Module 6 – Mernstack

Javascript Essential and Advanced

JavaScript Introduction

1. What is JavaScript? Explain the role of JavaScript in web development.

JavaScript is a programming language used to make websites interactive. While HTML builds the structure and CSS styles the appearance, JavaScript adds dynamic features like animations, buttons that respond to clicks, forms that validate inputs, and more.

It's like the brain of a website, allowing it to do things beyond just displaying text and images.

For example, JavaScript can:

- Make buttons clickable
- Create animations
- Update content without reloading the page
- Validate user input in forms

---

2 . How is JavaScript different from other programming languages like Python or Java?

- **JavaScript** is primarily for making websites interactive. It runs directly in your web browser.
- **Python** is versatile, used for things like data science, web development (on the server-side), and machine learning.

- **Java** is powerful and used for building large applications like mobile apps and enterprise software.

---

3. Discuss the use of <script>tag in HTML. How can you link an external

JavaScript file to an HTML document?

The <script> tag in HTML is used to include JavaScript code within an HTML document.

To link an external JavaScript file:

1. **Create a separate file:** Save your JavaScript code in a file with the .js extension (e.g., script.js).
2. **Use the src attribute:** In your HTML file, use the <script> tag with the src attribute to specify the path to your JavaScript file:

```
<script src="script.js"></script>
```

Variables and Data Types

---

4. What are variables in JavaScript? How do you declare a variable using var, let, and const?

Variables in JavaScript are like containers that store data. This data can be anything like numbers, text (called strings), or even more complex things.

Here's how to declare variables:

- var: This is the older way. Variables declared with var can be re-declared and re-assigned within the same scope.

```
var name = "Alice";
name = "Bob"; // Can be re-assigned
```

- **let:** A more modern approach. Variables declared with let can be re-assigned but cannot be re-declared within the same scope.

```
let age = 30;
age = 31; // Can be re-assigned
```

- const: Used for values that should never change.

```
const pi = 3.14159;
// Cannot be re-assigned
```

---

5. **Explain the different data types in JavaScript. Provide examples for each.**

Data Types in JavaScript are like different types of information you can store. Here are a few common ones:

- **Number:** Represents numerical values.
    - Example: let age = 30; let price = 9.99;
- **String:** Represents text.
    - Example: let name = "Alice"; let message = "Hello!";
- **Boolean:** Represents either true or false.
    - Example: let isLoggedIn = true; let isAvailable = false;
- **Array:** An ordered collection of values.
    - Example: let colors = ["red", "green", "blue"];
- **Object:** A collection of key-value pairs.

```
let person = {
    name: "Alice",
    age: 30
};
```

---

6. What is the difference between undefined and null in JavaScript?

- Undefined : A variable is declared but not assigned a value yet. JavaScript sets it automatically undefined.

```
let x; // x is undefined
```

- Null : Represents an **intentional absence of value**. You set it manually when you want to clear a variable.

```
let y = null; // y is explicitly empty
```

---

7. What are the different types of operators in JavaScript? Explain with examples.

- **Arithmetic Operators:**
    - + (addition): 5 + 3 = 8
    - - (subtraction): 10 - 4 = 6
    - * (multiplication): 2 * 6 = 12
    - / (division): 15 / 3 = 5
    - % (modulus - returns the remainder): 7 % 3 = 1

- **Comparison Operators:**
    - == (equal to): 5 == 5 // true
    - != (not equal to): 5 != 3 // true
    - > (greater than): 10 > 5 // true
    - < (less than): 3 < 7 // true
    - >= (greater than or equal to): 10 >= 10 // true
    - <= (less than or equal to): 5 <= 5 // true
- **Logical Operators:**

- && (and): true && true // true
- || (or): true || false // true
- ! (not): !true // false

- **Assignment operators:**
  - let x = 10; // x is assigned the value 10.
  - let name = "Alice"; // name is assigned the string value "Alice".
  - let isTrue = true; // isTrue is assigned the boolean value true.
  - let result = x + 5; // result is assigned the result of the expression x + 5 which is 15.

---

8. What is the difference between ==and ===in JavaScript?

- == (double equals): Checks if two values are equal after type conversion.
  - For example, "5" (string) == 5 (number) will be true because JavaScript tries to convert the string "5" to the number 5 before comparing.
- === (triple equals): Checks if two values are equal in both value and type.
  - For example, "5" (string) === 5 (number) will be false because they are different data types (string vs. number).

Control Flow (If-Else, Switch)

---

9. What is control flow in JavaScript? Explain how if-else statements work with an example.

Control Flow in JavaScript is how your code decides which parts to execute and in what order. It's like a roadmap for your program!

if-else Statements are a key part of control flow. They allow your code to make decisions based on conditions.

