LAB JS3: Asynchronous JavaScript

Before you start this lab - fork and clone this repo Day3 starter code

Part A: Constructing Objects with Classes

1. Classes

- Concept: Classes in JS are an ES6 feature that was released in 2015 and allowed developers to create a structure template for an object and use it to instantiate new instances of the object.
- Business Use Case: It makes creating records on a system more templated so when we add, edit, and delete records, they're added and manipulated in the same way using OO style of coding

Lab Challenge1

Create a class that is called Employee and follow the steps in the code comments on the JS file. Paste your code from the script.js below

```
class Employee {
  constructor(firstName, lastName, email, birthDate) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.email = email;
    this.birthDate = birthDate;
  }

    getEmployeeInfo() {
    return `${this.firstName} ${this.lastName} (${this.email}) - DOB: ${this.birthDate}`;
  }

    editEmployee(newEmail) {
    this.email = newEmail;
  }

    static addEmployee(fName, lName, email, dob) {
    return new Employee(fName, lName, email, dob);
  }
}

const suman = new Employee('Suman', 'Patra', 'sumanpatra131@gmail.com',
'12-03-2000');
console.log("Single Employee:", suman);

const empArr = [
  new Employee('John', 'Doe', 'johndoe@example.com', '31-12-1999'),
  new Employee('Jane', 'Smith', 'janesmith@example.com', '01-09-2015'),
  suman
];
```

```
suman.editEmployee('new.email@gmail.com');
const newHire = Employee.addEmployee('Alice', 'Wonder', 'alice@land.com',
'05-05-1995');
empArr.push(newHire);

function displayEmployees() {
  const tableBody = document.querySelector("#employeeTable tbody");
    tableBody.innerHTML = "";

  empArr.forEach(emp => {
    let row = `<tr>
      <td>${emp.firstName}</td>
      <td>${emp.lastName}</td>
      <td>${emp.email}</td>
      <td>${emp.birthDate}</td>
    </tr>`;
    tableBody.innerHTML += row;
  });
}

displayEmployees();
```

Understanding Callbacks, Promises, and Async/Await

Introduction

In this part of the lab, you'll learn how JavaScript handles asynchronous operations by working with examples and a real API. We'll fetch data from a free sample data API (https://jsonplaceholder.typicode.com/todos) and see how different approaches work.

First off…why We Can't Use Synchronous Code?

Example 1.1: Attempting Synchronous Code (This Won't Work!)

Open your browser's Developer Console (F12) and paste this code:

```
function getTodosSynchronous() {
  console.log("1. Starting to fetch todos...");
  const reply =  fetch('https://jsonplaceholder.typicode.com/todos');
  console.log("2. Response:", reply);
  console.log("3. This runs immediately, BEFORE data arrives!");
  return reply;
}

const result = getTodosSynchronous();

console.log("4. Result:", result);

console.log("5. We can't access the actual todo data this way!");
```

What happens:

- fetch() returns a Promise immediately
- JavaScript doesn't wait for the network request to complete
- You get a Promise object, not the actual data
- The code continues running before the data arrives

Why This Doesn't Work:

Network requests take time (milliseconds to seconds). JavaScript is single-threaded, so if it waited synchronously for every network request, your entire browser would freeze! That's why we need asynchronous code.

Callbacks (The Old Way)

Concept: Functions passed to other functions to execute after completing a task.
Business Use Case: Processing order data before sending confirmation emails.

Example to run in browser:

```
function fetchData(callback) {
  setTimeout(() => {
    const data = { name: 'John', age: 30 };
    callback(data);
  }, 2000);
}

function displayData(data) {
  console.log(`Name: ${data.name}, Age: ${data.age}`);
}

fetchData(displayData);
```

Demo: Order Processing System using Callback and arrow function

```
function processOrder(orderId, callback) {
  console.log(`Processing order #${orderId}...`);

  setTimeout(() => {
    callback(`Order #${orderId} processed successfully`);
  }, 2000);
}

// Usage
processOrder(101, (confirmation) => {
  console.log('Email sent:', confirmation);
});
```

Callbacks (API Example)

Callbacks are functions passed as arguments to be executed when an operation completes.

Example 2.1: Using Callbacks

```
function getTodosWithCallback(callback) {
    console.log("1. Starting to fetch todos...");

    fetch('https://jsonplaceholder.typicode.com/todos')
        .then(response => response.json())
        .then(data => {
            console.log("2. Response received and parsed!");
            callback(null, data); //Success: error is null, data is 2nd arg
        })
        .catch(error => {
            console.log("2. Error occurred!");
            callback(error, null); //Error: pass error as first arg, null data
        });

    console.log("3. Fetch initiated (but not complete yet)...");
}

// Usage with callback function
function displayData(error, todos) {
    if (error) {
        console.error("4. Error:", error);
        return;
    };

    console.log("4. Result (All):", todos);
    console.log("4. Result (first 3):", todos.slice(0,3));
    console.log("5. Now we CAN access the actual todo data!");
    console.log("6. First todo:", todos[0]);
};

getTodosWithCallback(displayData);
```

Example 2.2: Callback Hell

This demonstrates "callback hell" or "pyramid of doom" - when you need to make multiple sequential requests:

```
function getUserTodosCallback(userId, callback) {
    fetch(`https://jsonplaceholder.typicode.com/users/${userId}`)
        .then(response => response.json())
        .then(user => {
```

```
            console.log("User:", user.name);

            // First level of nesting - Find count of todos
            fetch(`https://jsonplaceholder.typicode.com/todos?userId=${userId}`)
                .then(response => response.json())
                .then(todos => {
                    console.log(`${user.name} has ${todos.length} todos`);

                    // Second level of nesting - Go get first to do for this user
                    if (todos.length > 0) {
                        fetch(`https://jsonplaceholder.typicode.com/todos/${todos[0].id}`)
                            .then(response => response.json())
                            .then(todoDetail => {

                                // Third level of nesting - return an object with user, all
todos, and firstToDo -- deeply nested!
                                callback(null, {
                                    user: user,
                                    todos: todos,
                                    firstTodo: todoDetail
                                });
                            })
                            .catch(error => callback(error, null));
                    }
                })
                .catch(error => callback(error, null));
        })
        .catch(error => callback(error, null));
}
// Usage
getUserTodosCallback(1, function(error, result) {
    if (error) {
        console.error("Error:", error);
        return;
    }
    console.log("Result:", result);
});
```

Why This Is "Callback Hell"

Callback Hell (also called the "Pyramid of Doom") refers to deeply nested callback functions that make code hard to read, maintain, and debug. Here's why this example demonstrates it:

Problems with this code:

1. **Deep Nesting (3 levels):** Each API call is nested inside the previous one, creating a "pyramid" shape that shifts the code further and further to the right
2. **Hard to Read:** You have to mentally track multiple levels of indentation to understand the flow
3. **Repetitive Error Handling:** Notice we have `.catch(error => callback(error, null))` repeated three times - one for each level
4. **Difficult to Modify:** Want to add another API call? You'd need to add another level of nesting
5. **Variable Scope Issues:** The `user` variable needs to be accessible all the way down in the innermost callback, making it harder to track

Visual representation of the nesting:

```
fetch user
└─ then parse
   └─ fetch todos
      └─ then parse
         └─ if todos exist
```

```
└─ fetch first todo
  └─ then parse
    └─ finally call callback with result
```

Part 3: Promises (The Better Way introduced with ES6)

A JavaScript promise is an object. It is an instance of the built-in Promise class and serves as a placeholder for the eventual result or failure of an asynchronous operation.

Key characteristics of a promise object:

- **State**: A promise object has an internal state, which can be one of three values:
  - **`pending`**: Initial state; the operation is still in progress.
  - **`fulfilled`**: The operation completed successfully, and the promise now has a resulting value.
  - **`rejected`**: The operation failed, and the promise now holds an error or "reason" for the failure.
- **Handlers**: You attach handler functions (callbacks) to the promise object using its methods, such as .then(), .catch(), and .finally(). These handlers are executed when the promise becomes settled, allowing you to work with the eventual value or error without blocking the main program thread.

In essence, the promise object provides a structured and manageable way to handle asynchronous code, moving away from "callback hell" toward more readable, chainable code.

Demo: Order Processing with Promises

```
function processOrder(orderId) {
  console.log(`Processing order #${orderId}...`);
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve(`Order #${orderId} processed successfully`);
    }, 2000); // Simulate 2-sec processing time
  });
}

// Usage
processOrder(101).then((confirmation) => {
  console.log('Email sent:', confirmation);
});
```

Example 3.1: Basic Promise (API Call)

```
//////////////////promises
function getTodosWithPromise() {
  console.log("Starting fetch with Promise...");

  return fetch("https://jsonplaceholder.typicode.com/todos")
    .then((response) => {
      console.log("Response received");
      return response.json();
    })
    .then((data) => {
      console.log("Data parsed");
      return data;
    })
    .catch((error) => {
      console.error("Error occurred:", error);
      throw error;
    });
}
```

```
// Usage:
getTodosWithPromise()
  .then((todos) => {
    console.log(`Received ${todos.length} todos`);
    console.log("First todo:", todos[0]);
  })
  .catch((error) => {
    console.error("Failed to get todos:", error);
  });
```

Example 3.2: Chaining Promises (No More Callback Hell!)

```
///////chaining promises
function getUserTodosPromise(userId) {
  let userData;
  return fetch(`https://jsonplaceholder.typicode.com/users/${userId}`)
    .then((response) => response.json())
    .then((user) => {
      console.log("User:", user.name);
      userData = user;
      return fetch(
        `https://jsonplaceholder.typicode.com/todos?userId=${userId}`,
      );
    })
    .then((response) => response.json())
    .then((todos) => {
      console.log(`Found ${todos.length} todos`);
      return { user: userData, todos: todos };
    })
    .catch((error) => {
      console.error("Error in chain:", error);
      throw error;
    });
}

// Usage: //
getUserTodosPromise(1)
  .then((result) => {
    console.log("Final result:", result);
  })
  .catch((error) => {
    console.error("Failed:", error);
  });
```

Part 4: Async/Await (The Modern Way)

Async/await makes asynchronous code look and behave more like synchronous code.

Demo: Order Processing using Async/Await (from ES8)

```
function processOrder(orderId) {
  console.log(`Processing order #${orderId}...`);
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve(`Order #${orderId} processed successfully`);
    }, 2000); // Simulate 2-sec processing time
  });
```

```
}

// Usage
async function handleOrder() {
  const confirmation = await processOrder(101);
  console.log('Email sent:', confirmation);
}

handleOrder();
```

Example 4.1: Basic Async/Await with API Call

```
/////async wait basic
async function getTodosAsync() {
  console.log("Starting fetch with async/await...");
  try {
    const response = await fetch("https://jsonplaceholder.typicode.com/todos");
    console.log("Response received");
    const data = await response.json();
    console.log("Data parsed");
    return data;
  } catch (error) {
    console.error("Error occurred:", error);
    throw error;
  }
}

// Usage: //
async function runExample() {
  try {
    const todos = await getTodosAsync();
    console.log(`Received ${todos.length} todos`);
    console.log("First todo:", todos[0]);
  } catch (error) {
    console.error("Failed to get todos:", error);
  }
}

runExample();
```

Example 4.2: Sequential Requests with Async/Await

```
////////////example 4.2
async function getUserTodosAsync(userId) {
  try {
    const userResponse = await fetch(
      `https://jsonplaceholder.typicode.com/users/${userId}`,
    );
    const user = await userResponse.json();
      console.log("User:", user.name);

    const todosResponse = await fetch(
      `https://jsonplaceholder.typicode.com/todos?userId=${userId}`,
    );
    const todos = await todosResponse.json();
      console.log(`Found ${todos.length} todos`);

    return { user: user, todos: todos };

  } catch (error) {
    console.error("Error:", error);
    throw error;
  }
}

// Usage: //
async function runUserExample() {
  try {
    const result = await getUserTodosAsync(1);
    console.log("Final result:", result);
  } catch (error) {
    console.error("Failed:", error);
  }
}
runUserExample();
```

