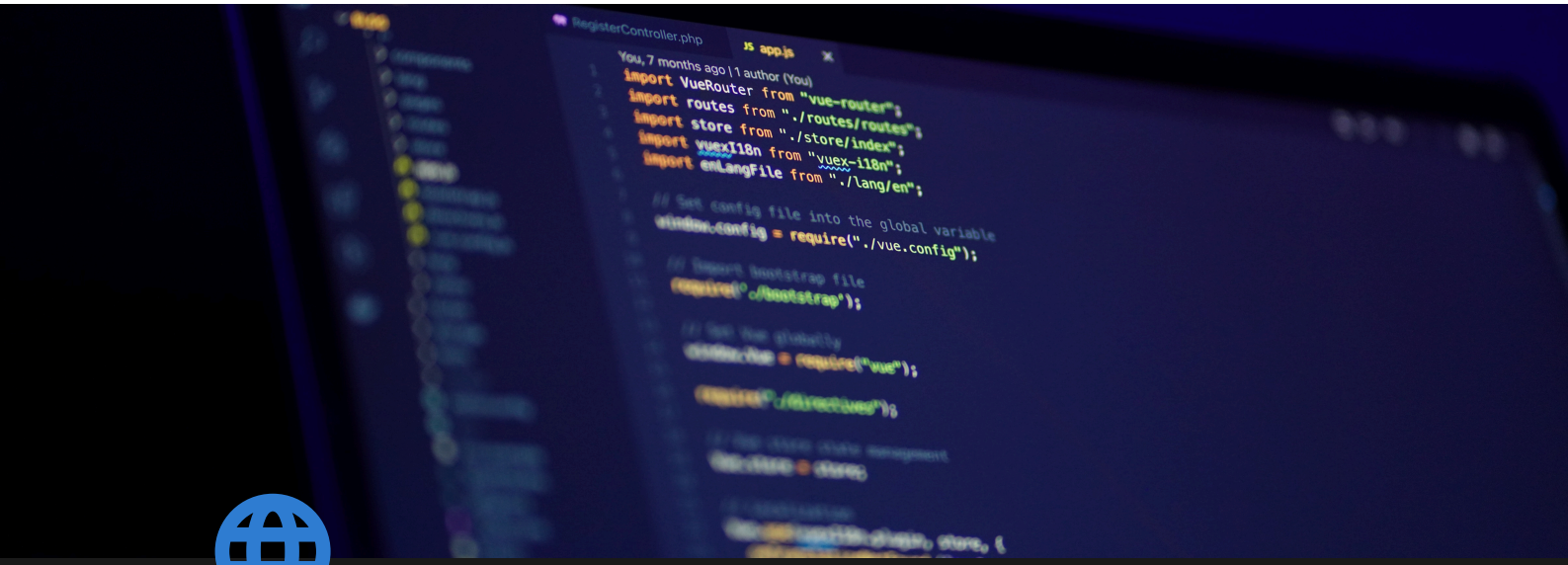

JS | Switch Case, Arrays, Objects, & Higher-Order Functions

Intended for Beginners

Topics

1. Introduction to JavaScript Switch Case (20 minutes)
2. Commonly Used Methods on Arrays (30 minutes)
3. In-Depth Look at Array Methods (50 minutes)
4. Array De-structuring and Spread Operator (30 minutes)
5. Loops and Iterators in Arrays (20 minutes)
6. Higher-Order Functions: Map, Filter, and Reduce (40 minutes)
7. Object Declaration and Accessing Properties (30 minutes)
8. Object Methods and Their Use Cases (20 minutes)

1. Introduction to JavaScript Switch Case (20 minutes)

Overview:

- The switch statement is used to perform different actions based on different conditions.

Links:

- MDN Documentation on switch

Example:

```
let fruit = "apple"; switch (fruit) { case "banana": console.log("Bananas are yellow."); break; case "apple": console.log("Apples are red."); break; case "grape": console.log("Grapes are purple."); break; default: console.log("Unknown fruit."); }
```

Exercise:

- Write a switch case to check the day of the week and print if it's a weekday or weekend.

