**THEORY ASSIGNMENT QUESTIONS**

* **JavaScript Introduction**

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

* JavaScript is scripting language that allows web browsers to respond to user Interactions and change the content of a web page.
* **Role of JavaScript: -**
* It allows elements on a webpage to react when you click, hover, or type

e.g., dropdown menus, image sliders

* JavaScript makes buttons, menus, and other elements clickable and responsive (e.g., forms, animations).

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

* JavaScript is Mainly used for web development, making web pages interactive.
* while python is general-purpose language used for data analysis, AI, automation, and backend development.
* Then java language is used for building large-scale applications, especially in enterprise, Android apps, and backend systems.
* JavaScript is best for the web.
* Python is best for data and simplicity.
* Java is best for large, robust applications.

1. **Discuss the use of <script> tag in HTML. How can you link an External JavaScript file to an HTML document?**

* The <script> tag is used to embed JavaScript code directly into an HTML document or link to an external JavaScript file.
* The <script> tag can reference an external JavaScript file, keeping your code organized and separate from HTML.
* <script src="script.js"></script>.
* Head section of <script> tag:-
* JavaScript will be loaded and executed before the content is fully loaded.

**Example of head section of <script>:-**

<head>

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

</head>

**Body section of <script> tag: -**

* The script will load after the page content is fully loaded, improving page performance.

**Example body <script> tag: -**

<body>

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

</body>

* **Variables and Datatypes**

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

* In JavaScript, variables are used to store data values that can be referenced and manipulated in a program. Variables act as containers for data.
* var keyword - redeclare & reassign it can be possible
* let keyword - redeclare not possible but reassign possible
* const - constant - fixed - Can’t redeclare & reassign

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

* **There are two types:**

1. Primitive Data Types
2. Non-primitive Data Types

**1. Primitive Data Types:**

**a) Number**

* Used for all types of numbers (integers, decimals, etc.).
* Example:

let age = 25; // Integer let price = 99.99;

// Decimal console.log (age, price);

// Output: 25, 99.99

**b) String**

* Used for text or characters, enclosed in quotes ("", '', or ````).

**Example:**

let name = "John";

let message = 'Hello, World!';

let template = `My age is ${age}`;

console.log (name, message, template); // Output: John Hello, World! My age is 25

**c) Boolean**

* Represents true or false values.
* **Example:**

let isLoggedIn = true;

let hasAccess = false;

console.log (isLoggedIn, hasAccess);

// Output: true, false

**d) Undefined**

* A variable that has been declared but not assigned a value yet.
* Example

let myVar;

console.log(myVar); // Output: undefined

**e) Null**

* Represents an intentional absence of value (similar to "nothing").
* Example

let emptyValue = null;

console.log(emptyValue);

// Output: null

**f) Symbol**

* Used for unique identifiers.
* Example: let id = Symbol('unique');
* console.log(id); // Output: Symbol(unique)

**g) BigInt**

* Used for numbers larger than Number.MAX\_SAFE\_INTEGER.
* Example: let bigNumber = 123456789012345678901234567890n;
  + - * console.log(bigNumber);
      * // Output: 123456789012345678901234567890n

2. **Non-Primitive Data Type**

**h) Object**

* Used to store collections of data or more complex entities.
* **Examples:**

Object:

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

console.log(person.name);

// Output: John

**i) Array:**

* An Array is a special kind of object used to store an ordered collection of values, which can be of any data type.

let colors = ["red", "green", "blue"];

console.log (colors [1]); // Output: green

**j) Function:** Function is a block of code designed to perform a specific task.

function greet () {return "Hello!"; }

console.log (greet ());

// Output: Hello!

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

* Undefined means a variable has been declared but has not yet been assigned a value, whereas null is an assignment value, meaning that a variable has been declared and given the value of null.
* **JavaScript Operators**

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

**• Assignment operators**

**• Comparison operators**

**• Logical operators**

* + An operator can be defined as the symbol that helps us to perform some specific operations.

