String Manipulation (15 minutes)

- **String Properties and Methods**: `length`, `toUpperCase()`, `toLowerCase()`, `indexOf()`, `slice()`, `split()`, `replace()`.
- **Template Literals**: Introduction to template literals using backticks (`) and expressions` ${}`.
- **Examples**: Create examples to demonstrate string manipulation and template literals.

1. String Properties and Methods

JavaScript provides various properties and methods to work with strings, allowing you to manipulate text effectively.

Example 1: String Properties

```
// String length property
const text = 'Hello, world!';
console.log('Length of the text:', text.length); // Output: Length of the text: 13
```

Example 2: `toUpperCase()` and `toLowerCase()`

```
// Converting to uppercase
const upperCaseText = text.toUpperCase();
console.log('Uppercase:', upperCaseText); // Output: Uppercase: HELLO, WORLD!

// Converting to lowercase
const lowerCaseText = text.toLowerCase();
console.log('Lowercase:', lowerCaseText); // Output: Lowercase: hello, world!
```

Example 3: `indexOf()`

```
// Finding the index of a substring
const index = text.indexOf('world');
console.log('Index of "world":', index); // Output: Index of "world": 7

// If substring is not found, indexOf returns -1
const notFoundIndex = text.indexOf('JavaScript');
```

```javascript
console.log('Index of "JavaScript":', notFoundIndex); // Output: Index of
"JavaScript": -1
```

Example 4: `slice()`

```javascript
// Extracting a part of a string
const slicedText = text.slice(7, 12);
console.log('Sliced text:', slicedText); // Output: Sliced text: world
```

Example 5: `split()`

```javascript
// Splitting a string into an array
const wordsArray = text.split(' ');
console.log('Words Array:', wordsArray); // Output: Words Array:
['Hello,', 'world!']
```

Example 6: `replace()`

```javascript
// Replacing a part of a string
const replacedText = text.replace('world', 'JavaScript');
console.log('Replaced text:', replacedText); // Output: Replaced text:
Hello, JavaScript!
```

2. Template Literals

Template literals are a new way to work with strings in JavaScript. They allow for easier string interpolation and multi-line strings.

Example 7: Using Template Literals

```javascript
// Using template literals for string interpolation
const name = 'Alice';
const greeting = `Hello, ${name}! Welcome to JavaScript.`;
console.log(greeting); // Output: Hello, Alice! Welcome to JavaScript.

// Multi-line strings using template literals
const multiLineText = `This is a multi-line string.
You can write text across multiple lines
without using escape characters.`;
console.log(multiLineText);
// Output:
```

```
// This is a multi-line string.
// You can write text across multiple lines
// without using escape characters.
```

Explanation:

1. **String Properties and Methods**:
   - `length`: A property that returns the length of the string.
   - **`toUpperCase()` and `toLowerCase()`**: Methods that convert the string to uppercase or lowercase, respectively.
   - `indexOf()`: Returns the index of the first occurrence of a specified substring. Returns `-1` if the substring is not found.
   - `slice()`: Extracts a part of a string and returns it as a new string. Takes two parameters: the starting index and the ending index (not included).
   - `split()`: Splits a string into an array of substrings based on a specified separator.
   - `replace()`: Replaces a specified substring with another substring in a string. Only the first occurrence is replaced unless a regular expression with the global flag (`/g`) is used.
2. **Template Literals**:
   - **Template literals** are enclosed by backticks (`) instead of single or double quotes.
   - They allow for **string interpolation**, which lets you embed expressions inside a string using `${expression}`.
   - They also support **multi-line strings** without the need for escape characters.