  Hint1 : There are 7 days in a week. 2 are Weekends & rest are Weekdays
  Hint2 : Think about utilising unique property break in Switch-Case

```
let day="tuesday"; // Condition Var. switch(day){ case "monday": case "tuesday": case "wednesday": case "thursday": case "friday": console.log("weekdays") break; case "saturday": case "sunday": console.log("weekends") break; default: console.log("Invalid Day Name") }
```

2. Commonly Used Methods on Arrays (30 minutes)

Overview:

- Arrays are a fundamental part of JavaScript. Learning to manipulate them is crucial.
- Arrays are collections of data; mostly of the same type though JS supports mixed arrays.
- Arrays Start with the Index as 0. So, for an array named `arr` ; the 1st element is accessed by using `arr[0]`

Links:

- MDN Array Documentation

Example:

```
let fruits = ["apple", "banana", "grape"]; console.log(fruits.join(",
")); // apple, banana, grape
```

Exercise:

- Create an array of your favorite movies and perform various operations like adding, removing, and joining elements.

  Here's how you can create an array of your favorite movies in JavaScript and perform some common operations like adding, removing, and joining elements.

1. Creating an Array

In JavaScript, you can create an array using square brackets `[]`. Here's an example:

```
let favoriteMovies = ["Inception", "The Matrix", "Interstellar", "The Dark Knight"];
```

2. Adding Elements to the Array

You can add elements to the end of the array using the `push()` method:

```
favoriteMovies.push("Pulp Fiction"); // Adds "Pulp Fiction" to the end of the array
```

To add an element at the beginning of the array, use the `unshift()` method:

```
favoriteMovies.unshift("Fight Club"); // Adds "Fight Club" to the beginning of the array
```

3. Removing Elements from the Array

To remove the last element from the array, use the `pop()` method:

```
favoriteMovies.pop(); // Removes the last movie ("Pulp Fiction") from the array
```

To remove the first element from the array, use the `shift()` method:

```
favoriteMovies.shift(); // Removes the first movie ("Fight Club") from the array
```

4. Joining Array Elements

If you want to join all the elements of the array into a single string, you can use the `join()` method:

```
let moviesString = favoriteMovies.join(", "); // Joins all movie titles into a single string separated by commas
```

Full Example

Here's the full code with all operations:

```
let favoriteMovies = ["Inception", "The Matrix", "Interstellar", "The Dark Knight"]; favoriteMovies.pop(); // Remove the last element favoriteMovies.shift(); // Remove the first element favoriteMovies.push("Pulp Fiction"); // Add to the end favoriteMovies.unshift("Fight Club"); // Add to the beginning let moviesString = favoriteMovies.join(", "); // Join the elements into a single string console.log(moviesString); // Output: Fight Club, The Matrix, Interstellar, Pulp Fiction
```

Summary of Operations:

- Create an array: `let favoriteMovies = ["Inception", "The Matrix", "Interstellar", "The Dark Knight"];`
- Add elements: `push()`, `unshift()`
- Remove elements: `pop()`, `shift()`
- Join elements: `join()`

3. In-Depth Look at Array Methods (50 minutes)

Overview:

- Focus on essential array methods: `pop`, `shift`, `push`, `unshift`, `concat`, `splice`, `slice`, `sort`, and `length`.

Links:

- MDN Array Methods

Examples and Exercises:

1. Pop and Shift:

```
let fruits = ["apple", "banana", "grape"]; fruits.pop(); // removes "grape" fruits.shift(); // removes "apple" console.log(fruits); // ["banana"]
```

   Exercise:

   - Perform `pop` and `shift` on an array of numbers.

2. Push and Unshift:

```
let fruits = ["banana"]; fruits.push("grape"); // adds "grape" fruits.unshift("apple"); // adds "apple" at the start console.log(fruits); // ["apple", "banana", "grape"]
```

   Exercise:

   - Perform `push` and `unshift` on an array of colors.

3. Concat:

```
let arr1 = [1, 2, 3]; let arr2 = [4, 5, 6]; let result = arr1.concat(arr2); console.log(result); // [1, 2, 3, 4, 5, 6]
```

   Exercise:

   - Merge two arrays of day names containing weekends and weekdays of the week.

4. Splice:

   Syntax : `splice(<Index of Add/Remove>, <No.Of Items to remove>, <The Item(s) to Insert>)`

```
let fruits = ["apple", "banana", "grape"]; fruits.splice(1, 0, "orang
e"); // adds "orange" at index 1 console.log(fruits); // ["apple", "or
ange", "banana", "grape"]
```

   Exercise:

- Add a roll number at position 4 and remove roll number at position 3 from an array using `splice`.

