Day 2 Trainee Tasks

Learnings covered:

ES6 Overview:

- **ES6**: Modernized JavaScript with new features for better readability, maintenance, and functionality.
- **Key Features**: Arrow functions, `let`, `const`, Promises, modules, and default parameters.

Variables and Parameters:

1. **`let` & `const`**: Block-scoped; `const` prevents reassignment.
2. **Destructuring**: Extract array/object values into variables.
3. **Default Params**: Set default function values.
4. **Rest Params**: Collect multiple arguments into an array.
5. **Spread Operator**: Expand arrays/objects into individual elements.
6. **Template Literals**: Use `${}` for embedded expressions and multi-line strings.

Traceur:

- **Traceur**: Transpiles ES6+ to ES5 for compatibility with older browsers.

Swapping with Destructuring:

- **Swap Variables**: `[a, b] = [b, a];`
- **Array/Object Swap**: Use destructuring to swap values.

Rest Parameters:

- **Use**: `function sum(...numbers) {}` to gather arguments.

Spread Operator:

- **Copy/Merge**: `let newArr = [...arr];`
- **Function Args**: `Math.max(...numbers)`

Template Literals:

- **String Interpolation**: `` `Hello, ${name}` ``
- **Multi-line**: Use backticks for multi-line text.

What is ES6?

ECMAScript 6 (ES6), also known as ECMAScript 2015, is the sixth edition of ECMAScript, a scripting language specification standardized by Ecma International. It is the version of JavaScript that introduced major updates and enhancements, making JavaScript more powerful and expressive. ES6 introduced many features aimed at making coding in JavaScript easier, more maintainable, and more structured.

Why Do We Use ES6?

1. **Improved Readability**: ES6 syntax is more concise and readable compared to previous versions of JavaScript (ES5). It introduced new features like arrow functions, classes, and template literals to make the code more declarative and modular.
2. **Better Maintenance**: Features like destructuring, `let` and `const`, modules, and classes make it easier to write maintainable and reusable code.
3. **New Functionalities**: ES6 brought several new functionalities like Promises, default parameters, and modules, making JavaScript suitable for complex applications.

Variables and Parameters in ES6

1. **`let` and `const`**
   - **`let`**: This keyword is used to declare block-scoped variables. Unlike `var`, which is function-scoped, `let` is constrained to the block in which it is declared (e.g., inside an `if` or `for` block).

     ```
     let name = "John";
     if (true) {
       let age = 30; // Only available inside this block
       console.log(age); // 30
     }
     console.log(age); // Error: age is not defined
     ```

   - **`const`**: This keyword is used to declare block-scoped constants, meaning the variable cannot be reassigned after its initial declaration. However, if the variable points to an object, the object's properties can still be changed.

     ```
     const PI = 3.14159;
     PI = 3; // Error: Assignment to constant variable

     const person = { name: "John" };
     person.name = "Jane"; // Works fine, only the reference is
     constant
     ```

2. **Destructuring**

Destructuring is a convenient way of extracting data from arrays or objects and assigning them to variables.

- **Array Destructuring**:

  ```
  const numbers = [1, 2, 3];
  const [one, two, three] = numbers;
  console.log(one); // 1
  ```

- **Object Destructuring**:

  ```
  const person = { name: "John", age: 30 };
  const { name, age } = person;
  console.log(name); // John
  ```

Destructuring makes it easier to work with objects and arrays, especially when dealing with API data or deeply nested structures.

3. **Default Parameters**

Default parameters allow function parameters to have a default value if no value is passed.

```
function greet(name = "Guest") {
  return `Hello, ${name}!`;
}

console.log(greet()); // Hello, Guest!
console.log(greet("John")); // Hello, John!
```

This feature reduces the need for checking if a parameter is provided and gives the code better readability.

4. **Rest Parameters**

Rest parameters allow you to represent an indefinite number of arguments as an array. This is useful when you want to pass multiple arguments to a function but don't know how many in advance.

```
function sum(...numbers) {
  return numbers.reduce((acc, curr) => acc + curr, 0);
}

console.log(sum(1, 2, 3, 4)); // 10
```

In this example, `...numbers` collects all arguments into an array, allowing us to handle them efficiently.

5. **Spread Operator**

The spread operator (`...`) allows you to spread elements from an array or object into individual elements or properties. It's often used to make copies or combine arrays/objects.