JavaScript Password Generator

```javascript
// Function to generate a random password
function generatePassword(length = 12) {
    // Define character sets for password
    const upperCase = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ';
    const lowerCase = 'abcdefghijklmnopqrstuvwxyz';
    const numbers = '0123456789';
    const specialChars = '!@#$%^&*()_+[]{}|;:,.<>?';

    // Combine all character sets into one
    const allChars = upperCase + lowerCase + numbers + specialChars;

    // Initialize password array
    let password = [];

    // Ensure the password includes at least one character from each set

    // Select a random uppercase letter
    // Math.random() generates a random decimal between 0 (inclusive) and
1 (exclusive).
    // Multiplying it by upperCase.length (26) gives a number between 0
and 25.999...
    // Math.floor() then rounds down to the nearest whole number, ensuring
an index between 0 and 25.
    // This is used to select a random character from the upperCase
string.
    password.push(upperCase[Math.floor(Math.random() *
upperCase.length)]);

    // Select a random lowercase letter
    // The same logic applies here: generate a random index to pick a
character from the lowerCase string.
    password.push(lowerCase[Math.floor(Math.random() *
lowerCase.length)]);

    // Select a random number
    // Math.random() is used again to pick a character from the numbers
string.
    password.push(numbers[Math.floor(Math.random() * numbers.length)]);
    // Select a random special character
    // Math.random() is used again to pick a character from the
specialChars string.
    password.push(specialChars[Math.floor(Math.random() *
specialChars.length)]);

    // Fill the remaining password length with random characters from all
sets
    for (let i = password.length; i < length; i++) {
        // Generate a random index for allChars and pick a character from
it
        // Math.floor(Math.random() * allChars.length) calculates a random
index across all combined character sets.
        password.push(allChars[Math.floor(Math.random() *
allChars.length)]);
    }

    // Shuffle the password array to randomize character order
    // The .sort() function with a random comparison function
(Math.random() - 0.5) shuffles the array.
    // Math.random() generates a number between 0 and 1. Subtracting 0.5
makes the range -0.5 to 0.5.
    // This causes the sort function to randomly decide the order of
elements, effectively shuffling them.
    password = password.sort(() => Math.random() - 0.5);

    // Join the password array into a string and return it
    return password.join('');
}

// Example usage
const newPassword = generatePassword();
console.log('Generated Password:', newPassword);
```

Introduction to Object-Oriented Programming (OOP) in JavaScript

Object-Oriented Programming (OOP) is a programming paradigm that uses objects to represent real-world entities and concepts. OOP is centered around the idea of creating "objects," which are instances of "classes." These objects can hold data (known as properties or attributes) and methods (functions that operate on the data).

What is OOP?

OOP is a way to model and structure code that makes it more intuitive to work with, especially when dealing with complex systems. In OOP, the primary focus is on objects that are created from classes. A **class** is like a blueprint that defines the properties and behaviors that the objects created from it will have. An **object** is an instance of a class; it's a specific implementation of the class with actual values and data.

Here are some key concepts of OOP:

1. **Classes and Objects:** A class is a template or blueprint for creating objects. For example, you might have a `Book` class that defines what properties (like `title`, `author`) and methods (like `describe()`) a book should have. An object is an instance of a class—imagine creating a specific book using the `Book` class blueprint.

```
// Define the Book class
class Book {
// Constructor method to initialize the properties of a new Book
object
    constructor(title, author) {
        this.title = title;  // Property: the title of the book
        this.author = author; // Property: the author of the book
    }
    // Method: a function that describes an action related to the book
    describe() {
        console.log(`"${this.title}" is written by ${this.author}.`);
    }
}
// Create an object (instance) of the Book class
const myBook = new Book('The Great Adventure', 'Alex Smith');
// Access the properties of the object
console.log(myBook.title);  // Output: The Great Adventure
console.log(myBook.author); // Output: Alex Smith
// Call the method of the object
myBook.describe(); // Output: "The Great Adventure" is written by Alex
Smith.
```

2. **Encapsulation:** Encapsulation is the bundling of data (properties) and methods (functions) that operate on the data into a single unit, or class. It restricts direct access to some of an object's components, which means that you can control and protect the internal state of the object and expose only what is necessary. This makes code easier to maintain and prevents unintended interference.

```
// Define the Book class
class Book {
    // Constructor method to initialize the properties of a new Book
object
    constructor(title, author) {
        this._title = title;  // Private-like property: the title of the
book
        this._author = author; // Private-like property: the author of the
book
    }

    // Getter method for the title
    getTitle() {
        return this._title;
    }

    // Setter method for the title
    setTitle(newTitle) {
        this._title = newTitle;
    }

    // Getter method for the author
    getAuthor() {
        return this._author;
    }

    // Setter method for the author
    setAuthor(newAuthor) {
        this._author = newAuthor;
    }

    // Method: a function that describes an action related to the book
    describe() {
        console.log(`"${this._title}" is written by ${this._author}.`);
    }
}

// Create an object (instance) of the Book class
const myBook = new Book('The Great Adventure', 'Alex Smith');

// Access the properties using getter methods
console.log(myBook.getTitle());  // Output: The Great Adventure
console.log(myBook.getAuthor()); // Output: Alex Smith

// Call the method of the object
myBook.describe(); // Output: "The Great Adventure" is written by Alex
Smith.

// Modify the properties using setter methods
myBook.setTitle('The Great Mystery');
myBook.setAuthor('Jordan Lee');

// Access the updated properties using getter methods
console.log(myBook.getTitle());  // Output: The Great Mystery
console.log(myBook.getAuthor()); // Output: Jordan Lee

// Call the method of the object with updated properties
myBook.describe(); // Output: "The Great Mystery" is written by Jordan
Lee.
```