  Create an Array of Roll Numbers

```
let rollNumbers = [101, 102, 103, 104, 105];
```

To add an element at a specific position, you can use `splice(start, deleteCount, item1, item2, ...)`. Here, `start` is the index at which to start changing the array, `deleteCount` is the number of elements to remove, and `item1, item2, ...` are the elements to add.

```
rollNumbers.splice(3, 0, 106); // Adds 106 at position 4 (index 3)
```

Heading 3

To remove an element at a specific position, use `splice()` with the `deleteCount` set to 1.

```
rollNumbers.splice(2, 1); // Removes the roll number at position 3 (index 2)
```

Complete Code

```
let rollNumbers = [101, 102, 103, 104, 105]; // Add 106 at position 4 rollNumbers.splice(3, 0, 106); // Remove the roll number at position 3 rollNumbers.splice(2, 1); console.log(rollNumbers); // Output: [101, 102, 104, 106, 105]
```

Summary:

- `splice(3, 0, 106)` adds `106` at position 4 (index 3).
- `splice(2, 1)` removes the element at position 3 (index 2).

The final array after these operations would be `[101, 102, 104, 106, 105]`.

5. **Slice:**

   Extracts and returns sub-array from an existing array

   Syntax: slice(<START INDEX - Inclusive of Element>,<END INDEX - Exclusive of element>)

```
let fruits = ["apple", "banana", "grape", "orange"]; let sliced = frui
ts.slice(1, 3); // slices out "banana" and "grape" console.log(slice
d); // ["banana", "grape"]
```

   Exercise:

   - Extract parts of an array using `slice`.

   Difference between Slice and Splice:

   - `slice`
     - does not modify the original array, it returns a new array.
     - EXTRACTS a sub-array
   - `splice`
     - modifies the original array by adding/removing elements.
     - INSERTS At An Index || DELETES From An Index

6. **Sort:**

   Arranging of Data i.e. members of the array, in an increasing/ASCENDING or decreasing/DESCENDING order

   Below is a basic example on sorting Strings in Ascending and Descending orders.

```
let fruits = ["banana", "apple", "grape"]; fruits.sort(); // sorts alp
habetically console.log(fruits); // ["apple", "banana", "grape"] - Asc
ending by Default fruits.reverse(); // reverses the ascending sorted a
rray i.e. Descending orders console.log(fruits); // ['grape', 'banan
a', 'apple']
```

In JavaScript, you can sort arrays of numbers or strings in both ascending and descending order using the `sort()` method. The `sort()` method can be used directly for strings and with a comparison function for numbers.

Sorting Numbers

1. Ascending Order

For numbers, it is **necessary to provide** a comparison function(aka comparator/compare function) to `sort()` to ensure correct numerical sorting:

```
let numbers = [40, 100, 1, 5, 25, 10]; numbers.sort(function(a, b) { return a - b; }); // Using Arrow Notation, same can also be written as :: numbers.sort((a,b) => a-b) console.log(numbers); // Output: [1, 5, 10, 25, 40, 100]
```

2. Descending Order

To sort in descending order, reverse the comparison:

```
let numbers = [40, 100, 1, 5, 25, 10]; numbers.sort(function(a, b) { return b - a; }); // Using Arrow Notation, same can also be written as :: numbers.sort((a,b) => b-a) console.log(numbers); // Output: [100, 40, 25, 10, 5, 1]
```

Sorting Strings

1. Ascending Order

For strings, the `sort()` method works directly since it sorts based on Unicode values:

```
let fruits = ["Banana", "Orange", "Apple", "Mango"]; fruits.sort(); console.log(fruits); // Output: ["Apple", "Banana", "Mango", "Orange"]
```

2. Descending Order

To sort strings in descending order, you can reverse the order by providing a comparison function:

```
let fruits = ["Banana", "Orange", "Apple", "Mango"]; fruits.sort(function(a, b) { return b.localeCompare(a); }); // Using Arrow Notation, same can also be written as :: numbers.sort((a,b) => b.localeCompare(a)) console.log(fruits); // Output: ["Orange", "Mango", "Banana", "Apple"]
```

Summary

- Numbers:
  - Ascending: `numbers.sort((a, b) => a - b);`
  - Descending: `numbers.sort((a, b) => b - a);`
- Strings:
  - Ascending: `strings.sort();`
  - Descending: `strings.sort((a, b) => b.localeCompare(a));`

Exercise:

- Sort an array of random numbers and strings in ascending and descending order using arrow and regular compare method syntaxes. [Optional : Use reverse()]

7. **Length:**

   Not a Function; but a property of array objects.