- **Array Spread**:

  ```
  const arr1 = [1, 2, 3];
  const arr2 = [4, 5, 6];
  const combined = [...arr1, ...arr2]; // [1, 2, 3, 4, 5, 6]
  ```

- **Object Spread**:

  ```
  const obj1 = { a: 1, b: 2 };
  const obj2 = { c: 3, d: 4 };
  const combined = { ...obj1, ...obj2 }; // { a: 1, b: 2, c: 3, d:
  4 }
  ```

Spread syntax is especially useful when merging arrays, copying objects, or passing arrays as arguments.

6. **Template Literals**

Template literals (also called template strings) provide a new way of working with strings. You can embed variables and expressions inside strings using `${}`.

```
const name = "John";
const greeting = `Hello, ${name}!`;
console.log(greeting); // Hello, John!
```

Template literals make string interpolation much easier compared to the concatenation methods used in ES5.

What is Traceur?

Traceur is a JavaScript transpiler created by Google that allows you to write ES6 (and beyond) code, which is then compiled down to ES5, making it compatible with older browsers that do not support ES6 features. The main purpose of Traceur is to allow developers to use the latest JavaScript features while maintaining browser compatibility.

Key Concepts of Traceur:

- **Transpiling**: This is the process of converting code from one version or dialect of a language to another. Traceur transpiles ES6+ code into ES5, making it executable in environments that only support older JavaScript versions.

- **Polyfilling**: In addition to transpiling, Traceur can add polyfills, which are functions that mimic the behavior of ES6+ features in older environments.

Traceur was one of the first widely-used transpilers, though tools like Babel have since become more popular.

Why Use Traceur or Babel?

1. **Backward Compatibility**: Transpilers like Traceur or Babel allow developers to write modern JavaScript (ES6+), which is transpiled into ES5 for better browser compatibility.
2. **Use of Modern Features**: Modern features such as `let`, `const`, classes, arrow functions, modules, and async/await are not supported by all browsers natively. By using a transpiler, you can write future-proof code.
3. **Development Flexibility**: It allows you to use new language features without waiting for all browsers to implement them.

1. Swapping with Destructuring (Different Scenarios)

Destructuring assignment allows you to unpack values from arrays or properties from objects into distinct variables. It also enables easy swapping of variables and values.

Basic Variable Swapping

You can easily swap the values of two variables using array destructuring:

```
let a = 1;
let b = 2;

[a, b] = [b, a];

console.log(a); // 2
console.log(b); // 1
```

Swapping Array Elements

You can swap elements within an array using destructuring as well:

```
let arr = [1, 2, 3, 4];

// Swap the first and last elements
[arr[0], arr[arr.length - 1]] = [arr[arr.length - 1], arr[0]];

console.log(arr); // [4, 2, 3, 1]
```

Swapping Values in Objects

While swapping values inside objects directly doesn't work as simply as arrays, you can still destructure and swap values between different object properties:

```
let obj = { x: 10, y: 20 };

[obj.x, obj.y] = [obj.y, obj.x];

console.log(obj); // { x: 20, y: 10 }
```

Swapping Values in Function Arguments

You can swap values directly inside a function that accepts parameters:

```
function swap([a, b]) {
  return [b, a];
}

let result = swap([10, 20]);
console.log(result); // [20, 10]
```

Swapping Nested Object Properties

You can swap nested properties inside complex objects:

```
let obj = {
  inner: {
    a: 5,
    b: 10
  }
};

[obj.inner.a, obj.inner.b] = [obj.inner.b, obj.inner.a];

console.log(obj); // { inner: { a: 10, b: 5 } }
```

---

2. Rest Parameters (Different Use Cases)

Rest parameters allow functions to accept an indefinite number of arguments as an array. It replaces the older `arguments`object and offers more flexibility.