3. **Inheritance:** Inheritance allows a class to inherit properties and methods from another class. This helps to create a new class based on an existing class, allowing for code reuse and a more hierarchical class structure. For example, you could have a `Vehicle` class and create a `Car` class that inherits from `Vehicle`. The `Car` class will have all the properties and methods of `Vehicle`, and you can also add specific ones to `Car`.

Inheritance allows a class to inherit properties and methods from another class. This helps in creating a new class based on an existing class, enabling code reuse and the creation of a hierarchical class structure.

Example: Inheritance

Let's create a base class called `Animal` and a derived class called `Dog` that inherits from `Animal`.

JavaScript Code Example

```
// Define the Animal class (base class)
class Animal {
    // Constructor method to initialize the properties of a new Animal
object
    constructor(name) {
        this.name = name; // Property: the name of the animal
    }

    // Method: a function that describes an action related to the animal
    speak() {
        console.log(`${this.name} makes a sound.`);
    }
}

// Define the Dog class (derived class) that inherits from Animal
class Dog extends Animal {
    // Constructor method to initialize the properties of a new Dog object
    constructor(name, breed) {
        // Call the parent class's constructor using 'super'
        super(name);
        this.breed = breed; // Additional property specific to Dog
    }

    // Method: overrides the speak method from the Animal class
    speak() {
        console.log(`${this.name} barks.`);
    }
}

// Create an object (instance) of the Dog class
const myDog = new Dog('Buddy', 'Golden Retriever');
```

```
// Access the properties of the object
console.log(myDog.name);  // Output: Buddy
console.log(myDog.breed); // Output: Golden Retriever

// Call the method of the object
myDog.speak(); // Output: Buddy barks.
```

4. **Polymorphism:** Polymorphism means "many shapes" and allows objects of different classes to be treated as objects of a common superclass. It allows a single interface to represent different underlying data types. For example, a `Vehicle` class might have a method `move()`, and the `Car` and `Bike` classes, which inherit from `Vehicle`, might implement `move()` differently. However, you can still call `move()` on any `Vehicle` object, whether it's a car or a bike.

5. **Abstraction:** Abstraction means hiding the complex reality while exposing only the essential parts. It reduces programming complexity and effort by hiding the unnecessary details from the user. For example, when you use a smartphone, you interact with its interface (the screen), without needing to understand the complex hardware and software working inside the device.

Why Do We Use OOP?

1. **Improved Code Organization:** OOP helps in organizing the code better by grouping related properties and methods into objects. This makes the code easier to understand, manage, and debug.

2. **Reusability:** Through inheritance, you can create new classes that reuse, extend, or modify the behavior defined in other classes. This avoids code duplication and enhances maintainability.

3. **Modularity:** By encapsulating the related data and behavior into objects, you make the code modular. You can independently develop, test, and maintain different parts of an application.

4. **Scalability:** OOP makes it easier to scale applications. As applications grow, the modular structure of OOP helps in managing the increasing complexity. You can add new functionality with minimal changes to existing code.

5. **Flexibility Through Polymorphism:** With polymorphism, you can write code that works on objects of different classes but shares a common superclass. This flexibility makes it easier to introduce new classes without modifying existing code.

6. **Abstraction for Simplicity:** By using abstraction, OOP allows programmers to focus on what the object does instead of how it does it. This simplifies programming and helps in managing complexity.

Conclusion

Object-Oriented Programming is a powerful paradigm that can help you write more structured, reusable, and maintainable code. By thinking in terms of objects and their interactions, you can model real-world systems more naturally and handle the complexity of large applications more effectively. As we dive deeper into OOP in JavaScript, you'll see how these concepts come to life and how they can be applied to solve real-world problems in a more elegant way.

Let's get started with exploring how JavaScript implements these OOP concepts!