtimeTask();
        if (result < 50) {
            reject('The result is less than 50');
        } else {
            resolve(result); // fulfilled
        }
    });
}
```

Call the `longtimeTask` function in an async way:

```
// call the longtimeTask function
console.log('Start the long running task');

async_longtimeTask()
    .then(result => console.log('The result is', result))
    .catch(error => console.error('An error occurred:', error));

console.log('Please wait for the long running task to complete...');
```

A possible output of the above code is:

```
'Start the long running task'
'== Enter the long task'
'== Exit the long task'
'Please wait for the long running task to complete...'
The result is 15.919520883194348
```

## Misconception in the above code:

Can wrapping the non-async function in an async function make it run asynchronously?

Answer: **No**.

Reasons:

- The `longtimeTask` function does not run asynchronously, although it returns a Promise object.
- **It run in the JS engine, not in the Web API.**
- Because the JS engine is single-threaded, the loop will block the execution of other tasks in the JS engine until the function is resolved.

- To run the task asynchronously, you need to **offload the task to the Web API (or the browser)**.
- Replace the `while` loop with the `setTimeout` function (Web API) to run the task asynchronously.
- See ex_13_07_1.js for the version that offloads the task to the Web API.

## 4 Async and Await

Use the Promise object might still cause the long Promise chain if you have many asynchronous tasks to be run in sequence.

Use the **async/await** syntax to avoid the long Promise chain.

- make the asynchronous code look like synchronous code.

The async/await syntax is a syntactic sugar for the Promise object.

## Rules to use the async and await keywords

- Use the `async` keyword to qualify a function as an async function when defining it.
  - The async function can be a named function or an arrow function.
  - The async function can be a method of an object or a class.
- The async function **always** returns a Promise object implicitly.
  - JS wraps the return value in a Promise object automatically.
- Use the `await` keyword with the async function to wait for the Promise object to be resolved.
- Important: The `await` keyword can only be used **inside** an async function.

## Pattern to define an async function

Define an async function using the `async` keyword.

```
async function myAsyncFunction() {
    // code to execute the async task
    // the return value will be wrapped in a Promise object automatically
    // error thrown in the async function will be wrapped in a rejected Promise object automatically

}
```

Note:

- The return value is wrapped in a Promise object automatically.
- The error thrown in the async function is wrapped in a rejected Promise object automatically.
- You don't have to explicitly call the `resolve` or `reject` function in the async function.

## Pattern to call the async function and await for the result

Resolve the Promise object of the async function using the `await` keyword.

- the `await` keyword can only be used inside an async function.

```
async function caller() {
    // call the async function
    const result = await myAsyncFunction();
    // use the result of the Promise object
}
```

## meme about async and await

Everyone in the chain becomes the Tinky Winky (the purple Teletubby) when the last one touches him (the async function).

- The await must be used in the async function.

## Example 13-8: Rewrite Example 13-6 using the async/await syntax

```
// call the longtimeTask function
console.log('Start the long running task');
// Use an Immediate Invoke Function Expression (IIFE) to call the function returning a promise
(async () => {
    let result = await longtimeTask();
    console.log('The result is: ', result);
})()
console.log('Please wait for the long running task to complete...');
```

The output of the above code is:

```
Start the long running task
== Enter the long task
== Exit the long task
Please wait for the long running task to complete...
The result is:  15.919520883194348
```

## Example 13-9: What's wrong with the following code?

```
// call the longtimeTask function
console.log('Start the long running task');

let result = await longtimeTask();
console.log('The result is', result);
console.log('Please wait for the long running task to complete...');
```

The above code will throw a syntax error because the `await` keyword can only be used inside an async function.

- Inside the async function, you write code to handle the resolved value or exception of the promise object.

```
SyntaxError: await is only valid in async functions and the top level bodies of modules
```

To fix the error, you can

- define a named function to wrap the code and call the function, or
- use an IIFE to call the function, or
- use the `then` method of the Promise object.

## Example 13-10: Fix the code in Example 13-9 by a named async function

```
console.log('Start the long running task');
// Define a named async function
async function runLongTimeTask(){
    try{
        let result = await longtimeTask();
        console.log('The result is', result);
    } catch(error){
        console.error('An error occurred:', error);
    }
}
// call the longtimeTask function
runLongTimeTask();
runLongTimeTask();
runLongTimeTask();
runLongTimeTask();
console.log('Please wait for the long running task to complete...');
```

Output:

```
'Start the long running task'
'== Enter the long task'
'== Exit the long task'
'== Enter the long task'
'== Exit the long task'
'== Enter the long task'
'== Exit the long task'
'== Enter the long task'
'== Exit the long task'
'Please wait for the long running task to complete...'
[ 'The result is', 54.37531724398479 ]
[ 'The result is', 22.829382717193592 ]
[ 'The result is', 67.3674835459125 ]
[ 'The result is', 74.8877783165409 ]
```

## 5 Handle the error in the async function

You can use the `try...catch` block to handle the error in the async function.

- the statement with the `await` keyword will throw an error if the promise object is rejected.

The pattern:

```
async function async_fun1() {
    try {
        // The async function that might throw an error
        const result = await async_fun2();
        ...
    } catch (error) {
      // catch and handle the error
        console.error('An error occurred:', error);
    }
}
```

## Example 13-12: Use the try...catch block to handle the error in the async function

```
async function myAsyncFunction() {
    try {
        // code to execute the async task
        const result = await longtimeTask();
        console.log('The result is', result);
    } catch (error) {
        console.error('An error occurred:', error);
    }
}
```

## 6 Use the async function with the Promise `then` method

You can use the async function with the `then` method of the Promise object.

- because the async function **always** returns a Promise object.

## The return value of the async function

Async functions always return a Promise object [2].

If the return value of an async function is not explicitly a promise, it will be implicitly wrapped in a promise.

For example, the code:

```
async function myAsyncFunction() {
    return 'Hello World';
}
```

is equivalent to:

```
function myAsyncFunction() {
    return Promise.resolve('Hello World');
}
```

## Use `then` method with the async function

you can use the `then` method of the Promise object with the async function.

- if you don't want to use the `await` keyword
  - because you don't want to create another async function to call the async function.
- Remember that the async function always returns a Promise object.

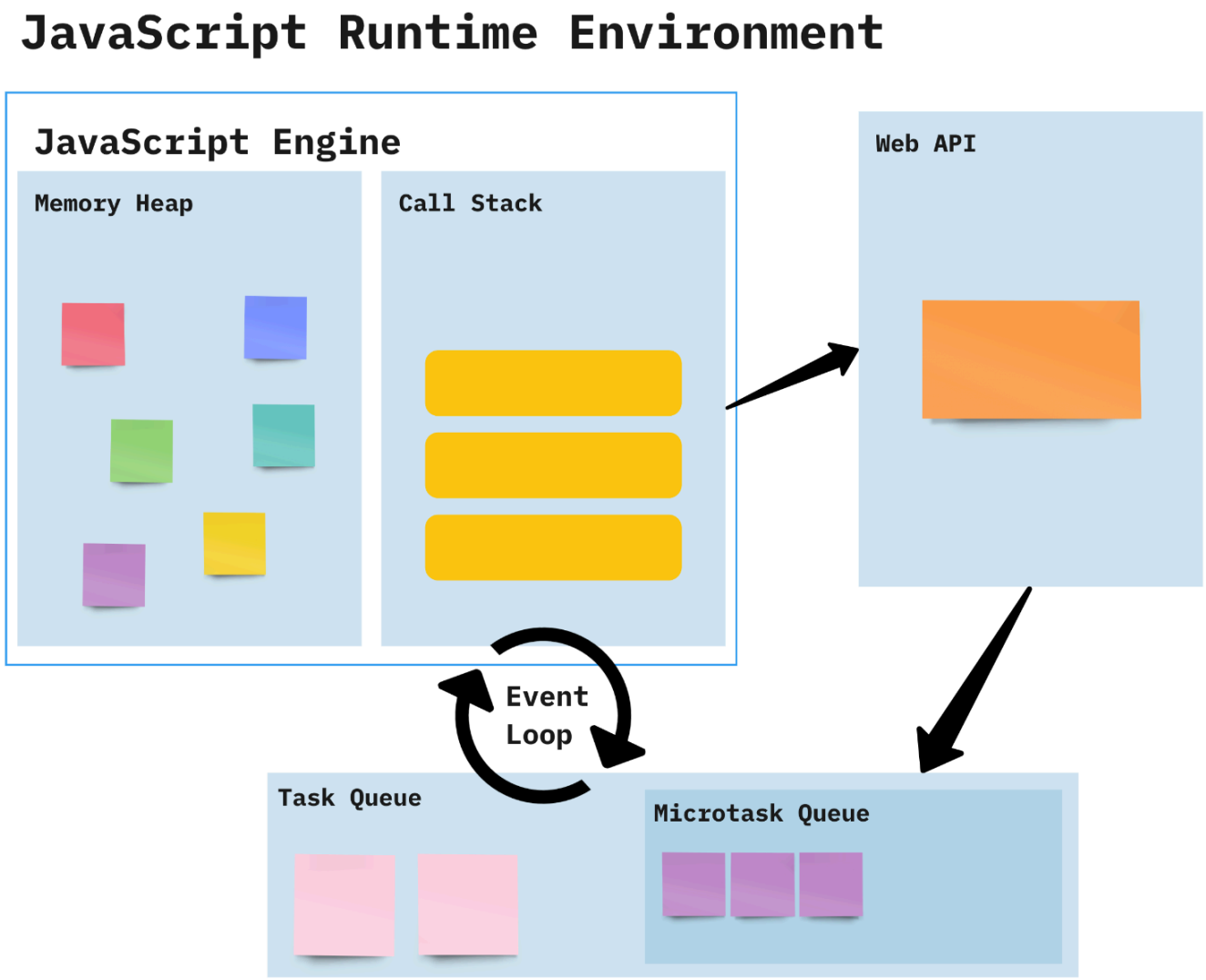
```
myAsyncFunction().then(value => console.log(value));
```

See ex_13_async_then.js for the complete code.

# 7 EVENT LOOP concurrency model in JS runtime environment

- JavaScript is a single-threaded language
    - JS can only execute one task at a time
- A thread in this context means a path of execution
- To handle asynchronous tasks, JS employs the **EVENT LOOP** concurrency model.

# The Event Loop concurrency model



## The interaction between the JS engine and the Web API

The browser is a multi-threaded environment.

- The Web API is a set of APIs provided by the browser
- The tasks invoked by the Web API are executed in the background (not in the JS engine).

The JS engine is a single-threaded environment.

- The JS engine can only execute one task at a time.
- The JS engine use the **Event Loop** to interact with the Web API.

## The structure of the JS runtime environment

JS runtime environment consists of the following components:

- Call Stack: LIFO, the code to be executed in the JS engine

Two queues to store the asynchronous tasks to be executed:

- Micro-task Queue: A queue to store the **promise-based function** (the promise handler).
  - Micro-tasks have higher priority than the Macro-tasks
  - represents the "tasks that need to be completed immediately but are asynchronous"
- Macro-task Queue (or Callback, or Task) Queue: A queue to store the **callback functions invoked by the Web APIs**
  - The tasks offloaded to the Web API

## How the Event Loop works

- Event Loop: A loop to check the Call Stack and the Callback Queue.

1.Run the tasks in the Call Stack first.

- The JS Engine runs tasks in the Call Stack first.

2.Move the tasks from the Micro task Queue to the Call Stack in the next.

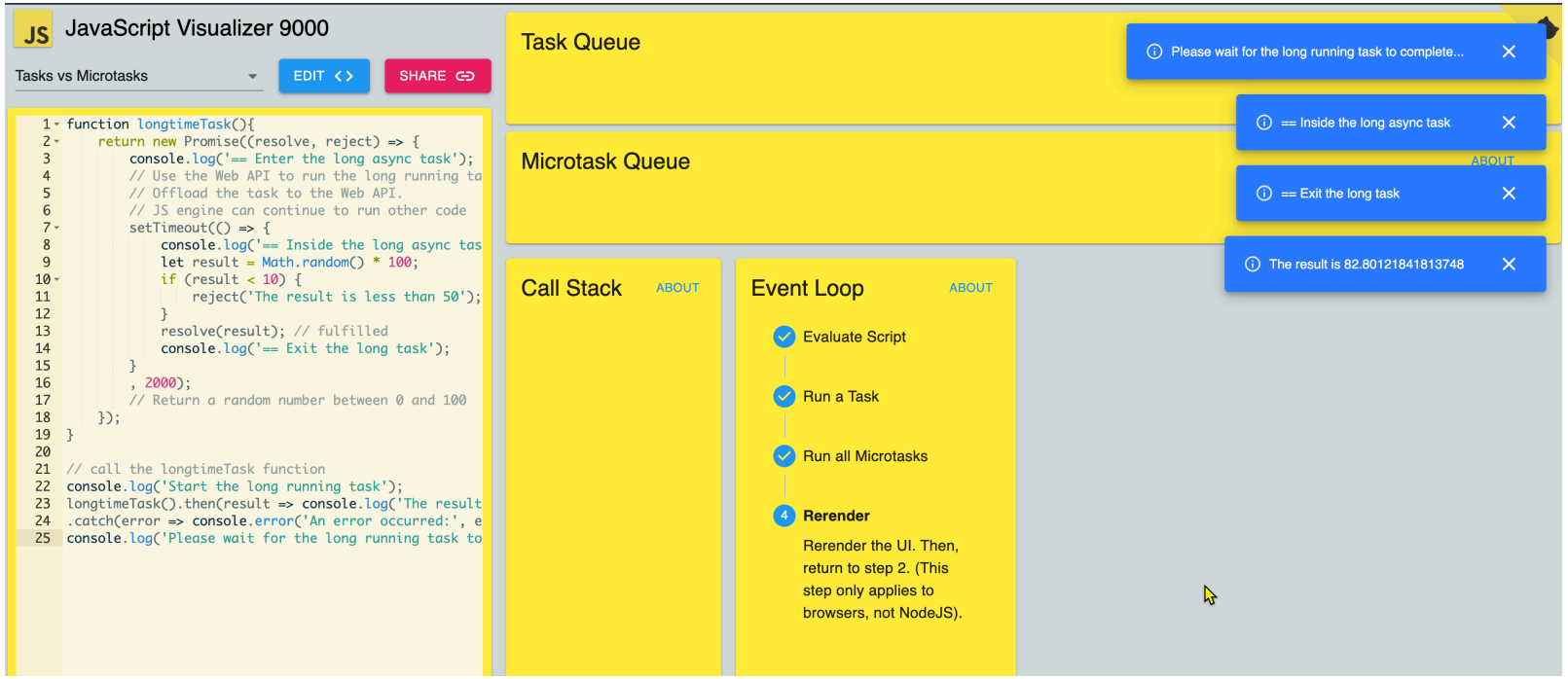
- If and only if the Call Stack is empty, the Event Loop will move the Micro-tasks from the queue to the Call Stack.
