**MERN Stack – JavaScript Essential and Advanced**

**JavaScript Introduction**

**Theory Assignment**

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

JavaScript (JS) is a high-level, interpreted programming language that is one of the core technologies of the World Wide Web, alongside HTML and CSS. While HTML provides the basic structure of web pages and CSS controls the presentation and layout, **JavaScript is used to make web pages interactive and dynamic**.

The role of JavaScript in web development includes:

* **Adding Interactivity:** It allows you to create responsive user interfaces, such as reacting to button clicks, mouse movements, or keyboard input.
* **Manipulating the DOM:** JS can add, delete, and change HTML elements and their attributes on a page *after* it has loaded, allowing for dynamic content updates without needing to reload the page.
* **Handling Events:** It can listen for user actions (like clicks, form submissions, etc.) and execute code in response.
* **Asynchronous Communication (AJAX):** It can fetch data from a server in the background (e.g., loading new posts or data) and update parts of a web page without a full page refresh.
* **Creating Animations & Effects:** JS can be used to create sliders, pop-up modals, animations, and other visual effects.

**Question 2: How is JavaScript different from other programming languages like Python or Java?**

| Feature | JavaScript | Python | Java |
| --- | --- | --- | --- |
| **Primary Use** | Client-side web development (browsers), but also server-side (Node.js). | General purpose: web development (Django, Flask), data science, AI, scripting. | Enterprise-level applications, Android apps, large-scale systems. |
| **Typing** | **Dynamically Typed:** Variable types are checked at runtime. let x = 5; x = "hello"; is valid. | **Dynamically Typed:** Similar to JavaScript. | **Statically Typed:** Variable types must be declared and are checked at compile time. int x = 5; x = "hello"; will cause an error. |
| **Execution** | **Interpreted:** Executed directly by an engine (like V8 in Chrome) in the browser or on a server. No compilation step is needed. | **Interpreted:** Executed by a Python interpreter. | **Compiled & Interpreted:** Source code is first compiled into bytecode, which is then interpreted by the Java Virtual Machine (JVM). |
| **Concurrency** | Single-threaded with an **event loop** for handling asynchronous operations. | Uses threading and multiprocessing, but can be limited by the Global Interpreter Lock (GIL). | Strong support for multi-threading for parallel processing. |
| **Environment** | Runs primarily in web browsers and on servers via Node.js. | Runs on almost any machine with a Python interpreter installed. | Runs on any machine with the Java Virtual Machine (JVM) installed ("write once, run anywhere"). |

Export to Sheets

**Question 3: Discuss the use of the <script> tag in HTML. How can you link an external JavaScript file to an HTML document?**

The **<script>** tag is used to embed or reference executable JavaScript code within an HTML document. There are two main ways to use it:

1. **Internal/Inline JavaScript:** You can place JavaScript code directly inside the <script> tag within your HTML file. This is useful for small, page-specific scripts.

HTML

<script>

// JavaScript code goes here

console.log("This is internal JavaScript!");

</script>

1. **External JavaScript:** You can link to an external JavaScript file (.js extension). This is the recommended approach for larger scripts as it keeps your code organized, reusable, and allows the browser to cache the file.

To link an external JavaScript file, you use the <script> tag with the src (source) attribute.

**Example:** If you have a file named main.js in the same directory as your HTML file, you would link it like this:

HTML

<!DOCTYPE html>

<html>

<head>

<title>My Page</title>

</head>

<body>

<h1>My Website</h1>

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

</body>

</html>

**Lab Assignment**

**Task:** Create a simple HTML page and add a <script> tag within the page. Write JavaScript code to display an alert box with the message "Welcome to JavaScript!" when the page loads.

**Solution:** Create an HTML file named index.html with the following code. When you open this file in a web browser, an alert box will appear immediately.

index.html

HTML

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>JavaScript Welcome</title>

</head>

<body>

<h1>Welcome to My Page</h1>

<p>An alert will show when this page loads.</p>

<script>

alert("Welcome to JavaScript!");

</script>

</body>

</html>

**Variables and Data Types**

**Theory Assignment**

**Question 1: What are variables in JavaScript? How do you declare a variable using var, let, and const?**

In JavaScript, **variables** are containers for storing data values. They are symbolic names for values in your application.

You can declare variables using three keywords: var, let, and const.

1. **var**:
   * This was the original way to declare variables.
   * Variables declared with var are **function-scoped**, meaning they are accessible anywhere within the function they are declared in. If declared outside any function, they are globally scoped.
   * var variables are "hoisted," meaning their declaration is moved to the top of their scope, but not their initialization.
   * **Example:** var name = "John";
2. **let**:
   * Introduced in ES6 (2015), let is the modern way to declare variables that can be reassigned.
   * Variables declared with let are **block-scoped**, meaning they are only accessible within the block ({...}) they are defined in (e.g., inside an if statement or a for loop).
   * **Example:** let age = 30;
3. **const**:
   * Also introduced in ES6, const is used to declare variables whose values are not intended to be changed (constants).
   * They are also **block-scoped**.
   * A const variable must be initialized at the time of declaration and cannot be reassigned.
   * **Example:** const PI = 3.14;

**Question 2: Explain the different data types in JavaScript. Provide examples for each.**

JavaScript has two main categories of data types: **Primitive Types** and **Object Type**.