Here's the basic structure:

```
if (condition) {
  // Code to execute if the condition is true
} else {
  // Code to execute if the condition is false
}
```

Example:

```
let age = 18;

if (age >= 18) {
  console.log("You are an adult.");
} else {
  console.log("You are a minor.");
}
```

---

10. Describe how switch statements work in JavaScript. When should you use a switch statement instead of if-else?

Switch Statements in JavaScript are like a more efficient way to handle multiple if-else conditions when you're comparing the same value against different cases.

Here's the basic structure:

```
switch(expression) {
  case value1:
    // Code to execute if expression matches value1
    break;
  case value2:
    // Code to execute if expression matches value2
    break;
  case value3:
    // Code to execute if expression matches value3
    break;
  default:
    // Code to execute if no case matches
}
```

Example:

```javascript
let day = "Sunday";

switch(day) {
  case "Monday":
    console.log("It's Monday!");
    break;
  case "Tuesday":
    console.log("It's Tuesday!");
    break;
  case "Sunday":
    console.log("It's the weekend!");
    break;
  default:
    console.log("It's another day.");
}
```

---

11. Explain the different types of loops in JavaScript (for, while, do-while) Provide a basic example of each.

Loops in JavaScript are used to repeat a block of code multiple times. Here are the main types:

- **for loop:**
    - Used when you know the number of times you want to repeat the code.
    - Example:

```javascript
for (let i = 0; i < 5; i++) {
  console.log(i); // Prints numbers from 0 to 4
}
```

- **while loop:**
    - Repeats the code as long as a given condition is true.
    - Example:

```
let count = 0;
while (count < 5) {
  console.log(count);
  count++;
}
```

- **do-while loop:**
  - Similar to while, but guarantees that the code inside the loop will execute at least once.
  - **Example:**

```
let i = 0;
do {
  console.log(i);
  i++;
} while (i < 5);
```

---

12. What is the difference between a while loop and a do-while loop?

- **while loop:** Checks the condition before running the code inside. If the condition is false initially, the code inside the loop won't run at all.
- **do-while loop:** Runs the code once and then checks the condition. This guarantees that the code inside will always execute at least once, even if the condition is initially false.

Functions

---

13. What are functions in JavaScript? Explain the syntax for declaring and calling a function.

Functions in JavaScript are reusable blocks of code that perform a specific task.

Declaring a Function:

```
function functionName() {
  // Code to be executed
}
```

- function: Keyword to define a function.
- functionName: The name you give to your function (e.g., greet, calculate).
- {}: The code block that contains the function's instructions.

Calling a Function:

```
functionName();
```

- Simply use the function's name followed by parentheses () to execute it.

Example:

```
function greet() {
  console.log("Hello!");
}

greet(); // This will print "Hello!" to the console
```

In short: Functions help you organize your code, make it more reusable, and avoid repeating the same code multiple times.

---

14. **What is the difference between a function declaration and a Function expression?**

- **Function Declaration:**
  - Declared with the function keyword followed by the function name.
  - Can be called before they are defined (hoisted).

    Example: function greet() { ... }
- **Function Expression:**
  - Assigned to a variable.
  - Not hoisted, so you can't call them before they are defined.
  - Can be anonymous (no name).

    Example: const greet = function() { ... }

In short: Declarations are named and hoisted, expressions are more flexible and can be anonymous.

---

15. **Discuss the concept of parameters and return values in functions.**

Parameters are like ingredients for your function. When you define a function, you can specify parameters inside the parentheses. These parameters act as placeholders for values that you'll provide when you call the function.

Example:

```
function greet(name) {
  console.log("Hello, " + name + "!");
}
```

In this example, name is a parameter. When you call this function, you'll provide a name (like "Alice") as an argument:

```
greet("Alice"); // Output: Hello, Alice!
```

Return Values are like the result of a recipe. A function can return a value using the return keyword. This value can then be used in other parts of your code.

Example:

```
function add(a, b) {
  return a + b;
}

let sum = add(5, 3); // sum will be 8
```

In this example, the add function takes two parameters (a and b) and returns their sum.

In simple words:

- **Parameters** are like the ingredients you give to a function.
- **Return values** are like the finished product that the function creates.

Arrays

16. What is an array in JavaScript? How do you declare and initialize an array?

In JavaScript, an **array** is like a list of items stored in a single variable. Think of it as a container that can hold multiple values, like a shopping list or a collection of numbers.

Declaring and Initializing an Array:

You can declare an array using square brackets `[]`:

- **Empty Array:**

```
let myArray = [];
```

  This creates an empty array with no items in it.

- **Array with Values:**

```
let fruits = ["apple", "banana", "orange"];
```

This creates an array named fruits containing three strings: "apple", "banana", and "orange".