   Last Index Value : Length-1

```
let fruits = ["apple", "banana", "grape"]; console.log(fruits.length);
// 3
```

Exercise:

- Find the length of an array of season names.

4. Array Destructuring and Spread Operator (30 minutes)

Overview:

- **Array Destructuring** allows you to unpack values from arrays into distinct variables.
- **Spread Operator** (`...`) allows you to expand an array into its individual elements.

Links:

- MDN Array Destructuring
- MDN Spread Operator

Example:

```
// Array Destructuring let fruits = ["apple", "banana", "grape"]; let [fi
rst, second, third] = fruits; console.log(first); // apple console.log(se
cond); // banana // Spread Operator let moreFruits = ["orange", ...fruit
s]; console.log(moreFruits); // ["orange", "apple", "banana", "grape"]
```

Exercise:

- De-structure an array of animals into separate pet-name variables.

```
const animals = ["Boa", "Python", "Crocodile", "Dog"]; let [Ajay,
Bijay, Srinu, Hilda ] = animals; console.log(Ajay) console.log(Bijay)
console.log(Srinu) console.log(Hilda)
```

- Use the spread operator to combine two arrays of Male and Female Actors into one.

5. Loops & Iterators on Arrays (30 minutes)

Overview:

Loops are a fundamental concept in programming and are especially useful when working with arrays. In JavaScript, there are several types of loops you can use to iterate through arrays.

Iterators allow you to loop through array elements.

Links:

- MDN Iterators

Example:

```
let fruits = ["apple", "banana", "grape"]; for (let fruit of fruits) { co
nsole.log(fruit); } // apple // banana // grape
```

Exercise:

- Use a `for...of` loop to iterate over an array of colors and log each one.

1. `for` Loop

The `for` loop is the most basic and versatile loop for iterating over arrays.

Example:

```
let fruits = ["Apple", "Banana", "Mango", "Orange"]; for (let i = 0; i < fruits.length; i++) { console.log(fruits[i]); }
```

- **Output:**

```
Apple Banana Mango Orange
```

2. `for...of` or Iterator based `for` Loop

The `for...of` loop is a simpler syntax for iterating over arrays. It directly gives you the value of each element.

Example:

```
let fruits = ["Apple", "Banana", "Mango", "Orange"]; for (let fruit of fruits) { console.log(fruit); }
```

- **Output:**

```
Apple Banana Mango Orange
```

3. `forEach` Method

The `forEach()` method is an array method that executes a provided function once for each array element.

Example:

```
let fruits = ["Apple", "Banana", "Mango", "Orange"]; fruits.forEach(fu
nction(fruit, index) { console.log(index + ": " + fruit); });
```

- Output:

```
0: Apple 1: Banana 2: Mango 3: Orange
```

4. `map` Method

The `map()` method creates a new array by applying a function to each element of the original array.

It's not technically a loop, and is called an iterative method. It's useful when you want to transform each element, and return a new array; without modifying the existing array.

Example:

```
let numbers = [1, 2, 3, 4]; let squaredNumbers = numbers.map(function
(num) { return num * num; }); console.log(squaredNumbers);
```

- Output:

```
[1, 4, 9, 16]
```

5. `while` Loop

The `while` loop continues to execute as long as a specified condition is true. It's less common for array iteration but still useful in certain cases.

Example:

```
let fruits = ["Apple", "Banana", "Mango", "Orange"]; let i = 0; while
(i < fruits.length) { console.log(fruits[i]); i++; }
```

- **Output:**

```
Apple Banana Mango Orange
```

6. `do...while` Loop

The `do...while` loop is similar to the `while` loop, but it guarantees that the loop body will be executed at least once.