1. **Arithmetic operators: -**

* An arithmetic operator is a symbol used to perform mathematical operations on operands such as Addition, Subtraction, Multiplication, Division, and modulo.

| **Operator** | **Description** | **Example** | **Output** |
| --- | --- | --- | --- |
| + | Addition | 5 + 3 | 8 |
| - | Subtraction | 5 – 3 | 2 |
| \* | Multiplication | 5 \* 3 | 15 |
| / | Division | 6 / 3 | 2 |
| % | Modulus (remainder) | 7 % 3 | 1 |
| ++ | Increment (add 1) | let x = 5; x++; | 6 |
| -- | Decrement (subtract 1) | let x = 5; x--; | 4 |

1. **Assignment operators: -**

* These are used to assign or update values.

|  |  |  |  |
| --- | --- | --- | --- |
| Operator | Description | Example | Output |
| = | Assign a value | let x = 10; | x = 10 |
| += | Add and assign | x += 5; | x = 15 |
| -= | Subtract and assign | x -= 3; | x = 12 |
| \*= | Multiply and assign | x \*= 2; | x = 24 |
| /= | Divide and assign | x /= 3; | x = 8 |
| %= | Modulus and assign | x %= 5; | x = 3 |

**3. Comparison Operators: -**

* The relational operator is are used for the comparison of the two operands.
* All these operators are binary operators that return true or false values as the result of comparison.

| **Operator** | **Description** | **Example** | **Output** |
| --- | --- | --- | --- |
| == | Equal to (value) | 5 == "5" | true |
| === | Equal to (value & type) | 5 === "5" | false |
| != | Not equal | 5! = "4" | true |
| !== | Not equal (value & type) | 5! == "5" | true |
| > | Greater than | 7 > 5 | true |
| < | Less than | 7 < 5 | false |
| >= | Greater than or equal | 7 >= 7 | true |
| <= | Less than or equal to | 7 <= 7 | true |

|  |
| --- |
|  |

1. **Logical Operators: -**

* Logical operators in JavaScript are used to combine conditions or check if something is **true** or **false**. They help make decisions in programs.

**Explanation:**

1. **&& (AND)**:
   * Returns true only if **both conditions are true**.

let x = 10, y = 20;

console.log(x > 5 && y > 15); // Output: true (both are true)

console.log(x > 15 && y > 15); // Output: false (one is false)

1. **|| (OR)**:
   * Returns true if **at least one condition is true**.

Example:

let x = 10, y = 5;

console.log(x > 15 || y > 3); // Output: true (one is true)

console.log(x > 15 || y > 10); // Output: false (both are false)

1. **! (NOT)**:
   * Reverses the result of a condition.

Example:

let isRaining = true;

console.log(!isRaining); // Output: false (reverses true to false)

1. **What is the difference between == and === in JavaScript?**
   * **== (Loose Equality)**
     + Compares **values only**, ignoring the data type.
     + Converts (coerces) one value to match the other type before comparing.
   * **=== (Strict Equality)**
     + Compares **both value and data type**.
     + No conversion is done — if types are different, it returns false.
   * The main difference between the two operators is how they compare values. The == operator compares the values of two variables after performing type conversion if necessary. On the other hand, the === operator compares the values of two variables without performing type conversion.

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

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

* Control flow refers to the **order** in which a program's instructions are executed. By default, JavaScript runs code from **top to bottom**, but you can change this flow using conditional statements, loops, and functions.
* **What is an if-else Statement?**
  + An **if-else statement** allows you to execute different blocks of code based on a condition:
  + **if**: Runs a block of code if the condition is **true**.
  + **else**: Runs another block of code if the condition is **false**.

if (condition) {

// Code to run if condition is true

} else {

// Code to run if condition is false

}

* + **Example:**

let age = 18;

if (age >= 18) {

console.log("You are eligible to vote."); // Runs if age is 18 or more

} else {

console.log("You are not eligible to vote."); // Runs if age is less than 18

}

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

* A **switch statement** is used to perform different actions based on the value of a variable or expression. It is an alternative to using multiple if-else if statements when you have many possible conditions to check.
* **Use switch**:
  + When you are comparing a single variable or expression with multiple **exact** values.
  + Example:

switch (fruit) {

case "apple":

console.log("Apple is $2.");

break;

case "banana":

console.log("Banana is $1.");

break;

default:

console.log("Fruit not available.");

}

* **Key Points**
* switch is easier to read than multiple if-else if statements for simple value-based comparisons.
* Always include break to prevent running into the next case.
* Use default to handle unmatched cases.
* **Loops (For, While, Do-While)**
  1. **Explain the different types of loops in JavaScript (for, while, do-while). Provide a basic example of each?**
* Loops in JavaScript allow you to **run a block of code multiple times** until a condition is met.
* **For loop:** Used when the number of iterations is known.
* **Example:**