In short: An array is a collection of values, and you declare it using square brackets `[]`.

---

17. **Explain the methods push(), pop(), shift(), and unshift()used in arrays.**

push()

- **What it does:** Adds an element to the **end** of the array.
- **Example:**

```
let fruits = ["apple", "banana"];
fruits.push("orange");
console.log(fruits); // Output: ["apple", "banana", "orange"]
```

pop()

- **What it does:** Removes the **last** element from the array.
- **Example:**

```
let fruits = ["apple", "banana", "orange"];
fruits.pop();
console.log(fruits); // Output: ["apple", "banana"]
```

shift()

- **What it does:** Removes the **first** element from the array.
- **Example:**

```
let fruits = ["apple", "banana", "orange"];
fruits.shift();
console.log(fruits); // Output: ["banana", "orange"]
```

unshift()

- **What it does:** Adds an element to the **beginning** of the array.
- **Example:**

```
let fruits = ["banana", "orange"];
fruits.unshift("apple");
console.log(fruits); // Output: ["apple", "banana", "orange"]
```

Objects

---

18. What is an object in JavaScript? How are objects different from arrays?

In JavaScript, an **object** is like a container that holds a collection of **key-value pairs**.

Think of it like a dictionary where each word (**key**) has a definition (**value**).

Here's how they differ from arrays:

- **Arrays** are ordered lists of values.

- **Objects** are unordered collections of key-value pairs.

Example:

- **Array:** ["apple", "banana", "orange"] (ordered list of fruits)
- **Object:**

```
let person = {
    name: "Alice",
    age: 30,
    city: "New York"
};
```

In short: Arrays are like lists, while objects are like dictionaries with key-value pairs.

---

19. **Explain how to access and update object properties using dot notation and bracket notation.**

- Dot Notation:

```
let person = { name: "Alice", age: 30 };
let name = person.name; // Accesses the "name" property
```

- Bracket Notation:

```
let person = { name: "Alice", age: 30 };
let age = person["age"]; // Accesses the "age" property
```

- Bracket notation is useful when you have property names that are dynamically determined or contain special characters.

Updating Properties:

- **Dot Notation:**

```
person.age = 31; // Updates the "age" property
```

- Bracket Notation:

```
person["city"] = "New York"; // Adds a new property "city"
```

In short: Dot notation is generally preferred for simple property names, while bracket notation is more flexible for dynamic or complex property names.

JavaScript Events

---

20. **What are JavaScript events? Explain the role of event listeners.**

In JavaScript, events are actions that occur as a result of user interactions or system-generated activities. These interactions can include:

- **User Interactions:** Clicking a button, hovering over an element, typing in a text field, scrolling the page, etc.
- **System Events:** Page loading, window resizing, timer expiration, etc.

Event Listeners

Event listeners are like watchful guards that monitor elements for specific events. When an event occurs on an element with an attached event listener, the listener triggers a predefined action, such as executing a function.

---

21. How does the addEventListener() method work in JavaScript? Provide an example.

Imagine you have a button on a webpage. You want something to happen when someone clicks it, like showing a message.

- addEventListener() is like putting a tiny listener on that button.
- This listener is always watching for a specific event, like a click.
- When the event happens (someone clicks), the listener wakes up and calls a special function.
- This function does the action you want, like showing the message.

Example:

1. Get the Button: Find the button on the page using its ID (like "myButton").

2. Create a Function: Make a function that does what you want (like showing an alert).
3. Add the Listener: Use addEventListener() to tell the button: "Hey, listen for clicks, and when someone clicks, run that function!"

That's it! Now, whenever someone clicks the button, the message will appear.

Key Points:

- addEventListener() is like setting up a trigger.
- You can use it for many events, not just clicks (like hovering, typing, etc.).
- It makes your webpage more interactive and responsive.

```
<body>

  <button id="myButton">Click Me</button>

  <script>
    // Get the button element
    const button = document.getElementById("myButton");

    // Function to be executed when the button is clicked
    function buttonClicked() {
      alert("Button clicked!");
    }

    // Add an event listener to the button
    button.addEventListener("click", buttonClicked);
  </script>

</body>
```

DOM Manipulation

22. **What is the DOM (Document Object Model) in JavaScript? How does JavaScript interact with the DOM?**

What is the DOM?

- Imagine a webpage as a tree-like structure. The DOM is like a map of this structure.
- It represents every single element on the page (like headings, paragraphs, images, buttons) as objects.
- These objects have properties (like their ID, class, content) and methods (like changing their style, adding/removing content).

