Overview of JavaScript Execution
JavaScript operates as a single-threaded, non-blocking, asynchronous, concurrent programming language. Here's how each part of that description plays into synchronous and asynchronous behavior:

Single-Threaded: JavaScript executes code in a single sequence, meaning only one piece of code runs at a time.
Non-Blocking: JavaScript can handle tasks without waiting for one task to complete before starting another. This is achieved through asynchronous programming.
Asynchronous: Allows certain operations, like network requests, to happen independently of the main thread, so the rest of the code can continue executing.
Concurrent: Can handle multiple tasks seemingly at once by managing execution with non-blocking behavior.
Synchronous JavaScript
Function Execution Stack (Call Stack): When a function is called, it is added to the call stack. The stack keeps track of the current function and its calls. Functions execute in a last-in, first-out order.

Example:

javascriptCopy code
```
function f1() { console.log('f1'); }
function f2() { console.log('f2'); }
function f3() { console.log('f3'); }

f1(); // Logs 'f1'
f2(); // Logs 'f2'
f3(); // Logs 'f3'
```

Here, f1, f2, and f3 execute one after the other in sequence.
Asynchronous JavaScript
Browser APIs/Web APIs: Functions like setTimeout, fetch, and event handlers manage asynchronous operations. They use callback functions to

handle the results of asynchronous tasks.

Example with setTimeout:

javascriptCopy code
```
function printMe() { console.log('print me'); }
setTimeout(printMe, 2000); // Executes printMe after 2 seconds
console.log('This line runs first');
```

Output:

arduinoCopy code
```
This line runs first
print me
```

Callback Queue: When an asynchronous task completes, its callback is placed in the callback queue. The event loop periodically checks this queue and executes callbacks when the call stack is empty.
Event Loop: Manages the call stack and the callback queue. It ensures that callbacks from the callback queue are executed only when the call stack is clear.

Example:

javascriptCopy code
```
function f1() { console.log('f1'); }
function f2() { console.log('f2'); }
function main() {
```

```
setTimeout(f1, 0);
f2();
}
main();
```

Output:

Copy code

```
f2
f1
```

Despite f1 being scheduled with zero delay, f2 runs first because the callback queue is processed after the call stack is empty.

Promises and Job Queue

Promises: Represent asynchronous operations. They have three states: pending, fulfilled, and rejected. When a promise is fulfilled or rejected, its then or catch handlers are executed.

Example:

javascriptCopy code

```
const promise = new Promise((resolve) => {
resolve('Resolved');
});
promise.then(result => console.log(result));
```

Output:

Copy code
Resolved

Job Queue (Microtask Queue): Handles promises and has higher priority than the callback queue. Promises are placed in the job queue and executed before any callback from the callback queue.

Example:

javascriptCopy code
```
function main() {
setTimeout(() => console.log('timeout'), 0);
Promise.resolve().then(() => console.log('promise'));
}
main();
```

Output:

arduinoCopy code
```
promise
timeout
```

The promise handler runs before the setTimeout callback because the job queue has higher priority over the callback queue.
Summary
Synchronous Execution: Code runs line by line, managed by the call stack.
Asynchronous Execution: Managed by browser APIs and involves a callback queue and event loop.
Promises and Job Queue: Promises use the job queue for handling async results, giving them priority over callback functions.
This distinction helps understand how JavaScript handles operations that involve waiting or concurrency while maintaining a single-threaded

execution model.