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

{console.log ("Count:", i);

}

**// Output: Count: 0, Count: 1, Count: 2, Count: 3, Count: 4**

* **While loop: -** The JavaScript while loop is a control flow statement that runs a block of code for as long as a specified condition is true.
* **Example: -**

let i = 0;

while (i < 5) {

console.log ("Count:", i);

i++;

}

**// Output: Count: 0, Count: 1, Count: 2, Count: 3, Count: 4**

* **Do- while loop: -** Similar to a while, but it **always runs at least once**, even if the condition is false.
* The code block runs first, then the condition is checked
* **Example: -** let i = 0;

do {

console.log ("Count:", i);

i++;

} while (i < 5);

**// Output: Count: 0, Count: 1, Count: 2, Count: 3, Count: 4**

* 1. **What is the difference between a while loop and a do-while loop?**
* While loop checks the condition before the execution of the statement(s) whereas the do-while loop ensures that the statement(s) is executed at least once before evaluating the condition.
* Do-while Loop uses a semicolon in the syntax, whereas While Loop does not use a semicolon
* **Functions**

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

* A function in JavaScript is a reusable block of code that performs a specific task. You define it once, and then you can run (or “call”) it whenever you need that task done in your program.
* **Syntax for Declaring a Function**
  + There are several ways to declare functions in JavaScript:

**1. Function Declaration**

function functionName(parameters) {

// Code to be executed

return result; // (optional)

}

**Example:**

function greet(name) {

return "Hello, " + name + "!";

}

**2. Function Expression**

const functionName = function(parameters) {

// Code to be executed

return result; // (optional)

};

**Example:**

const multiply = function(a, b) {

return a \* b;

};

**3. Arrow Function (ES6)**

const functionName = (parameters) => {

// Code to be executed

return result; // (optional)

};

**Example:**

const square = (x) => x \* x;

Syntax for Calling a Function

To call a function, use its name followed by parentheses. If the function has parameters, pass values inside the parentheses.

functionName(arguments);

Examples:

Using the greet function:

console.log(greet("Nisarg")); // Output: Hello, Nisarg!

Using the multiply function:

console.log(multiply(4, 5)); // Output: 20

Using the square function:

console.log(square(3)); // Output: 9

Key Points

Functions can have default parameter values:

function sayHi(name = "Guest") {

return "Hi, " + name;

}

console.log(sayHi());

**// Output: Hi, Guest**

* Functions can return a value or undefined if no return statement is provided.
* Use functions to reduce redundancy and improve code readability

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

| **Function Declaration** | **Function Expression** |
| --- | --- |
| A function declaration must have a function name. | A function expression is similar to a function declaration without the function name. |
| Function declaration does not require a variable assignment. | Function expressions can be stored in a variable assignment. |
| These are executed before any other code. | Function expressions load and execute only when the program interpreter reaches the line of code. |
| The function in function declaration can be accessed before and after the function definition. | The function in function expression can be accessed only after the function definition. |
| Function declarations are hoisted | Function expressions are not hoisted |
| **Syntax:** function geeksforGeeks(paramA, paramB) { // Set of statements } | **Syntax:** var geeksforGeeks= function(paramA, paramB) { // Set of statements } |

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

**Parameters:**

* **Definition:** Parameters are variables listed in a function's definition that act as placeholders for values that will be passed into the function when it's called.
* **Purpose:** They allow functions to work with different data each time they are invoked, making them more flexible and reusable.

**Return Values:**

* **Definition:** A return value is the value that a function "returns" to the caller after it has finished executing.
* **Purpose:** They enable functions to produce output or results that can be used elsewhere in your code.
* **Arrays**
  1. **What is an array in JavaScript? How do you declare and initialize an array?**
* An **array** is a special variable in JavaScript that can hold multiple values **in a single variable**. These values can be of any type, like numbers, strings, or even other arrays.
* Arrays are useful when you want to store a **list of items** and perform operations like looping through them, adding, or removing items.

