17 – Asynchronous JavaScript

**JavaScript Engine vs Host Environment**

Execution of JavaScript requires the cooperation of two pieces of software. The JavaScript engine and the Host environment:

1) JavaScript Engine

* Understands the JavaScript code, parses, and runs it.
* Follows the official rules of the language (ECMAScript).
* Example: Chrome uses the V8 Engine

2) Host Environment

* The world around the JS Engine, which provides tools that the engine doesn’t have, such as the DOM (to interact with the webpage), setTimeout (for delays), console.log (to print output), and APIs
* Example: Browser, Node.js

**Agent Execution Model (How JS Works Internally)**

An agent is the environment inside the JavaScript engine that executes the code, step-by-step, in an isolated space. These agents handle three key things:

1) Heap (Memory Storage): A large, unstructured region of memory that stores objects and data.

2) Queue (Event Loop): Enables asynchronous programming in JavaScript. Follows first-in-first-out.

3) Stack (Call Stack): The region where the program is run and the Global Execution Context, as well as subsequent Function Execution Contexts, are created.

**Asynchronous Programming**

Asynchronous Programming is a technique that enables one’s program to start a potentially long-running task and still be responsive to other events in the code while running that task, rather than having to wait for that task to finish.

Many functions provided by the browser may take a long time to execute and thus, can be considered asynchronous:

* Making API requests using fetch().
* Accessing a user’s camera/microphone using getUserMedia().
* Requiring a specific amount of time to finish before executing a callback using setTimeout().

Event handlers are a form of asynchronous programming; you provide a function (the event handler) that will be called, not right away, but whenever the event happens. If “the event” is “the asynchronous code has completed”, then we can say that that event could be used to notify the caller about the result of the async operation.

**Callbacks**

A callback is just a function passed into another function with the expectation that it will be called at an appropriate time. Callbacks used to be the main way asynchronous functions were implemented in JavaScript.

Functionally, callback-based code is equivalent to promises and async/await keywords. However, it leads to Callback Hell, i.e., a situation where multiple callbacks for multiple asynchronous tasks need to be called to perform some operation.

Example:

function doStep1(init, callback) {

const result = init + 1;

callback(result);

}

function doStep2(init, callback) {

const result = init + 2;

callback(result);

}

function doStep3(init, callback) {

const result = init + 3;

callback(result);

}

function doOperation() {

doStep1(0, (result1) => {

doStep2(result1, (result2) => {

doStep3(result2, (result3) => {

console.log(`result: ${result3}`); // callback hell

});

});

});

}

doOperation();

**Promises**

The Promise object represents the eventual completion (or failure) of an asynchronous operation and its resulting value.

A Promise is a proxy for a value not necessarily known at the time it is created. Initially, a Promise is in the “pending” state. This allows asynchronous methods to return a synchronous-like result.

Once the asynchronous operation is completed, the promise is then in either the “fulfilled” state with a value or the “rejected” state with a message (error).

A promise is said to be “settled” if it is either fulfilled or rejected, but not pending.

**Chained Promises**

The Promise methods then(), catch(), and finally() are used to associate further action with a promise that becomes settled. The then() method takes two arguments; the first is a callback function for the case that a promise is fulfilled, while the second (optional) is a callback function for the rejected case. The catch() and finally() methods call then() internally and make error handling less verbose.

const myPromise = new Promise((resolve, reject) => {

setTimeout(() => {

resolve("foo");

}, 300);

});

myPromise

.then(handleFulfilledA, handleRejectedA)

.then(handleFulfilledB, handleRejectedB)

.then(handleFulfilledC, handleRejectedC);

In the above code, if myPromise is rejected, the first .then() will call handleRejectedA, which also returns a Promise (hypothetically). Then, if handleRejectedA completes normally (does not throw an error), the promise returned by the first then will be in a fulfilled state instead of staying rejected.

**Microtask Queue**

JavaScript is single-threaded. So, to handle tasks like promises, it uses the microtask queue to manage which task to execute next. The following tasks are added to the microtask queue:

1) The executor inside the new Promise(executorFn)

2) The functions passed into then(), catch(), and finally()

3) API fetch requests such as fetch().

Promise.resolve(1)

.then((value) => value + 1)

.then((value) => value \* 2);

In the above code, each then() doesn’t run immediately. The first then() handler is added to the queue only after the initial promise is resolved. Subsequently, the second then() handler is added to the queue after the first then() handler is resolved.

A Promise may also participate in multiple then() chains.

const promiseA = new Promise(myExecutorFunc);

const promiseB = promiseA.then(handleFulfilled1, handleRejected1);

const promiseC = promiseA.then(handleFulfilled2, handleRejected2);

In the above code, the fulfillment of promiseA will cause both handleFulfilled1 and handleFulfilled2 to be added to the Microtask Queue. As handleFulfilled1 was registered first, it will be executed first.

const promiseA = new Promise((resolve, reject) => {

resolve(777);

});

// At this point, "promiseA" is already settled.

promiseA.then((val) => console.log("asynchronous logging has val:", val));

console.log("immediate logging");

// produces output in this order:

// immediate logging

// asynchronous logging has val: 777

The above code shows that tasks added to the microtask queue will execute only after the program in the call stack finishes executing. This implies that an action for an already “settled” promise will occur only after the current synchronous code completes. This shows that promises are asynchronous in nature.

Promise.resolve() – Returns a Promise object that has been resolved with a value.

Promise.reject() – Returns a Promise object that has been rejected with an error message.