**Primitive Data Types:**

1. **String:** A sequence of characters used for text. Must be in quotes.
   * *Example:* let name = "Alice";
2. **Number:** Represents both integer and floating-point numbers.
   * *Example:* let age = 25; let price = 99.99;
3. **Boolean:** Represents a logical entity and can have one of two values: true or false.
   * *Example:* let isLoggedIn = true;
4. **Undefined:** A variable that has been declared but has not yet been assigned a value has the type undefined.
   * *Example:* let user; // user is undefined
5. **Null:** Represents the intentional absence of any object value. It's a special value that means "no value" or "empty."
   * *Example:* let data = null;
6. **Symbol:** A unique and immutable primitive value that may be used as the key of an Object property.
   * *Example:* let id = Symbol('id');
7. **BigInt:** Used to store integer values that are too large to be represented by the Number type.
   * *Example:* const veryLargeNumber = 1234567890123456789012345678901234567890n;

**Non-Primitive Data Type:**

1. **Object:** A collection of key-value pairs. Arrays, functions, and dates are also objects in JavaScript.
   * *Example:* let person = { firstName: "John", lastName: "Doe" };

**Question 3: What is the difference between undefined and null in JavaScript?**

While both undefined and null represent the absence of a value, they are used in different contexts.

* **undefined** typically means a value has **not been assigned**. It's the default value of a declared variable that hasn't been initialized. It can also be the return value of a function that doesn't explicitly return anything. It's an accidental lack of value.
* **null** is an **explicit assignment**. It's a value you, as the programmer, intentionally assign to a variable to signify that it holds "no value" or is empty. It's a deliberate lack of value.

| Feature | undefined | null |
| --- | --- | --- |
| **Meaning** | Value is not assigned. | Value is explicitly set to nothing. |
| **Type** | The type is undefined. | The type is object (this is a long-standing quirk in JS). |
| **Usage** | Default value of uninitialized variables. | Intentionally assigned to represent "no value". |

Export to Sheets

**Example:**

JavaScript

let a; // a is undefined

let b = null; // b is null

**Lab Assignment**

**Task:** Write a JavaScript program to declare variables for different data types (string, number, boolean, null, and undefined). Log the values of the variables and their types to the console using console.log().

**Solution:** Create a JavaScript file named datatypes.js and run it (e.g., using Node.js or by linking it to an HTML file and opening the browser console).

datatypes.js

JavaScript

// 1. String

let myName = "John Doe";

// 2. Number

let myAge = 30;

// 3. Boolean

let isStudent = false;

// 4. Null

let noValue = null;

// 5. Undefined

let notAssigned;

// Log the values and their types to the console

console.log("Value:", myName, ", Type:", typeof myName);

console.log("Value:", myAge, ", Type:", typeof myAge);

console.log("Value:", isStudent, ", Type:", typeof isStudent);

console.log("Value:", noValue, ", Type:", typeof noValue); // Note: typeof null is 'object'

console.log("Value:", notAssigned, ", Type:", typeof notAssigned);

**Console Output:**

Value: John Doe , Type: string

Value: 30 , Type: number

Value: false , Type: boolean

Value: null , Type: object

Value: undefined , Type: undefined

**JavaScript Operators**

**Theory Assignment**

**Question 1: What are the different types of operators in JavaScript? Explain with examples.**

Operators are special symbols used to perform operations on operands (values and variables).

* **Arithmetic Operators:** Used to perform mathematical calculations.
  + + (Addition): 5 + 3; // 8
  + - (Subtraction): 5 - 3; // 2
  + \* (Multiplication): 5 \* 3; // 15
  + / (Division): 15 / 3; // 5
  + % (Modulus - remainder of division): 5 % 3; // 2
  + \*\* (Exponentiation): 5 \*\* 2; // 25
  + ++ (Increment): let a = 5; a++; // a is now 6
  + -- (Decrement): let b = 5; b--; // b is now 4
* **Assignment Operators:** Used to assign values to variables.
  + = (Assignment): let x = 10;
  + += (Add and assign): x += 5; // equivalent to x = x + 5;
  + -= (Subtract and assign): x -= 5; // equivalent to x = x - 5;
  + \*= (Multiply and assign): x \*= 2; // equivalent to x = x \* 2;
  + /= (Divide and assign): x /= 2; // equivalent to x = x / 2;
* **Comparison Operators:** Used to compare two values, resulting in a boolean (true or false).
  + == (Equal to): 5 == '5'; // true (type coercion)
  + === (Strictly equal to - value and type): 5 === '5'; // false
  + != (Not equal to): 5 != '6'; // true
  + !== (Strictly not equal to): 5 !== '5'; // true
  + > (Greater than): 10 > 5; // true
  + < (Less than): 10 < 5; // false
  + >= (Greater than or equal to): 10 >= 10; // true
  + <= (Less than or equal to): 5 <= 10; // true
* **Logical Operators:** Used to combine or invert boolean values.
  + && (Logical AND): (5 > 3) && (10 > 5); // true (both conditions are true)
  + || (Logical OR): (5 > 10) || (10 > 5); // true (at least one condition is true)
  + ! (Logical NOT): !(5 > 10); // true (inverts false to true)

**Question 2: What is the difference between == and === in JavaScript?**

The primary difference between == (loose/abstract equality) and === (strict equality) lies in how they handle data types during comparison.