- **All** the tasks in the Micro-task Queue will be moved to the Call Stack when the Call Stack is empty.
- Micro-tasks have higher priority than the Macro-tasks in the Callback Queue.
- Micro-tasks queue store the **promise-based function** (the promise handler).

3.Move the tasks from Macro-task Queue to the Call Stack.

- If no tasks in the Micro-task Queue, the Event Loop will move the tasks from the Macro-task Queue to the Call Stack.
- Move **one task** from the Macro-task Queue at a Event Loop tick.
- Macro-task queue stores the timer-based tasks (setTimeout, setInterval) and the I/O tasks (e.g. fetch, XMLHttpRequest, etc.) and tasks that offloaded to the Web API.

# Example 13-11: Use JavaScript Visualizer 9000 to visualize the runtime process for the code in Example 13-7

JS Visualizer 9000 - An interactive JavaScript runtime model visualizer.




Note: The JS Visualizer 9000 cannot accept the async/await syntax.

## 8 Video and Interactive Tools to help you understand the JS runtime model

Lydia Hallie, 2024. JavaScript Visualized - Event Loop, Web APIs, (Micro)task Queue

JS Visualizer 9000 - An interactive JavaScript runtime model visualizer.

## Example: Analyze the async code

What is the output of the following code?

```
console.log('1');

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

Promise.resolve().then(() => {
  console.log('4');
});

console.log('5');
```

- Answer

## 9 Summary

- JavaScript is single-threaded but handles concurrency using the Event Loop model.
- Non-async functions can be wrapped in Promises to enable asynchronous behavior, but the code still runs on the main thread unless offloaded to Web APIs.
- The `async` and `await` keywords provide syntactic sugar for working with Promises, making asynchronous code easier to read and maintain.
- Async functions always return a Promise, and `await` can only be used inside async functions.
- The Event Loop manages the execution order of synchronous code, micro-tasks (Promise handlers), and macro-tasks (Web API callbacks).

- Micro-tasks have higher priority than macro-tasks and are executed before macro-tasks when the call stack is empty.
- Understanding the interaction between the call stack, micro-task queue, macro-task queue, and Web APIs is essential for writing efficient asynchronous JavaScript code.