How JavaScript interacts with the DOM:

- **Access Elements:** JavaScript can use methods like `getElementById()` , `getElementsByClassName()`, or `querySelector()` to find specific elements within the DOM.
- **Modify Elements:** Once an element is found, JavaScript can change its:
    - **Content:** Update the text inside the element.
    - **Style:** Modify its appearance (color, size, position).
    - **Attributes:** Change properties like `href` (for links) or `src` (for images).
    - **Structure:** Add, remove, or rearrange elements within the page.
- **Events:** JavaScript can respond to user interactions (like clicks, hovers, key presses) on DOM elements. This allows for dynamic and interactive web pages.

In essence: The DOM provides a structured way for JavaScript to "talk" to the webpage and make it dynamic and responsive.

---

23. **Explain the methods getElementById(), getElementsByClassName(),and querySelector() used to select elements from the DOM.**

1. `getElementById()`

- **Purpose:** Selects a single element based on its unique id attribute.
- **Example:**

HTML

```html
<p id="myParagraph">This is a paragraph.</p>
```

JavaScript

```javascript
const paragraph = document.getElementById("myParagraph");
```

2. getElementsByClassName()

- **Purpose:** Selects a collection of elements that share the same class name.
- **Example:**

HTML

```html
<p class="myClass">Paragraph 1</p>
<p class="myClass">Paragraph 2</p>
```

JavaScript

```javascript
const paragraphs = document.getElementsByClassName("myClass");
```

(This returns a collection, so you might use a loop to iterate through it)

3. querySelector()

- **Purpose:** Selects the first element that matches a CSS selector. This is very versatile!
- **Examples:**

```html
<p id="myParagraph">This is a paragraph.</p>
<div class="container">
    <p>Another paragraph.</p>
</div>
```

- Select by ID: `document.querySelector("#myParagraph")`
- Select by class: `document.querySelector(".myClass")`
- Select by tag name: `document.querySelector("p")`
- More complex selectors: `document.querySelector("div > p")` (selects the first paragraph within a div)

JavaScript Timing Events (setTimeout, setInterval)

---

24. **Explain the setTimeout() and setInterval() functions in JavaScript. How are they used for timing events?**

1. `setTimeout()`

- **What it does:** Executes a piece of code (a function) **once** after a specified delay (in milliseconds).
- **Analogy:** Imagine setting a timer on your microwave. It will beep only *once* after the time is up.

2. `setInterval()`

- **What it does:** Executes a piece of code (a function) **repeatedly** at a specified interval (in milliseconds).
- **Analogy:** Imagine setting an alarm clock to repeat every hour.
- **Example:**

```
setInterval(() => {
    console.log("This message will repeat every second.");
}, 1000); // 1000 milliseconds = 1 second
```

Key Differences:

- **`setTimeout()` :** One-time execution after a delay.
- **`setInterval()` :** Repeated execution at regular intervals.

---

25. Provide an example of how to use setTimeout() to delay an action by 2 seconds.

```
function greet() {
  alert("Hello!");
}

setTimeout(greet, 2000); // 2000 milliseconds = 2 seconds
```

Explanation:

1. **greet() function:** This is the function that will be executed after the delay. In this case, it simply displays an alert box with the message "Hello!".
2. **setTimeout(greet, 2000);:**
    - greet: This is the name of the function that should be executed.
    - 2000: This is the delay in milliseconds (2 seconds).

This code will wait for 2 seconds before displaying the "Hello!" alert.

Key Points:

- You can use any function within setTimeout().
- The delay is specified in milliseconds.
- setTimeout() provides a way to schedule actions to happen at a later time.

JavaScript Error Handling

---

26. **What is error handling in JavaScript? Explain the try, catch, and finally blocks with an example.**

Error Handling in JavaScript : It's like a safety net for your JavaScript code. It helps you gracefully deal with unexpected situations (like invalid user input, missing files, network issues) that might cause your code to crash.

try, catch, and finally

- **try block:** This is where you place the code that might potentially throw an error.
- **catch block:** This block executes only if an error occurs within the try block. You can use the error object (usually named error) to get information about the error (like the error message).
- **finally block:** This block executes regardless of whether an error occurred or not. It's often used for cleanup tasks (like closing files or releasing resources)

```
try {
  // Code that might throw an error
  const result = 10 / 0; // This will cause a "Divide by zero" error
  console.log(result);
} catch (error) {
  console.error("An error occurred:", error);
} finally {
  console.log("This block always executes.");
}
```

- try : Attempt to execute code.
- Catch : Handle errors if they happen.
- Finally : Always execute, for cleanup.

---

27. Why is error handling important in JavaScript applications?

- **Keeps your code from crashing:** Like a safety net for your code.
- **Provides helpful messages:** Tells the user what went wrong.
- **Makes your code more reliable:** Prevents unexpected breakdowns.
- **Helps you find problems:** Easier to fix bugs.

In short: Makes your JavaScript applications more stable and user-friendly.