* **== (Loose Equality):** This operator compares two values for equality **after** performing type coercion if the types are different. This means it might convert one of the values to match the other's type before comparing.

**Example:**

JavaScript

5 == "5"; // true, because the string "5" is converted to the number 5 before comparison.

0 == false; // true, because false is converted to the number 0.

null == undefined; // true

* **=== (Strict Equality):** This operator compares both the **value and the data type** of the two operands. No type conversion is performed. If the types are different, the result is always false. It is generally recommended to use === to avoid unexpected bugs from type coercion.

**Example:**

JavaScript

5 === "5"; // false, because one is a number and the other is a string.

0 === false; // false, because one is a number and the other is a boolean.

null === undefined; // false

**Lab Assignment**

**Task:** Create a JavaScript program to perform various operations using arithmetic, comparison, and logical operators.

**Solution:** operators.js

JavaScript

// Define two numbers

let num1 = 20;

let num2 = 10;

let a = 15;

let b = 4;

console.log("----- Arithmetic Operators -----");

console.log("Addition (num1 + num2):", num1 + num2); // 30

console.log("Subtraction (num1 - num2):", num1 - num2); // 10

console.log("Multiplication (num1 \* num2):", num1 \* num2); // 200

console.log("Division (num1 / num2):", num1 / num2); // 2

console.log("\n----- Comparison Operators -----");

console.log("Are num1 and 20 equal? (num1 == 20):", num1 == 20); // true

console.log("Is num1 greater than num2? (num1 > num2):", num1 > num2); // true

let strNum = "20";

console.log("Loose equality (num1 == strNum):", num1 == strNum); // true

console.log("Strict equality (num1 === strNum):", num1 === strNum); // false

console.log("\n----- Logical Operators -----");

// Check if a is greater than 10 AND b is less than 5

let condition1 = a > 10; // true

let condition2 = b < 5; // true

console.log("Is a > 10 AND b < 5?", condition1 && condition2); // true

// Check if a is greater than 20 OR b is less than 5

let condition3 = a > 20; // false

console.log("Is a > 20 OR b < 5?", condition3 || condition2); // true

**Control Flow (If-Else, Switch)**

**Theory Assignment**

**Question 1: What is control flow in JavaScript? Explain how if-else statements work with an example.**

**Control flow** is the order in which a computer executes statements in a script. By default, code is executed line by line from top to bottom. However, control flow statements allow you to alter this sequence, enabling the program to make decisions, execute code conditionally, or repeat actions.

The **if-else** statement is a fundamental control flow structure that executes a block of code if a specified condition is true. If the condition is false, another block of code can be executed.

**Syntax:**

JavaScript

if (condition) {

// block of code to be executed if the condition is true

} else if (anotherCondition) {

// block of code to be executed if the first condition is false

// and anotherCondition is true

} else {

// block of code to be executed if all preceding conditions are false

}

**Example:** Checking if a person is old enough to vote.

JavaScript

let age = 19;

if (age >= 18) {

console.log("You are eligible to vote.");

} else {

console.log("You are not eligible to vote yet.");

}

// Output: "You are eligible to vote."

**Question 2: Describe how switch statements work in JavaScript. When should you use a switch statement instead of if-else?**

A **switch** statement evaluates an expression, matching the expression's value against a series of case clauses. It executes statements associated with the first matching case clause, as well as statements in clauses that follow the matching clause until a break statement is encountered.

**Syntax:**

JavaScript

switch (expression) {

case value1:

// Statements to execute when expression matches value1

break;

case value2:

// Statements to execute when expression matches value2

break;

// ... more cases

default:

// Statements to execute if no case matches

}

* The break keyword is crucial; without it, the code will "fall through" and execute the code in the next case as well.
* The default clause is optional and runs if no case matches.

**When to use switch over if-else:** You should use a switch statement when you are comparing a **single variable** against a list of **discrete, known values** (e.g., numbers or strings). It often results in cleaner and more readable code than a long chain of if-else if statements.

* **Good for switch:** Checking the day of the week (1, 2, 3...), user roles ("admin", "editor", "guest"), or menu options.
* **Better for if-else:** Checking conditions with ranges (e.g., age > 18), multiple different variables, or complex boolean expressions.

**Lab Assignment**

**Task 1:** Write a JavaScript program to check if a number is positive, negative, or zero using an if-else statement.

**Solution:** checkNumber.js

JavaScript

let number = -5;

if (number > 0) {

console.log("The number is positive.");

} else if (number < 0) {

console.log("The number is negative.");

} else {

console.log("The number is zero.");

}

// Output for number = -5: "The number is negative."

