the shortcut

Fullstack Web Development Tutorial Lesson 16

Today's lesson will cover

- Callbacks
- Promise basics

the shortcut

JavaScript fundamentals

Callbacks

- Many functions are provided by JavaScript host environments that allow you to schedule *asynchronous* actions. In other words, actions that we initiate now, but they finish later.
- “Callback-based” style of asynchronous programming. A function that does something asynchronously should provide a `callback` argument where we put the function to run after it’s complete.
- “Error-first callback” style convention is:
  - The first argument of the `callback` is reserved for an error if it occurs. Then `callback(err)` is called.
  - The second argument (and the next ones if needed) are for the successful result. Then `callback(null, result1, result2…)` is called.
  - So the single `callback` function is used both for reporting errors and passing back results.
- As calls become more nested, the code becomes deeper and increasingly more difficult to manage, especially if we have real code instead of `...` that may include more loops, conditional statements and so on.
- That’s sometimes called “callback hell” or “pyramid of doom.” The “pyramid” of nested calls grows to the right with every asynchronous action. Soon it spirals out of control. So this way of coding isn’t very good.

Promise

- A *promise* is a special JavaScript object that links the "producing code" and the "consuming code" together. The "producing code" takes whatever time it needs to produce the promised result, and the "promise" makes that result available to all of the subscribed code when it's ready.
- Syntax of a promise object

  -
    ```
    let promise = new Promise(function(resolve, reject) {
    // executor - the producing code
    });
    ```
- When the executor obtains the result, be it soon or late, doesn't matter, it should call one of these callbacks:
  - `resolve(value)` — if the job finished successfully, with result `value`.
  - `reject(error)` — if an error occurred, `error` is the error object.
- So to summarize: the executor runs automatically and attempts to perform a job. When it is finished with the attempt it calls `resolve` if it was successful or `reject` if there was an error.
- The `promise` object returned by the `new Promise` constructor has these internal properties:
  - `state` — initially `"pending"`, then changes to either `"fulfilled"` when `resolve` is called or `"rejected"` when `reject` is called.
  - `result` — initially `undefined`, then changes to `value` when `resolve(value)` called or `error` when `reject(error)` is called.

Promise consumers: then, catch, finally

- Consuming functions can be registered (subscribed) using methods `.then`, `.catch` and `.finally`.
- The most important, fundamental one is `.then`.
  - The syntax is:

```
Promise
     .then(
          function(result) { /* handle a successful result */ },
          function(error) { /* handle an error */ }
     );
```

  - The first argument of `.then` is a function that runs when the promise is resolved, and receives the result.
  - The second argument of `.then` is a function that runs when the promise is rejected, and receives the error.
- The call `.catch(f)` is a complete analog of `.then(null, f)`, it's just a shorthand.
- The call `.finally(f)` is similar to `.then(f, f)` in the sense that `f` always runs when the promise is settled: be it resolve or reject.

Promises vs Callbacks

Promises	Callbacks
Promises allow us to do things in the natural order. First, we run `loadScript(script)`, and `.then` we write what to do with the result.	We must have a `callback` function at our disposal when calling `loadScript(script, callback)`. In other words, we must know what to do with the result *before* `loadScript` is called.
We can call `.then` on a Promise as many times as we want. Each time, we're adding a new "fan", a new subscribing function, to the "subscription list".	There can be only one callback.

Exercise

- Create a function called `greetHi` which takes name as a parameter and logs “Hi GIVENNAME”
- Create another function called `greetBye` which takes name as a parameter and logs “Bye GIVENNAME”
- Write another function called `userInfo` which takes `firstName`, `lastName` and a `callback` function as parameters. The function stores `fullName` based on given `firstName` and `lastName`, and uses the `fullName` as parameter for the `callback` function.
- Call `userInfo` function with choice of names as parameter and `greetHi` function as the callback function
- Call `userInfo` function with choice of names as parameter and `greetBye` function as the callback function

Exercise

- The built-in function `setTimeout` uses callbacks. Create a promise-based alternative.
- The function `delay(ms)` should return a promise. That promise should resolve after `ms` milliseconds, so that we can add `.then` to it, like this:

-
  ```
  function delay(ms) {
   // your code
  }
  delay(3000).then(() => console.log('runs after 3 seconds'));
  ```

Exercise

- Add the necessary pieces to fix the promise and the then function.
- It should resolve to a message on console success!.
- Re-assign the result to the settled value inside the then function.

```
let result = ""

let promise = new Promise(() => {
})

promise.then()
```

Exercise

- Async operations don't always go as planned. When errors creep up we need to know how to handle them. We can pass the **reject** callback to our **executor** function to pass errors to our promise.

  -
    ```
    let promise = new Promise( (resolve, reject) => {
      setTimeout(( ) => {
        /* something went wrong */
         reject('oops!')
     }, 1000)
    })
    ```
  - You can pass **Error** objects as well. Here we pass a simple string `"oops!"`.
- Task: Reject the promise with the simple string `"It's not a dog!"`.

```
let promise = new Promise( (resolve) => {
    let animal = "cat"
  setTimeout(() => {
        if(animal === "dog") {
              resolve("It's a dog!")
        }
        if(animal !== "dog") {
              /* need something here, you might also need to pass
              something else besides the resolve callback */
        }
  }, 1000)
})
```

the shortcut

Self Study Assignments

To Dos

- Create a game of Rock, Paper and Scissors using JS which works on console, or with interactive UI using HTML, CSS and JS however you prefer ***(If you are working on your own project where you are using JS already, feel free to ignore this task but please share the project update with Lena.)***
- Continue freecodecamp (FCC) Javascript. Ideally finish before we resume after summer.
- Continue with FCC HTML, CSS lessons. Ideally finish all the lessons by end of this month.
- If you believe FCC exercises aren’t the best for you if you are quite advanced already, please start working on your own project and reach out to mentors for help if needed.