Basic Example of Rest Parameters

Rest parameters gather the remaining arguments into an array. For example:

```
function sum(...numbers) {
  return numbers.reduce((acc, curr) => acc + curr, 0);
}

console.log(sum(1, 2, 3)); // 6
console.log(sum(1, 2, 3, 4, 5)); // 15
```

Using `length` with Rest Parameters

With `rest`, the array length can be used, unlike the `arguments` object:

```
function countArgs(...args) {
  console.log(args.length);
}

countArgs(1, 2, 3); // 3
countArgs(1, 2);    // 2
```

Rest Parameters with Named Arguments

You can have named parameters along with the rest parameter:

```
function logFirstAndRest(first, ...rest) {
  console.log('First:', first);
  console.log('Rest:', rest);
}

logFirstAndRest(1, 2, 3, 4);
// First: 1
// Rest: [2, 3, 4]
```

Combining Rest Parameters and Destructuring

You can combine rest parameters with destructuring to capture part of an array:

```
const team = ["Alice", "Bob", "Charlie", "Dave"];

const [leader, ...members] = team;
console.log(leader);  // Alice
console.log(members); // [Bob, Charlie, Dave]
```

Iterating with Rest Parameters

Iterate through a rest parameter to manipulate each value:

```
function multiply(multiplier, ...numbers) {
  return numbers.map(num => num * multiplier);
}

console.log(multiply(2, 1, 2, 3)); // [2, 4, 6]
```

---

3. Spread Operator (Different Use Cases)

The **spread operator (`...`)** expands an iterable (like an array or object) into its individual elements or properties. It’s useful for copying, merging, and more.

Copying Arrays

Spread is often used to create shallow copies of arrays:

```
let arr = [1, 2, 3];
let copy = [...arr];
console.log(copy); // [1, 2, 3]
```

Merging Arrays

It can also merge multiple arrays into one:

```
let arr1 = [1, 2];
let arr2 = [3, 4];
let merged = [...arr1, ...arr2];
console.log(merged); // [1, 2, 3, 4]
```

Using Spread in Function Calls

The spread operator can be used to pass arrays as arguments to functions:

```
let numbers = [1, 2, 3];
function sum(a, b, c) {
  return a + b + c;
}

console.log(sum(...numbers)); // 6
```

Spreading in Objects

In ES6, the spread operator works with objects to copy or merge them.

- **Copying Objects**:

  ```
  const person = { name: "Alice", age: 25 };
  const clone = { ...person };
  console.log(clone); // { name: 'Alice', age: 25 }
  ```

- **Merging Objects**:

  ```
  const user = { name: "John", age: 30 };
  const details = { job: "Developer", city: "New York" };

  const userDetails = { ...user, ...details };
  console.log(userDetails); // { name: "John", age: 30, job: "Developer",
  city: "New York" }
  ```

Spread Operator in Function Arguments

Passing an array as separate arguments using spread:

```
const numbers = [5, 6, 7];
const maxNumber = Math.max(...numbers);
console.log(maxNumber); // 7
```

Combining Spread and Rest

You can use spread and rest together in functions:

```
function sortAndLog(...nums) {
  return [...nums].sort((a, b) => a - b);
}

console.log(sortAndLog(3, 1, 4, 1, 5, 9)); // [1, 1, 3, 4, 5, 9]
```

---

4. Template Literals

Template literals (also called template strings) are enclosed by backticks (`` ` ``) instead of single or double quotes. They allow embedding variables and expressions inside strings using the `${}` syntax, providing a more readable and flexible way to work with strings.

Basic Example of Template Literals

Instead of using string concatenation:

```
let name = "John";
let age = 30;

let greeting = `Hello, my name is ${name} and I am ${age} years old.`;
console.log(greeting);
// Hello, my name is John and I am 30 years old.
```

Multi-line Strings

Template literals allow multi-line strings without needing to use `\n` or string concatenation:

```
let multiLine = `This is a string
that spans multiple
lines.`;

console.log(multiLine);
// This is a string
// that spans multiple
// lines.
```

Embedding Expressions

You can embed any valid JavaScript expression inside `${}`:

```
let a = 10;
let b = 20;

let result = `The sum of a and b is ${a + b}.`;
console.log(result); // The sum of a and b is 30.
```

Tagged Template Literals

A more advanced feature is **tagged template literals**, where a function can process a template string. This can be useful for string sanitization or localization.

```
function tag(strings, ...values) {
  return strings[0] + values[0].toUpperCase() + strings[1];
}

let name = "john";
let taggedResult = tag`Hello ${name}, how are you?`;
console.log(taggedResult); // Hello JOHN, how are you?
```

Nested Template Literals

You can nest template literals within other template literals, combining multiple expressions:

```
const name = "Alice";
const age = 28;
const message = `My name is ${name}, and ${age > 18 ? `I am an adult` : `I am
not an adult`}.`;
console.log(message);
// My name is Alice, and I am an adult.
```