**Task 2:** Create a JavaScript program using a switch statement to display the day of the week based on the user input (e.g., 1 for Monday, 2 for Tuesday, etc.).

**Solution:** dayOfWeek.js

JavaScript

let dayNumber = 3;

let dayName;

switch (dayNumber) {

case 1:

dayName = "Monday";

break;

case 2:

dayName = "Tuesday";

break;

case 3:

dayName = "Wednesday";

break;

case 4:

dayName = "Thursday";

break;

case 5:

dayName = "Friday";

break;

case 6:

dayName = "Saturday";

break;

case 7:

dayName = "Sunday";

break;

default:

dayName = "Invalid day number. Please enter a number between 1 and 7.";

}

console.log(dayName);

// Output for dayNumber = 3: "Wednesday"

**Loops (For, While, Do-While)**

**Theory Assignment**

**Question 1: Explain the different types of loops in JavaScript (for, while, do-while). Provide a basic example of each.**

Loops are used to execute a block of code repeatedly as long as a certain condition is met.

1. **for loop:** Repeats a block of code a **known number of times**. It consists of three parts: initialization, condition, and increment/decrement.
   * **Syntax:** for (initialization; condition; finalExpression) { ... }
   * **Example:**

JavaScript

for (let i = 0; i < 5; i++) {

console.log("The number is " + i);

}

1. **while loop:** Repeats a block of code as long as a specified condition is true. The condition is checked **before** the loop body is executed.
   * **Syntax:** while (condition) { ... }
   * **Example:**

JavaScript

let i = 0;

while (i < 5) {

console.log("The number is " + i);

i++;

}

1. **do-while loop:** Similar to a while loop, but the loop body is executed at least **once** before the condition is checked.
   * **Syntax:** do { ... } while (condition);
   * **Example:**

JavaScript

let i = 0;

do {

console.log("The number is " + i);

i++;

} while (i < 5);

**Question 2: What is the difference between a while loop and a do-while loop?**

The key difference is **when the condition is checked**.

* **while loop:** The condition is checked at the **beginning** of each iteration. If the condition is initially false, the loop's body will **never** execute.
* **do-while loop:** The condition is checked at the **end** of each iteration. This guarantees that the loop's body will be executed **at least once**, regardless of whether the condition is true or false.

**Example:**

JavaScript

// while loop

let i = 5;

while (i < 5) {

// This code will never run

console.log("While loop executed");

}

// do-while loop

let j = 5;

do {

// This code will run once

console.log("Do-while loop executed");

} while (j < 5);

// Output: "Do-while loop executed"

**Lab Assignment**

**Task 1:** Write a JavaScript program using a for loop to print numbers from 1 to 10.

**Solution:** printNumbers.js

JavaScript

for (let i = 1; i <= 10; i++) {

console.log(i);

}

**Task 2:** Create a JavaScript program that uses a while loop to sum all even numbers between 1 and 20.

**Solution:** sumEven.js

JavaScript

let sum = 0;

let i = 1;

while (i <= 20) {

if (i % 2 === 0) { // Check if the number is even

sum = sum + i;

}

i++;

}

console.log("The sum of all even numbers between 1 and 20 is:", sum); // Output: 110

**Task 3:** Write a do-while loop that continues to ask the user for input until they enter a number greater than 10. *(Note: This requires a browser environment for the prompt function)*

**Solution:** userInput.js

JavaScript

// This code should be run in a browser console or linked HTML file.

let number;

do {

// The prompt returns a string, so we convert it to a number

number = parseInt(prompt("Please enter a number greater than 10:"));

} while (isNaN(number) || number <= 10);

alert("Thank you! You entered " + number + ", which is greater than 10.");

**Functions**

**Theory Assignment**

**Question 1: What are functions in JavaScript? Explain the syntax for declaring and calling a function.**

A **function** is a reusable block of code designed to perform a particular task. Functions help organize code into logical, manageable pieces, reduce repetition, and improve readability.

**Declaring a Function:** You define a function using the function keyword, followed by a name, parentheses (), and a block of code {}.

JavaScript

function greet() {

console.log("Hello, world!");

}

**Calling a Function:** To execute the code inside a function, you "call" or "invoke" it by writing the function's name followed by parentheses ().

JavaScript

greet(); // This will execute the code inside the greet function

// Output: "Hello, world!"

