Week 1.5

Created	December 27, 2023 11:13 PM
Created by	A Aditya Kulkarni
Tags	Empty

Async functions vs sync functions

Synchronous JavaScript

- Synchronous means executing statements one after the other, which implies the next statement will wait for the previous one to complete before executing1.
- Synchronous JavaScript uses a function execution stack (or call stack) to track the current function in execution1.
- The call stack works in a last-in, first-out (LIFO) order, meaning the last function pushed into the stack is the first one to pop out when the function returns1.
- Synchronous JavaScript is simple and easy to follow, but it can also block the execution of the rest of the code until the current function is finished, which can lead to poor performance and unresponsive user interface2.

Example:

```
// Define three functions function f1() { console.log("f1"); } function f2() { console.log("f2"); } function f3() { console.log("f3"); } // Invoke the functions f1(); // Push f1 to the call stack and execute it f2(); // Push f2 to the call stack and execute it after f1 is popped out f3(); // Push f3 to the call stack and execute it after f2 is popped out // Output: f1, f2, f3
```

Asynchronous JavaScript

- Asynchronous means executing statements without waiting for the previous one to complete, which implies the next statement can execute even if the previous one is still being processed.
- Asynchronous JavaScript uses a callback queue (or task queue) to handle the functions that are deferred for later execution.
- The callback queue works in a first-in, first-out (FIFO) order, meaning the first function pushed into the queue is the first one to be executed when the call stack is empty.
- Asynchronous JavaScript is more complex and requires callbacks, promises, or async/await syntax, but it can also improve the performance and responsiveness of the code by allowing other operations to run in parallel.

Example:

```
// Define a function that takes a callback function f1(callback) { setTimeout(() => { console.log("f1"); callback(); }, 1000); // Wait for 1 second before executing the callback } // Define another function function f2() { console.log("f2"); } // Invoke the functions f1(f2); // Push f1 to the call stack and pass f2 as the callback // f1 is moved to the web API and f2 is pushed to the callback queue // The call stack is empty and the event loop checks the callback queue // f2 is moved to the call stack and executed // f1 is finished and moved back to the call stack and executed // Output: f2, f1
```

Common async functions

- Some common async functions are `setTimeout`, `fs.readFile`, and `fetch`3.

- `setTimeout` is a browser API that allows you to execute a function after a specified delay3.
- `fs.readFile` is a Node.js API that allows you to read a file from your filesystem3.
- `fetch` is a browser API that allows you to fetch some data from an API endpoint3.

Example:

```
// Use setTimeout to log a message after 2 seconds setTimeout(() => { consol
e.log("Hello after 2 seconds"); }, 2000); // Use fs.readFile to read a file a
nd log its contents const fs = require("fs"); // Import the fs module fs.read
File("test.txt", "utf8", (err, data) => { // Read the file asynchronously if
(err) { console.error(err); // Handle the error } else { console.log(data);
// Log the data } }); // Use fetch to get some data from an API and log it fe
tch("https://jsonplaceholder.typicode.com/todos/1") // Make a GET request to
the API .then((response) => response.json()) // Parse the response as JSON .t
hen((data) => console.log(data)) // Log the data .catch((error) => console.er
ror(error)); // Handle the error
```

Promises

- A promise is an object that represents the eventual completion or failure of an asynchronous operation4.
- A promise can be in one of three states: pending, fulfilled, or rejected4.
- A promise can be created using the `Promise` constructor, which takes a function as an argument. The function has two parameters: `resolve` and `reject`, which are functions that can be used to settle the promise4.
- A promise can be consumed using the `then` method, which takes two callbacks as arguments: one for the fulfillment case and one for the rejection case4.
- A promise can also be consumed using the `catch` method, which takes a callback as an argument for the rejection case4.
- A promise can also be consumed using the `finally` method, which takes a callback as an argument that will be executed regardless of the promise state4.

Example:

```
// Create a promise that resolves with "Hello" after 1 second const p = new P
romise((resolve, reject) => { setTimeout(() => { resolve("Hello"); }, 1000);
}); // Consume the promise using then, catch, and finally p.then((value) => {
// This will be executed if the promise is fulfilled console.log(value); // H
ello }) .catch((reason) => { // This will be executed if the promise is rejec
ted console.error(reason); }) .finally(() => { // This will be executed regar
dless of the promise state console.log("Done"); });
```

Async/await

- Async/await is a syntactic feature that allows you to write asynchronous code in a synchronous-like manner5.
- Async/await is based on promises, but it uses the `async` and `await` keywords to simplify the syntax5.
- The `async` keyword is used to declare an asynchronous function, which returns a promise implicitly5.
- The `await` keyword is used to pause the execution of an async function until a promise is settled, and then resume it with the promise value5.
- The `await` keyword can only be used inside an async function, otherwise it will cause a syntax error5.

Example:

```
// Declare an async function that returns "Hello" after 1 second async functi
on f() { const value = await new Promise((resolve, reject) => { setTimeout(()
=> { resolve("Hello"); }, 1000); }); return value; } // Consume the async fun
ction using then, catch, and finally f().then((value) => { // This will be ex
ecuted if the async function is fulfilled console.log(value); // Hello }) .ca
tch((reason) => { // This will be executed if the async function is rejected
console.error(reason); }) .finally(() => { // This will be executed regardles
s of the async function state console.log("Done"); });
```

Why even make it async?

- Making a function async can have several benefits, such as:
  - Improving the performance and responsiveness of the code by avoiding blocking operations
  - Handling multiple concurrent tasks more efficiently and elegantly
  - Simplifying the error handling and chaining of async operations
  - Writing cleaner and more readable code with async/await syntax
- However, making a function async can also have some drawbacks, such as:
  - Introducing more complexity and challenges in debugging and testing
  - Creating potential problems with memory leaks and race conditions
  - Requiring more careful design and planning of the code logic and flow
  - Adding more dependencies and compatibility issues with older browsers or environments

Example:

```
// A synchronous function that blocks the code for 3 seconds function syncFun
c() { const start = Date.now(); while (Date.now() - start < 3000) {} // Busy
waiting console.log("Sync func done"); } // An asynchronous function that doe
s not block the code for 3 seconds async function asyncFunc() { await new Pro
mise((resolve, reject) => { setTimeout(() => { resolve(); }, 3000); }); cons
ole.log("Async func done"); } // Invoke the functions and measure the time co
nsole.time("sync"); syncFunc(); // This will block the code for 3 seconds con
sole.timeEnd("sync"); // sync: 3000.123ms console.time("async"); asyncFunc();
// This will not block the code for 3 seconds console.timeEnd("async"); // as
ync: 0.456ms
```