Example:

```
let fruits = ["Apple", "Banana", "Mango", "Orange"]; let i = 0; do { c
onsole.log(fruits[i]); i++; } while (i < fruits.length);
```

- **Output:**

```
Apple Banana Mango Orange
```

Summary

- `for` **loop**: Most versatile, great for when you need an index.
- `for...of` **loop**: Simplified syntax when you just need the element value.
- `forEach()` : Array method for easy iteration with optional index.
- `map()` : Creates a new array with the results of applying a function to every element.
- `while` **and** `do...while` **loops**: Useful when the number of iterations isn't determined by the array length but by a condition.
- Condition Placement :
  `while` -> At the beginning of the loop. If condition is false, the loop won't even execute once
  `do...while` -> At the end of the loop. Even if condition is false, the loop will execute at-least once

6. Higher-Order Functions: Map, Filter, and Reduce (40 minutes)

Higher-order functions are functions that operate on other functions, either by taking them as arguments or by returning them.

In JavaScript, three commonly used higher-order functions are `map()`, `filter()`, and `reduce()`. These functions are particularly useful for working with arrays.

1. `map()`

The `map()` method creates a new array by applying a provided function to every element in the original array.

Syntax:

```
let newArray = array.map(callback);
```

- `callback`: A function that takes three arguments:
  - `currentValue`: The current element being processed.
  - `index` (optional): The index of the current element.
  - `array` (optional): The array `map` was called upon.

Example:

```
let numbers = [1, 2, 3, 4]; let squaredNumbers = numbers.map(function (num) { return num * num; }); console.log(squaredNumbers); // Output: [1, 4, 9, 16]
```

2. `filter()`

The `filter()` method creates a new array with all elements that pass the test implemented by the provided function.

Syntax:

```
javascriptCopy code let newArray = array.filter(callback);
```

- `callback` : A function that takes three arguments:
  - `currentValue` : The current element being processed.
  - `index` (optional): The index of the current element.
  - `array` (optional): The array `filter` was called upon.
  - The function should return `true` to keep the element, or `false` otherwise.

Example:

```
let numbers = [1, 2, 3, 4, 5, 6]; let evenNumbers = numbers.filter(fun
ction(num) { return num % 2 === 0; }); console.log(evenNumbers); // Ou
tput: [2, 4, 6]
```

3. `reduce()`

The `reduce()` method executes a reducer function (that you provide) on each element of the array, resulting in a single output value.

Syntax:

```
let result = array.reduce(callback, initialValue);
```

- `callback` : A function that takes four arguments:
  - `accumulator` : Accumulates the callback's return values; it is the accumulated value previously returned in the last invocation.
  - `currentValue` : The current element being processed.
  - `index` (optional): The index of the current element.
  - `array` (optional): The array `reduce` was called upon.
- `initialValue` (optional): A value to use as the first argument to the first call of the `callback` . If no initial value is supplied, the first element in the array will be used.

Example:

```
javascriptCopy code let numbers = [1, 2, 3, 4]; let sum = numbers.redu
ce(function(accumulator, currentValue) { return accumulator + currentV
alue; }, 0); console.log(sum); // Output: 10
```

Summary

- `map()` : Transforms each element in the array and returns a new array.
  - Example use case: Converting an array of numbers to their squares.
- `filter()` : Filters elements in the array based on a condition and returns a new array.
  - Example use case: Extracting all even numbers from an array.
- `reduce()` : Reduces the array to a single value by accumulating the results of applying a function.
  - Example use case: Summing all numbers in an array.

These higher-order functions allow for concise and expressive operations on arrays, making your code more readable and functional.

7. Object Declaration & Accessing Properties

In JavaScript, objects are used to store collections of data and more complex entities.

Objects are made up of key-value pairs where the key (also called a property) is a string (or symbol), and the value can be any data type, including another object or a function.

Object Declaration

There are multiple ways to declare an object in JavaScript.

1. Object Literal Syntax

The most common way to create an object is by using the object literal syntax.

Example:

```
let person = { firstName: "John", lastName: "Doe", age: 30, isEmployed: true };
```

In this example:

- `firstName`, `lastName`, `age`, and `isEmployed` are properties of the `person` object.
- `"John"`, `"Doe"`, `30`, and `true` are the values associated with those properties.