**Question 2: What is the difference between a function declaration and a function expression?**

| Feature | Function Declaration | Function Expression |
| --- | --- | --- |
| **Syntax** | Starts with the function keyword. | A function is created and assigned to a variable. The function can be named or anonymous. |
| **Hoisting** | **Hoisted**. The entire function definition is moved to the top of its scope, so it can be called before it is defined in the code. | **Not hoisted**. The variable declaration (var, let, const) is hoisted, but the function assignment is not. It cannot be called before the line where it is defined. |
| **Name** | Must have a name. | The name is optional (anonymous function). |
| **Example** | function doSomething() { ... } | const doSomething = function() { ... }; |

Export to Sheets

**Hoisting Example:**

JavaScript

// Function Declaration - This works

sayHi();

function sayHi() {

console.log("Hi!");

}

// Function Expression - This will cause a TypeError

// sayHello(); // Uncommenting this line will throw an error

const sayHello = function() {

console.log("Hello!");

};

sayHello(); // This works

**Question 3: Discuss the concept of parameters and return values in functions.**

* **Parameters:** These are the names listed in the function's definition. They act as placeholders for the values that will be passed into the function when it is called. A function can have zero or more parameters.
* **Arguments:** These are the actual values passed to the function when it is invoked.
* **Return Values:** A function can pass data back to the code that called it using the return statement. When a return statement is executed, the function stops executing and the specified value is "returned". If a function does not have a return statement, it implicitly returns undefined.

**Example:**

JavaScript

// 'num1' and 'num2' are parameters

function add(num1, num2) {

let sum = num1 + num2;

return sum; // Returns the calculated sum

}

// 5 and 3 are arguments

let result = add(5, 3); // The return value of add() is stored in 'result'

console.log(result); // Output: 8

**Lab Assignment**

**Task 1:** Write a function greetUser that accepts a user’s name as a parameter and displays a greeting message (e.g., "Hello, John!").

**Solution:** greet.js

JavaScript

function greetUser(name) {

console.log("Hello, " + name + "!");

}

// Call the function with different names

greetUser("John"); // Output: Hello, John!

greetUser("Alice"); // Output: Hello, Alice!

**Task 2:** Create a JavaScript function calculateSum that takes two numbers as parameters, adds them, and returns the result.

**Solution:** sum.js

JavaScript

function calculateSum(num1, num2) {

return num1 + num2;

}

// Call the function and store the returned value

let result1 = calculateSum(10, 5);

console.log("The sum is:", result1); // Output: The sum is: 15

let result2 = calculateSum(-4, 20);

console.log("The sum is:", result2); // Output: The sum is: 16

**Arrays**

**Theory Assignment**

**Question 1: What is an array in JavaScript? How do you declare and initialize an array?**

An **array** in JavaScript is a single, special type of object used to store an ordered collection of multiple values. These values, called elements, can be of any data type (numbers, strings, objects, etc.) and are accessed using a zero-based index.

**Declaration and Initialization:** You can create an array in a few ways:

1. **Array Literal (most common):**

JavaScript

// Declare an empty array

let emptyArray = [];

// Declare and initialize an array with elements

let fruits = ["Apple", "Banana", "Cherry"];

let mixedData = [1, "Hello", true, null];

1. **Array Constructor:**

JavaScript

// Declare an empty array

let anotherEmptyArray = new Array();

// Declare and initialize

let numbers = new Array(10, 20, 30);

**Question 2: Explain the methods push(), pop(), shift(), and unshift() used in arrays.**

These are common methods used to add or remove elements from the beginning or end of an array.

* **push()**: **Adds** one or more elements to the **end** of an array and returns the new length of the array.

JavaScript

let fruits = ["Apple", "Banana"];

fruits.push("Cherry"); // fruits is now ["Apple", "Banana", "Cherry"]

* **pop()**: **Removes** the **last** element from an array and returns that element.

JavaScript

let fruits = ["Apple", "Banana", "Cherry"];

let lastFruit = fruits.pop(); // lastFruit is "Cherry", fruits is now ["Apple", "Banana"]

* **unshift()**: **Adds** one or more elements to the **beginning** of an array and returns the new length of the array.

JavaScript

let fruits = ["Banana", "Cherry"];

fruits.unshift("Apple"); // fruits is now ["Apple", "Banana", "Cherry"]

* **shift()**: **Removes** the **first** element from an array and returns that element.

JavaScript

let fruits = ["Apple", "Banana", "Cherry"];

let firstFruit = fruits.shift(); // firstFruit is "Apple", fruits is now ["Banana", "Cherry"]

**Lab Assignment**

**Task 1:** Declare an array of fruits (["apple", "banana", "cherry"]). Use JavaScript to add a fruit to the end, remove the first fruit, and log the modified array.

**Solution:** arrayManipulation.js

JavaScript

let fruits = ["apple", "banana", "cherry"];

console.log("Original array:", fruits);

// Add a fruit to the end of the array

fruits.push("date");

console.log("After push('date'):", fruits);

// Remove the first fruit from the array

fruits.shift();

console.log("After shift():", fruits);

// Final modified array

console.log("Modified array:", fruits); // Output: Modified array: [ 'banana', 'cherry', 'date' ]

**Task 2:** Write a program to find the sum of all elements in an array of numbers.

**Solution:** sumArray.js

JavaScript

let numbers = [5, 10, 15, 20];

let sum = 0;

for (let i = 0; i < numbers.length; i++) {

sum += numbers[i];

}

console.log("The array is:", numbers);

console.log("The sum of all elements is:", sum); // Output: The sum of all elements is: 50

**Objects**

**Theory Assignment**

**Question 1: What is an object in JavaScript? How are objects different from arrays?**

An **object** in JavaScript is a complex data type that allows you to store a collection of **key-value pairs**. These pairs are called **properties**. The key is a string (or Symbol), and the value can be any data type, including other objects or functions (which are called methods when they are properties of an object).

