Asynchronous Programming (1)

Asynchronus means we are dependant on the user or some other task to finish.

- JS has synchronus execution - code is executed line-by-line
- on the real world web however, we often times need to wait or pause whilst waiting a reponse etc etc before execution
- this pause **must not** stop the rest of our program from executing
- therefore we **need** asynchronus execution as well, so JS has something called **callbacks and promises**

Callbacks

- JS is a language which has **first-class functions**
  - this means we can pass functions as arguments to other functions
- these functions are then “called back” (executed) at a later stage
- **Why Callbacks?** JavaScript is single-threaded, meaning it can only execute one task at a time. Asynchronous operations, like making network requests or reading files, don't block the main thread. Callbacks allow you to specify what code to run when the asynchronous operation finishes.

Callback Hell

While callbacks are useful, nesting them deeply can lead to complex and unreadable code, often referred to as "callback hell" or the "pyramid of doom."

```
getUserData(123, function(data) {
  console.log("User Data:", data);
```

```
  getOrderHistory(data.userId, function(orders) {
    console.log("User Orders:", orders);

    for (const order of orders) {
      getOrderDetails(order.id, function(details) {
        console.log("Order Details:", details);
      });
    }
  });
});
```

Problems:

- The code becomes deeply nested, making it hard to follow the logic flow.
- Debugging issues can be challenging.
- Error handling becomes cumbersome within nested callbacks.
- you may not have written every function, so you might not fully understand how the function you are passing your function to works - this could lead to your function never executing due to a bug/misundertsanding of the other function
- callbacks can lead to a loss of control of our code
- callbacks might not do what we want out code to do (we lost trust in the code)

Alternatives to Callback Hell:

- **Promises:** Introduced in ES6, Promises offer a cleaner way to handle asynchronous operations. They provide a more structured approach for dealing with success and failure scenarios.
- **Async/Await (ES7):** Building on Promises, async/await syntax makes asynchronous code look more synchronous, improving readability.

In essence, callbacks are essential for asynchronous programming, but use them judiciously. Consider Promises and async/await for better code

organization as your application grows in complexity.

Promises

- the Promise Object represents the eventual completion or failure of an asynchronus operation and the resulting value

Concept: A Promise is an object that represents the eventual completion (or failure) of an asynchronous operation. It has three states:

- **Pending**: Initial state, the operation hasn't finished yet.
- **Fulfilled**: The operation succeeded, and a result is available.
- **Rejected:** The operation failed, and an error is available.

Creating Promises: You use the `Promise` constructor to create a Promise object. It takes an executor function as an argument. This executor function has two parameters:

- `resolve`: A function to call when the operation succeeds, passing a value as an argument.
- `reject`: A function to call when the operation fails, passing an error object as an argument.

Example:

```
function getUserData(userId) {
  return new Promise((resolve, reject) => {
    // Simulate asynchronous data fetching (like a network reque
    setTimeout(() => {
      const userData = { name: "Alice", age: 30 };
      resolve(userData); // Operation successful, call resolve v
    }, 1000); // Simulate a delay of 1 second
  });
}
```

Consuming Promises: You use the `then` and `catch` methods on a Promise object to handle its eventual state.

```
getUserData(123)
  .then(data => {
    console.log("User Data:", data);
    return getOrderHistory(data.userId); // Chain to another pr
  })
  .then(orders => {
    console.log("User Orders:", orders);
  })
  .catch(error => {
    console.error("Error:", error);
  });
```

Explanation:

- `getUserData` returns a Promise object.
- We call `then` on the Promise, providing a callback function for the successful case.
- Inside the `then` callback, we can access the resolved data and potentially chain to another Promise (like `getOrderHistory`).
- We can use additional `.then` calls for further actions after successful resolutions.
- The final `.catch` method acts as a central error handler for any rejections within the Promise chain.

Benefits of Promises:

- **Improved Readability:** Promise chains provide a clearer flow of logic compared to nested callbacks.
- **Error Handling:** A single `.catch` can handle errors throughout the chain.
- **Chaining:** Promises allow you to chain asynchronous operations more elegantly.

Key Points:

- Promises provide a more structured and manageable way to work with asynchronous code.
- They are a fundamental building block for many asynchronous operations in JavaScript.
- Consider using async/await (built on top of Promises) for even cleaner asynchronous code

Rejected: