Day 1 Trainee Tasks

Learnings covered:

- Learned JavaScript characteristics (dynamic, OOP, FP, syntax differences) and importance (ubiquity, browser support).
- Covered non-blocking, event-driven architecture and async/sync concepts.
- Explored comments and variable types (`var`, `let`, `const`).
- Learned about hoisting.
- Understood `null` and `undefined` as primitive types and their relations to booleans and numbers.

JavaScript Characteristics

1. **JavaScript is Dynamic**:
    - JavaScript is considered a **dynamic language** because it allows you to:
        - **Modify objects at runtime**: You can add, change, or remove properties from objects while the program is running.
        - **Dynamic typing**: Variables in JavaScript are not bound to a specific type, meaning you can reassign variables to different types. For example:

```
let x = 42;  // Initially a number
x = "Hello"; // Now it's a string
```

Dynamic data structures: You can dynamically modify arrays, objects, and other data structures.

JavaScript is Object-Oriented and Functional:

- **Object-Oriented Programming (OOP)**:
    - JavaScript supports **object-oriented programming** using objects and prototypes (rather than classical inheritance like in languages such as Java).
    - You can create objects with properties and methods, and use **prototypal inheritance** to extend objects and share behavior between them.

```
const person = {
  name: 'John',
  greet() {
    console.log(`Hello, I'm ${this.name}`);
  }
};

person.greet();  // "Hello, I'm John"
```

Functional Programming (FP):

- JavaScript also supports **functional programming** principles, such as treating functions as first-class citizens, meaning functions can be passed as arguments, returned from other functions, and assigned to variables.
- You can write pure functions, use higher-order functions like `map`, `filter`, and `reduce`, and embrace immutability.

```
const numbers = [1, 2, 3];
const doubled = numbers.map(n => n * 2);
console.log(doubled);  // [2, 4, 6]
```

- **Java Syntax (but not a lot like Java)**:
  - JavaScript shares some **syntax similarities** with Java, such as the use of curly braces `{}` to define blocks and `if`/`else` structures. Both languages use similar naming conventions, such as camelCase for variable names.
  - **Differences from Java**: JavaScript is a **dynamically typed** language, while Java is **statically typed**. JavaScript doesn't require you to explicitly define data types, whereas Java enforces type declarations. JavaScript also lacks many features common in Java, such as access modifiers (`public`, `private`), and is more lightweight in how it handles classes and inheritance (via prototypes instead of traditional classes).

- **Scheme and Self Semantics**:
  - **Scheme**: JavaScript's functional programming side has roots in **Scheme**, a minimalist Lisp dialect. JavaScript, like Scheme, supports first-class functions (functions as data), closures (functions retaining access to variables in their outer scope), and recursion. These features enable powerful functional programming capabilities.

```
function outer() {
  const message = "Hello from the outer function";
  return function inner() {
    console.log(message);
  };
}

const innerFunc = outer();
innerFunc();  // "Hello from the outer function"
```

Self: JavaScript also takes inspiration from **Self**, an object-oriented language based on **prototypal inheritance**. In JavaScript, objects inherit from other objects directly, rather than through traditional class-based inheritance.

```
const animal = {
  speak() {
    console.log("Animal speaking");
  }
};

const dog = Object.create(animal);
dog.speak();  // "Animal speaking"
```

Why JavaScript is Important

1. **Ubiquity**:
   - JavaScript is **everywhere**. It's the only language that runs natively in **browsers** (thanks to its integration with the Document Object Model (DOM)), making it essential for web development. Virtually every website you visit uses JavaScript.
   - With technologies like **Node.js**, JavaScript has become increasingly popular on the **server-side**, allowing developers to use one language for both frontend and backend.
2. **Browser Support**:
   - JavaScript is supported by all modern web browsers, including Chrome, Firefox, Safari, Edge, and more. This makes it the **de facto language** for client-side scripting in web development.
   - Browsers come with JavaScript engines (like V8 in Chrome) that have made JavaScript execution extremely fast.

3. **AJAX Rich Clients**:
    - JavaScript powers **AJAX (Asynchronous JavaScript and XML)**, allowing web applications to fetch data from servers in the background without reloading the entire page. This enables the creation of **rich, dynamic user experiences**.
    - Popular frameworks like **React**, **Angular**, and **Vue** rely on JavaScript to provide fast and interactive interfaces.
4. **Greatness**:
    - JavaScript is flexible, powerful, and versatile, with support for **object-oriented**, **functional**, and **event-driven** programming.
    - It has a massive ecosystem of libraries and frameworks (e.g., React, Angular, Vue) and tools (e.g., Webpack, Babel), which makes development faster and easier.
    - **NPM (Node Package Manager)**, which is the largest software registry, makes it simple to integrate third-party libraries into your projects.
5. **Increasing Usage on the Server-Side**:
    - Thanks to **Node.js**, JavaScript has become popular on the server-side, allowing developers to write full-stack applications using just one language.
    - Server-side JavaScript enables non-blocking, event-driven architectures, which are efficient for real-time applications like chat apps, games, or data streaming.
    - Many companies are adopting JavaScript for backend development due to the simplicity and speed offered by tools like **Express.js** and **NestJS**.

Here's a brief explanation of each point:

1. **JavaScript is important**:
    - JavaScript is crucial because it powers web development, running on both the client side (browser) and server side (Node.js). It's the only language native to browsers, making it vital for creating interactive web pages and applications.
2. **Significantly different from 'mainstream' languages like C# and Java**:
    - JavaScript differs from statically typed languages like C# and Java because it's dynamically typed and uses prototypal inheritance rather than class-based inheritance. It also runs in a browser environment, whereas C# and Java are typically used for backend or desktop applications.
3. **Easy to learn and extremely powerful**:
    - JavaScript is beginner-friendly due to its simple syntax and immediate feedback in the browser. Despite its simplicity, it's extremely powerful with support for advanced concepts like functional programming, asynchronous operations, and server-side development through Node.js.

Server-side JavaScript (Non-blocking Event-driven Architecture):

- **Non-blocking**: JavaScript on the server (using Node.js) can handle multiple requests without waiting for one task to finish before starting another. It doesn't "block" the

execution of other code while waiting for operations (like reading a file or fetching data) to complete.

- **Event-driven**: Node.js uses an event loop, which continuously listens for events (like incoming requests) and triggers callbacks or handlers when the event occurs, making it highly efficient for I/O operations.

Synchronous (Sync) = Blocking:

- In synchronous (sync) operations, each task **blocks** the execution of subsequent tasks until it is complete.
- This means the program must wait for a current task to finish before moving on to the next task.

Example:

```
const result = readFileSync('file.txt');  // Blocking: Waits for the file to be
console.log(result);                      // Only runs after file reading comple
```

In the above example, other tasks are **blocked** from running until the file is fully read.

Asynchronous (Async) = Non-blocking:

- Asynchronous (async) operations are **non-blocking** because they allow the program to continue executing other tasks while waiting for a particular operation (like file reading or fetching data) to complete.
- The result of the async task is handled later, usually with a **callback**, **promise**, or `async/await`.

```
readFile('file.txt', (err, result) => {   // Non-blocking: Reads file asynchrono
  console.log(result);                    // Runs when file reading is done
});
console.log('This runs immediately');     // This runs while the file is being r
```

Here, the program doesn't wait for `readFile()` to complete and instead moves on to the next task (`console.log('This runs immediately')`), making it **non-blocking**.

Why non-blocking and async matter:

Non-blocking and async make server-side JavaScript efficient, especially for handling numerous tasks like handling thousands of simultaneous web requests without performance bottlenecks.

Types of Variables in JavaScript:

var:

- Used before ES6, has **function scope** and can be **redeclared**.
- It can be hoisted but might lead to unintended behavior due to hoisting.

Example:

```
var x = 5;
```

let:

- Introduced in ES6, has **block scope** (confined to the block where it's defined).
- Cannot be redeclared within the same scope but can be reassigned.

Example:

```
let y = 10;
```

const:

- Introduced in ES6, has **block scope** like `let`, but cannot be **reassigned** once initialized.
- Primarily used for variables that are not supposed to change.

Example:

```
const z = 15;
```

Each variable type has its specific use case, with `let` and `const` being the preferred choices due to their block-scoping and better handling of variable declarations in modern JavaScript development.

Hoisting is a JavaScript behavior where **variable and function declarations** are moved (or "hoisted") to the top of their scope **before** the code execution. This means that you can use variables and functions **before they are declared** in the code.

```
console.log(x);  // Output: undefined
var x = 5;
console.log(x);  // Output: 5
```

What happens:

- The declaration `var x;` is hoisted to the top, but the initialization (`x = 5`) is **not** hoisted.
- Before the initialization, the variable `x` exists but is **undefined**.

The code above is interpreted by JavaScript as:

```
var x;

console.log(x);  // Output: undefined

x = 5;

console.log(x);  // Output: 5
```

Unlike `var`, `let` and `const` do not allow accessing variables before they are declared. This is because `let` and `const` are hoisted but are placed in a "temporal dead zone" until the code reaches the declaration.

Primitive Types:

A **primitive type** is a basic data type in JavaScript that is not an object and has no methods. JavaScript's primitive types include:

- **`null`**
- **`undefined`**
- **Boolean** (`true` or `false`)
- **Number**
- **String**
- **Symbol** (ES6)
- **BigInt** (ES2020)

`null`:

- **Primitive type** representing the intentional **absence of any object value**.
- Typically used when you explicitly want to denote an **empty or non-existent value**.

`undefined`:

- **Primitive type** that signifies a variable has been declared but **not assigned a value**.
- Automatically assigned to variables that are declared but not initialized.

Link to Booleans:

- **`null`** and **`undefined`** are both **falsy** values.
  - When converted to boolean:
    - `Boolean(null)` → `false`
    - `Boolean(undefined)` → `false`

Link to Numbers:

- When coerced to numbers:
  - **`null`** is converted to `0`.
    - `Number(null)` → `0`
  - **`undefined`** is converted to `NaN` (Not a Number).
    - `Number(undefined)` → `NaN`

Link to 0, 1, and -1:

- **Boolean to Numbers**:
  - `false` converts to `0`.
  - `true` converts to `1`.
- **Comparisons**:
  - `null` equals `0` in numeric comparisons (`null == 0` is `false`, but `Number(null) == 0` is `true`).
  - **`undefined` does not equal `0`, `1`, or `-1`**, and it returns `NaN` in most numeric contexts.

In short:

- **`null`** is intentionally empty and coerces to `0` when converted to a number.
- **`undefined`** means a value hasn't been assigned and coerces to `NaN` in numeric contexts.