**Differences between Objects and Arrays:**

| Feature | Objects | Arrays |
| --- | --- | --- |
| **Structure** | Unordered collection of key-value pairs. | Ordered collection of elements. |
| **Access** | Elements are accessed by **keys** (property names). | Elements are accessed by a numerical **index** (0, 1, 2...). |
| **Use Case** | Best for storing and structuring data with descriptive labels (e.g., a user object with name and email properties). | Best for storing an ordered list of items (e.g., a list of products or tasks). |
| **Order** | The order of properties is not guaranteed (though modern JS engines often maintain insertion order). | The order of elements is guaranteed and is a fundamental characteristic. |
| **Example** | let car = { brand: "Ford", model: "Mustang" }; | let colors = ["Red", "Green", "Blue"]; |

Export to Sheets

**Question 2: Explain how to access and update object properties using dot notation and bracket notation.**

You can access and modify the properties of an object using two different syntaxes:

1. **Dot Notation (.):**
   * This is the most common and straightforward way.
   * You use it when the property key is a valid JavaScript identifier (e.g., contains no spaces or special characters).
   * **Accessing:** object.propertyName
   * **Updating:** object.propertyName = newValue;
2. **Bracket Notation ([]):**
   * This syntax is more flexible.
   * You must use it when the property key is not a valid identifier (e.g., first-name) or when the key is stored in a variable.
   * **Accessing:** object['propertyName']
   * **Updating:** object['propertyName'] = newValue;

**Example:**

JavaScript

let person = {

name: "Alice",

"job-title": "Developer"

};

// --- Dot Notation ---

console.log(person.name); // Accessing -> Output: Alice

person.name = "Bob"; // Updating

console.log(person.name); // -> Output: Bob

// --- Bracket Notation ---

console.log(person["job-title"]); // Accessing with a key containing a hyphen -> Output: Developer

let propertyKey = "name";

console.log(person[propertyKey]); // Accessing with a variable -> Output: Bob

person["job-title"] = "Senior Developer"; // Updating

console.log(person["job-title"]); // -> Output: Senior Developer

**Lab Assignment**

**Task:** Create a JavaScript object car. Access and print its properties, update a property, and add a new property.

**Solution:** carObject.js

JavaScript

// Create the car object

let car = {

brand: "Toyota",

model: "Camry",

year: 2021

};

// Access and print the car's brand and model

console.log("Car Brand:", car.brand); // Using dot notation

console.log("Car Model:", car['model']); // Using bracket notation

// Update the year property

car.year = 2022;

console.log("Updated Year:", car.year);

// Add a new property 'color'

car.color = "blue";

console.log("Newly added color:", car.color);

// Log the final car object

console.log("Final Car Object:", car);

**Console Output:**

Car Brand: Toyota

Car Model: Camry

Updated Year: 2022

Newly added color: blue

Final Car Object: { brand: 'Toyota', model: 'Camry', year: 2022, color: 'blue' }

**JavaScript Events**

**Theory Assignment**

**Question 1: What are JavaScript events? Explain the role of event listeners.**

**JavaScript events** are actions or occurrences that happen in the browser, such as a user clicking a button, a web page finishing loading, a key being pressed on the keyboard, or the mouse being moved over an element. JavaScript can "listen" for these events and execute code in response, creating dynamic and interactive web pages.

An **event listener** is a function that waits for a specific event to occur on a specific HTML element. When the event happens (e.g., a 'click' event on a button), the event listener "fires" and executes the code block or function it is attached to. This is the core mechanism for making web pages respond to user input.

**Question 2: How does the addEventListener() method work in JavaScript? Provide an example.**

The addEventListener() method is the modern and recommended way to handle events in JavaScript. It attaches an event handler function (the "listener") to a specified HTML element.

**Syntax:** element.addEventListener(event, function, useCapture);

* element: The HTML element to attach the listener to (e.g., a button).
* event: The name of the event as a string (e.g., 'click', 'mouseover', 'keydown').
* function: The function to be called when the event occurs.
* useCapture: (Optional) A boolean value specifying the event propagation phase. It's rarely used and defaults to false.

One of its key advantages is that you can add **multiple event listeners** for the same event to a single element without overwriting previous ones.

**Example:** Let's say you have an HTML button: <button id="myBtn">Click Me</button>

The JavaScript would be:

JavaScript

// 1. Select the element from the DOM

const myButton = document.getElementById('myBtn');

// 2. Define the function to run on the event

function showAlert() {

alert('Button was clicked!');

}

// 3. Attach the event listener

myButton.addEventListener('click', showAlert);

Now, whenever the button with the ID myBtn is clicked, the showAlert function will be executed.

**Lab Assignment**

**Task:** Create a simple webpage with a button that, when clicked, displays an alert saying "Button clicked!" using JavaScript event listeners.

**Solution:** Create an HTML file named event.html with the following code.

event.html

HTML

<!DOCTYPE html>

<html lang="en">

<head>

<title>Event Listener Example</title>

</head>

<body>

<button id="clickMeButton">Click Me!</button>

<script>

// Select the button element

const myButton = document.getElementById('clickMeButton');

// Add a 'click' event listener to the button

myButton.addEventListener('click', function() {

// This function will run when the button is clicked

alert('Button clicked!');

});

</script>

</body>

</html>

**DOM Manipulation**

**Theory Assignment**

**Question 1: What is the DOM (Document Object Model) in JavaScript? How does JavaScript interact with the DOM?**

The **DOM (Document Object Model)** is a programming interface for web documents. It represents the structure of an HTML or XML document as a logical, tree-like structure. Each part of the document—such as elements, attributes, and text—is organized into a hierarchy of "nodes." The top-level node is the document object itself.

**JavaScript interacts with the DOM** to dynamically access and manipulate the content, structure, and style of a web page. When a browser loads an HTML file, it creates the DOM representation of that page in memory. JavaScript can then:

* **Find and select** any HTML element on the page.
* **Change** the text content or HTML structure inside an element.
* **Modify** the attributes of an element (like src of an image or href of a link).
* **Alter** the CSS styles of an element (change color, size, position, etc.).
* **Create and add** new elements to the page.
* **Remove** existing elements.
* **React** to DOM events (like clicks) to trigger these changes.

**Question 2: Explain the methods getElementById(), getElementsByClassName(), and querySelector() used to select elements from the DOM.**

These are methods on the document object used to select HTML elements.

1. **getElementById()**:
   * Selects a **single** element that has a specific id attribute.
   * Since IDs must be unique within a page, this method always returns at most one element object, or null if no element with that ID is found.
   * It is very fast and efficient.
   * **Example:** const header = document.getElementById('main-header');
2. **getElementsByClassName()**:
   * Selects all elements that have a specific class name.
   * It returns a live **HTMLCollection** (an array-like object) of all matching elements.
   * You access individual elements by their index (e.g., elements[0]).
   * **Example:** const items = document.getElementsByClassName('list-item');
3. **querySelector()**:
   * A more modern and versatile method. It returns the **first** element within the document that matches a specified CSS selector.
   * If no matches are found, it returns null.
   * It can select by ID ('#myId'), class ('.myClass'), tag ('p'), or any other complex CSS selector ('div > p').
   * To select *all* matching elements, you would use querySelectorAll(), which returns a static NodeList.
   * **Example:** const firstItem = document.querySelector('.list-item');

**Lab Assignment**

**Task:** Create an HTML page with a paragraph. Use JavaScript to change its text and color.

**Solution:** Create an HTML file named dom.html with the following code.

dom.html

HTML

<!DOCTYPE html>

<html lang="en">

<head>

<title>DOM Manipulation</title>

</head>

<body>

<p id="myParagraph">Hello, World!</p>

<script>

// 1. Select the paragraph element using its ID

const paragraph = document.getElementById('myParagraph');

// 2. Change the text inside the paragraph

// The 'textContent' property is often preferred over 'innerHTML' for changing just text

paragraph.textContent = "JavaScript is fun!";

// 3. Change the color of the paragraph to blue

paragraph.style.color = "blue";

// You could also change other styles

paragraph.style.fontSize = "24px";

paragraph.style.fontWeight = "bold";

</script>

</body>

</html>

**JavaScript Timing Events (setTimeout, setInterval)**

**Theory Assignment**

**Question 1: Explain the setTimeout() and setInterval() functions in JavaScript. How are they used for timing events?**

setTimeout and setInterval are two key functions in JavaScript that allow you to execute code after a specified time delay or at regular intervals. They are fundamental to creating timed and asynchronous behavior.

* **setTimeout(function, delay)**:
  + Executes a function **once** after waiting for a specified number of milliseconds (the delay).
  + It's used for delaying an action. For example, showing a popup message 5 seconds after a page loads.
  + It returns a unique ID which can be used with clearTimeout(id) to cancel the execution before it happens.
* **setInterval(function, interval)**:
  + Executes a function **repeatedly**, at a specified time interval (in milliseconds).
  + The execution continues until it is stopped. It's used for tasks that need to run continuously, like updating a clock every second or checking for new notifications every minute.
  + It returns a unique ID which must be used with clearInterval(id) to stop the repeated execution.

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

This example will log a message to the console after a 2-second (2000 millisecond) delay.