2. Using the `new Object()` Syntax

You can also create an object using the `new Object()` constructor, though this is less common.

Example:

```
let person = new Object(); person.firstName = "John"; person.lastName = "Doe"; person.age = 30; person.isEmployed = true;
```

Accessing Object Properties

There are two main ways to access properties in an object:

1. Dot Notation

Dot notation is the most common and straightforward way to access or modify an object's properties.

Example:

```
console.log(person.firstName); // Output: "John" console.log(perso
n.age); // Output: 30 // Modify a property person.age = 31; consol
e.log(person.age); // Output: 31
```

2. Bracket Notation

Bracket notation allows you to access properties using a string key. This is useful when the property name is stored in a variable or contains special characters.

Example:

```
console.log(person["lastName"]); // Output: "Doe" console.log(pers
on["isEmployed"]); // Output: true // Modify a property person["is
Employed"] = false; console.log(person["isEmployed"]); // Output:
false
```

Dynamic Property Access:

```
let key = "firstName"; console.log(person[key]); // Output: "John"
```

Adding New Properties

You can add new properties to an object at any time.

Example:

```
person.nationality = "American"; console.log(person.nationality);
// Output: "American"
```

Deleting Properties

You can delete properties from an object using the `delete` keyword.

Example:

```
delete person.age; console.log(person.age); // Output: undefined
```

Checking if a Property Exists

You can check if a property exists in an object using the `in` operator or the `hasOwnProperty()` method.

Example:

```
console.log("firstName" in person); // Output: true console.log(pe
rson.hasOwnProperty("age")); // Output: false
```

Nested Objects

Objects can also contain other objects as properties, creating a nested structure.

Example:

```
let student = { name: "Alice", age: 20, address: { street: "123 Ma
in St", city: "New York", zip: "10001" } }; console.log(student.ad
dress.city); // Output: "New York"
```

Summary

- **Declaration:** Objects can be created using object literal syntax `{}` or the `new Object()` constructor.
- **Accessing Properties:** Use dot notation (`object.property`) or bracket notation (`object["property"]`) to access or modify properties.
- **Adding/Deleting Properties:** You can dynamically add new properties or delete existing ones using the `delete` keyword.
- **Nested Objects:** Objects can have properties that are other objects, allowing for a nested structure.

8. Object Methods and Uses

---

1. Defining Object Methods

An object method is simply a function defined as a property of an object. Here's how you can define and use an object method:

```
const user = { name: 'Alice', age: 25, greet: function() { return `Hello, my name is ${this.name}.`; } }; console.log(user.greet()); // Output: Hello, my name is Alice.
```

In the example above, `greet` is an object method that returns a greeting message. The `this` keyword is used to refer to the current object (`user` in this case).

---

2. Common Object Methods

Object methods can be used to manipulate the object's properties or perform specific actions. Some common use cases include:

1. **Updating Object Properties**: Methods can update the properties of an object.

```
const car = { make: 'Toyota', model: 'Corolla', year: 2015, updateYear: function(newYear) { this.year = newYear; } }; car.updateYear(2020); console.log(car.year); // Output: 2020
```

1. **Calculating Values Based on Object Data**: Methods can perform calculations based on the object's properties.

```
const rectangle = { width: 10, height: 5, area: function() { return this.width * this.height; } }; console.log(rectangle.area()); // Output: 50
```

1. **Combining Object Data**: Methods can combine data from multiple properties to produce a new result.

```
const person = { firstName: 'John', lastName: 'Doe', fullName: fun
ction() { return `${this.firstName} ${this.lastName}`; } }; consol
e.log(person.fullName()); // Output: John Doe
```

---

3. `this` Keyword in Object Methods

The `this` keyword is essential when working with object methods. It refers to the object that is currently calling the method. The value of `this` depends on how the method is called.

Example:

```
const book = { title: 'JavaScript Basics', author: 'Jane Doe', get
Summary: function() { return `${this.title} by ${this.author}`; }
}; console.log(book.getSummary()); // Output: JavaScript Basics by
Jane Doe
```

Here, `this.title` refers to `book.title`, and `this.author` refers to `book.author`.

Important Note:
When using `this` inside an object method, make sure the method is called on the object itself. If you assign the method to a variable and then call it, `this` might refer to the global object (`window` in a browser), which can lead to unexpected results.