Lab Challenge2 - Async/Await

Write an async function getTodoById(id) that fetches a specific todo by ID.

```
async function getTodoById(id) {
try {
const response = await fetch(`https://jsonplaceholder.typicode.com/todos/${id}`);
if (!response.ok) {
throw new Error(`HTTP error! status: ${response.status}`);
}
const todo = await response.json();
return todo;
}
catch (error) {
console.error("Error fetching todo:", error);
throw error;
}
  // TODO: Implement this using async/await
}

// Test it:
async function testGetTodo() {
  try {
    const todo = await getTodoById(1);
    console.log("Todo:", todo);
  } catch (error) {
    console.error(error);
  }
}

testGetTodo();
```

Key Takeaways

- Synchronous code doesn't work for network requests because JavaScript is single-threaded
- Callbacks work but can lead to "callback hell" with nested operations
- Promises provide better control flow and chaining capabilities
- Async/Await makes asynchronous code look synchronous and is easier to read
- All three approaches are still used in modern JavaScript, but async/await is preferred for new code

How to Run This Lab

1. Open your browser (Chrome, Firefox, Edge, or Safari)
2. Press F12 to open Developer Tools (or right-click anywhere and select 'Inspect')
3. Click on the 'Console' tab
4. Copy and paste each code example from this document into the console
5. Press Enter to run the code
6. Observe the console output to see how each approach works
7. Complete Lab challenges 1 and 2

Happy coding!

Appendix: How to handle many requests (Just for reference)

Example 3.3: Promise.all (Multiple Requests in Parallel)

```
function getMultipleUsersTodos() {
  const user1Promise = fetch(
    "https://jsonplaceholder.typicode.com/todos?userId=1",
  ).then((response) => response.json());
  const user2Promise = fetch(
    "https://jsonplaceholder.typicode.com/todos?userId=2",
  ).then((response) => response.json());
  const user3Promise = fetch(
    "https://jsonplaceholder.typicode.com/todos?userId=3",
  ).then((response) => response.json());
  return Promise.all([user1Promise, user2Promise, user3Promise]).then(
    (results) => {
      console.log("User 1 todos:", results[0].length);
      console.log("User 2 todos:", results[1].length);
      console.log("User 3 todos:", results[2].length);
      return results;
    },
  );
}
// Usage:
getMultipleUsersTodos()
  .then((allTodos) => {
    console.log("All requests completed!");
  })
  .catch((error) => {
    console.error("At least one request failed:", error);
  });
```

Example 4.3: Parallel Requests with Async/Await

```
//////4.3
async function getMultipleUsersTodosAsync() {
  try {
    const user1Promise = fetch(
      "https://jsonplaceholder.typicode.com/todos?userId=1",
    ).then((r) => r.json());

    const user2Promise = fetch(
      "https://jsonplaceholder.typicode.com/todos?userId=2",
    ).then((r) => r.json());

    const user3Promise = fetch(
```

```
      "https://jsonplaceholder.typicode.com/todos?userId=3",
    ).then((r) => r.json());

    const [user1Todos, user2Todos, user3Todos] = await Promise.all([
      user1Promise,
      user2Promise,
      user3Promise,
    ]);

    console.log("User 1 todos:", user1Todos.length);
    console.log("User 2 todos:", user2Todos.length);
    console.log("User 3 todos:", user3Todos.length);
    return [user1Todos, user2Todos, user3Todos];
  } catch (error) {
    console.error("Error:", error);
    throw error;
  }
}
// Usage: //
async function runParallelExample() {
  const results = await getMultipleUsersTodosAsync();
  console.log("All requests completed!");
}
runParallelExample();
```