JavaScript

console.log("Message will appear in 2 seconds...");

// setTimeout takes a function and a delay in milliseconds

setTimeout(function() {

// This code will run after 2000ms

console.log("Hello after 2 seconds!");

}, 2000); // 2000 milliseconds = 2 seconds

**Lab Assignment**

**Task 1:** Write a program that changes the background color of a webpage after 5 seconds using setTimeout().

**Solution:** Create an HTML file named timedColor.html with this code.

timedColor.html

HTML

<!DOCTYPE html>

<html lang="en">

<head>

<title>Timed Background Color Change</title>

</head>

<body>

<h1>Wait for it...</h1>

<p>The background color will change to lightblue in 5 seconds.</p>

<script>

setTimeout(function() {

// This function will execute after 5000 milliseconds (5 seconds)

document.body.style.backgroundColor = 'lightblue';

}, 5000);

</script>

</body>

</html>

**Task 2:** Create a digital clock that updates every second using setInterval().

**Solution:** Create an HTML file named clock.html with this code.

clock.html

HTML

<!DOCTYPE html>

<html lang="en">

<head>

<title>Digital Clock</title>

<style>

body { font-family: sans-serif; display: grid; place-items: center; height: 100vh; background: #222; color: #fff; }

#clock { font-size: 5rem; font-weight: bold; }

</style>

</head>

<body>

<div id="clock">Loading...</div>

<script>

const clockElement = document.getElementById('clock');

function updateClock() {

const now = new Date();

const hours = now.getHours().toString().padStart(2, '0');

const minutes = now.getMinutes().toString().padStart(2, '0');

const seconds = now.getSeconds().toString().padStart(2, '0');

clockElement.textContent = `${hours}:${minutes}:${seconds}`;

}

// Call updateClock once immediately to show the time right away

updateClock();

// Then, set an interval to call updateClock every 1000 milliseconds (1 second)

setInterval(updateClock, 1000);

</script>

</body>

</html>

**JavaScript Error Handling**

**Theory Assignment**

**Question 1: What is error handling in JavaScript? Explain the try, catch, and finally blocks with an example.**

**Error handling** is the process of anticipating and responding to errors that might occur while a program is running. Instead of letting an error crash the script, you can "catch" it and execute code to handle it gracefully, for example, by logging the error or showing a friendly message to the user.

JavaScript provides the **try...catch...finally** statement for this purpose.

* **try block:** This block contains the code that you suspect might throw an error. The JavaScript engine will attempt to execute this code.
* **catch block:** If an error occurs within the try block, the program's control immediately jumps to the catch block. The error object, containing details about the error, is passed as an argument to this block. This block will only execute if an error is thrown.
* **finally block:** This block contains code that will be executed **regardless** of whether an error occurred or not. It runs after the try block (if successful) or after the catch block (if an error was caught). It's often used for cleanup tasks, like closing a file or a network connection.

**Example:**

JavaScript

function riskyOperation() {

try {

console.log("Trying to execute code...");

// Let's create an intentional error

let user = null;

console.log(user.name); // This will throw a TypeError

console.log("This line will not be reached."); // This won't run

} catch (error) {

// This block runs because an error occurred

console.error("An error was caught!");

console.error("Error message:", error.message); // e.g., "Cannot read properties of null (reading 'name')"

} finally {

// This block always runs

console.log("The 'finally' block has executed.");

}

}

riskyOperation();

**Question 2: Why is error handling important in JavaScript applications?**

Error handling is crucial for building robust and user-friendly applications for several reasons:

1. **Prevents Application Crashes:** A single unhandled error can halt the execution of your entire script. Proper error handling ensures that the rest of your application can continue to function even if one part fails.
2. **Improves User Experience:** Instead of showing users a broken page or a cryptic error message from the browser console, you can catch the error and display a helpful, user-friendly message (e.g., "Sorry, we couldn't load the data. Please try again later.").
3. **Aids in Debugging:** The catch block gives you access to the error object, which contains the error message and a stack trace. You can log this information to a server or the console, making it much easier to diagnose and fix bugs.
4. **Enhances Security:** It can help prevent the exposure of sensitive system information that might be leaked through raw error messages.
5. **Graceful Degradation:** It allows your application to fail gracefully. For example, if a non-essential feature (like loading a social media feed) fails, the main functionality of the site can still work.

**Lab Assignment**

**Task:** Write a JavaScript program that attempts to divide a number by zero. Use try-catch to handle the error and display an appropriate error message.

*(Note: In JavaScript, dividing by zero results in Infinity, not a runtime error. To demonstrate try-catch effectively, we will manually throw a custom error when this case is detected.)*

**Solution:** errorHandling.js

JavaScript

function divideNumbers(numerator, denominator) {

try {

// Check if the denominator is zero

if (denominator === 0) {

// Manually throw an error to be caught by the catch block

throw new Error("Division by zero is not allowed.");

}

// This part will only run if no error is thrown

let result = numerator / denominator;

console.log("Result of division:", result);

} catch (error) {

// This block executes because we threw an error

console.error("An error occurred!");

console.error("Details:", error.message);

} finally {

console.log("Division attempt finished.");

}

}

// Case 1: Successful division

console.log("--- Attempting valid division ---");

divideNumbers(10, 2);

console.log("\n--- Attempting division by zero ---");

// Case 2: Division by zero

divideNumbers(10, 0);

**Console Output:**

--- Attempting valid division ---

Result of division: 5

Division attempt finished.

--- Attempting division by zero ---

An error occurred!

Details: Division by zero is not allowed.

Division attempt finished.