I/O, Callbacks, Promises, and Async / Await

Software Development Bootcamp

How computers handle information and Node.js operations

Topic

Input and Output

What Is Input and Output

Input and Output (I/O) refers to the ways computers interact with the outside world.

- **Input**: Ways computers take in information (keyboards, mice, cameras, joysticks)
- **Output**: Ways computers send out information (text, graphics, sound)
- Together, input and output are referred to as **I/O**

Memory vs. I/O

- Computers perform computations and access memory in nanoseconds.
- I/O operations take milliseconds to seconds, causing Node.js (JavaScript) to pause the program for the CPU to perform other tasks.
- Node.js requires defining a function to run when the program is ready to proceed.

Using I/O Data in the Terminal

- `process.stdin`: Terminal input
- `.once(data...)`: Event listener waits for data
- `(input)`: uses data as a parameter
- `=> {console.log...}`: converts data to a string and prints it to the console

```
process.stdin.once('data',
(input) => {
  console.log(input.toString())
})
```

Topic

Event Listeners and Callback Functions

What Are Event Listeners?

Event listeners are functions that wait for a specific event to occur and then execute a block of code in response to that event.

- They “listen” for certain actions or changes in the program or user interface
- Common events include clicks, key presses, data input, and page loads

Why We Use Event Listeners

1. **Multi-tasking**: They let the program do other things while waiting for events
2. **Responding to Users**: They help make programs react when users do something
3. **Saving Power**: They only run code when needed, not all the time
4. **Tidy Code**: They keep event-handling separate from other code, making it easier to understand
5. **Quick Updates**: They help programs respond right away when something changes

What Are Callback Functions?

Callback functions are functions that are passed as arguments to other functions. They're "called back" (or executed) after the main function finishes its job

Key Points About Callback Functions

- They're just regular functions, but used in a special way
- They're passed as arguments to another function
- They run after the main function is done
- They can receive data from the main function

Callback Example

```
// This is the main function, it takes a parameter we've named callback
function mainFunction(callback) {
    console.log("Doing some work");
    callback();
}
// This is the callback function, it will be passed as an argument to
mainFunction
function callbackFunction() {
    console.log("I'm the callback!");
}
// calling mainFunction with the argument callbackFunction
mainFunction(callbackFunction);
```

Why We Use Callback Functions

1. **Waiting for Tasks**: They help manage tasks that take time, like reading files or getting data from the internet
2. **Customizing Actions**: They let us change what a function does without changing its main code
3. **Handling Events**: They're great for responding to things that happen, like button clicks or data arrival
4. **Step-by-Step Tasks**: They help break big jobs into smaller steps that happen in the right order
5. **Keeping Code Flexible**: They make our code more adaptable to different situations

Topic

Promises, and Async / Await

What Is Asynchronous Code?

Asynchronous code is a way of writing programs that can do multiple things at once without waiting for each task to finish before starting the next one.

Key Points About Asynchronous Code

- It's like multitasking in everyday life - you can start the laundry and then cook dinner while the clothes are washing
- Especially useful for operations that might take a while, like reading files or making network requests
- In JavaScript, asynchronous code is commonly handled using callbacks, promises, or async/await

Connecting I/O and Asynchronous Code

- I/O operations often take a long time compared to other computer tasks
- Waiting for I/O can slow down a program if not handled properly
- Asynchronous code helps manage I/O operations efficiently

What Are Promises?

Promises are a way to handle tasks that take time in JavaScript. They're like a "promise" for a result in the future.

Key Points About Promises

- They represent a value that might not be available right away
- They can be in one of three states: **pending, resolved, rejected**
- They make it easier to deal with tasks that happen one after another

Promise Example

`myPromise` waits for 2 seconds, then resolves with “Done!”

```
let myPromise = new Promise((resolve, reject) => {
   setTimeout(() => {
     resolve("Done!");
   }, 2000);
 });
  myPromise.then((result) => {
   console.log(result);
// Prints "Done!" after 2 seconds
 });
```

The `.then()` Method

- Used to specify what should happen when the Promise is fulfilled (resolved successfully)
- It takes a function as an argument, which will be called with the Promise's resolved value
- You can chain multiple **.then()** calls to handle a sequence of asynchronous operations

What Is Async / Await?

Async/Await is a way to work with promises that makes asynchronous code look and behave more like synchronous code.

- `async` is used to declare a function that will work with asynchronous operations
- `await` is used inside an async function to wait for a promise to resolve

Async / Await Example

- We define a **`delay`** function that returns a promise that resolves after a given number of milliseconds
- The **`example`** function is declared as **`async`**
- Inside **`example`**, we use **`await`** to pause execution at each **`delay`** call
- The **`console.log`** statements show the order of execution.

```
function delay(ms) {
 return new Promise((resolve) =>
setTimeout(resolve, ms));
}
async function example() {
 console.log("Start of async function");
 await delay(2000);
 console.log("After 2 seconds");
 await delay(1000);
 console.log("After another 1 second");
 console.log("End of async function");
}
console.log("Before calling async
function");
example();
console.log("After calling async function");
```

Why We Use Promises And Async/Await

1. **Better Error Handling**: They make it easier to catch and handle errors in asynchronous code
2. **Avoid Callbacks**: They help write cleaner code when dealing with multiple asynchronous operations
3. **Easy to Read**: Especially with async/await, the code looks more like normal, step-by-step instructions
4. **Control Flow**: They give better control over the order of asynchronous operations
5. **Built-in Support**: Modern JavaScript and many libraries use promises, making them a standard way to handle asynchronous tasks

Exercise

Hello Frenemy I/O