// Declare and initialize an array

let numbers = new Array(1, 2, 3, 4);

console.log(numbers); // Output: [1, 2, 3, 4]

let dynamicArray = [];

dynamicArray.push(10); // Add elements using push

dynamicArray.push(20);

console.log(dynamicArray);

**// Output: [10, 20]**

* 1. **Explain the methods push(), pop(), shift(), and unshift() used in arrays.**
* These methods are used to add or remove elements from arrays.

1. **Push**() - Adds one or more elements to the **end** of an array.

**Syntax** :- array.push(Elements1, elements2,…);

**Example** :- let fruits = ["Apple", "Banana"];

fruits.push("Cherry"); // Adds "Cherry" at the end

console.log(fruits);

**// Output: ["Apple", "Banana", "Cherry"]**

1. **Pop()** - Removes the **last** element from an array.

**Syntax** :- array.pop();

**Example** :- let fruits = ["Apple", "Banana", "Cherry"];

let lastFruit = fruits.pop(); // Removes "Cherry"

console.log(fruits);

**// Output: ["Apple", "Banana"]**

console.log(lastFruit);

**// Output: "Cherry"**

**3. shift()** - Removes the **first** element from an array.

**Syntax** :- array.shift();

**Example** :- let fruits = ["Apple", "Banana", "Cherry"];

let firstFruit = fruits.shift();

console.log(fruits);

**// Output: ["Banana", "Cherry"]**

console.log(firstFruit);

**// Output: "Apple"**

**4. unshift()** - Adds one or more elements to the **beginning** of an array.

**Syntax** :- array.unshift();

**Example** :- let fruits = ["Banana", "Cherry"];

fruits.unshift("Apple");

console.log(fruits);

**// Output: ["Apple", "Banana", "Cherry"]**

* **Objects**
  1. **What is an object in JavaScript? How are objects different from arrays?**
* An **object** in JavaScript is a collection of **key-value pairs**, where the keys (called **properties**) are strings (or symbols), and the values can be any data type, including numbers, strings, arrays, functions, or even other objects.

**Syntax** :- Creating Object (Using Object Literal Notation)

Let objectName ={

Key1: value1,

Key2: value2,

//….

};

**Example** :-

Let person ={

Name: ”Siddhraj”,

age: 23,

isStudent: true

};

Console.log(person);

**// Output: {name: “Siddhraj”, age: 23,**

isStudent: true}

| **Feature** | **Objects** | **Arrays** |
| --- | --- | --- |
| **Structure** | Collection of key-value pairs. | Ordered list of elements. |
| **Key Type** | Keys are usually strings or symbols. | Keys are numeric indices starting from 0. |
| **Access** | Use object.key or object["key"]. | Use array[index]. |
| **Order** | Keys are not ordered. | Elements are ordered sequentially. |
| **Length Property** | No length property. | Has a length property to get the size. |
| **Purpose** | Used for structured data with named properties. | Used for ordered collections of items. |
| **Iteration** | Use for...in, Object.keys(), or Object.entries(). | Use for, for...of, or forEach(). |
| **Mutability** | Keys and values can be added or removed dynamically. | Elements can be added or removed dynamically. |
| **Best Use Case** | Representing entities like a person, car, etc. | Storing lists like names, colors, numbers, etc. |

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

1. **Dot Notation –**

**Syntax** – objectName.propertyName

**Example** –

let person = {

name: "Kirtan",

age: 23

};

console.log(person.name);

**// Accessing: Outputs "Kirtan"**

person.age = 24;

console.log(person.age);

**// Output: 24**

1. **Bracket Notation –**

**Syntax** – objectName[“propertyName”]

**Example** –

let person = {

name: "Kirtan",

age: 23

};

console.log(person [“first name”]); // Accessing: Outputs "Kirtan"

person[“age”] = 24;

console.log(person[‘age”]);

**// Output: 24**

let key =”age”;

console.log(person[key]);

**// Output: 26**

* **JavaScript Events**

**Events**

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

* In JavaScript, events are actions or occurrences that happen in the browser, like clicking a button, moving the mouse, or loading a page. They allow you to create interactive web pages that respond to user actions.

**Role of Event Listeners –**

**Key Roles of Event Listeners:**

1. **Detect User Actions**:
   * Event listeners monitor actions like clicks, mouse movements, key presses, or form submissions.
   * Example: Detecting when a button is clicked.
2. **Execute Associated Code**:
   * When the specified event occurs, the event listener triggers a function or a block of code (often called a callback function).
   * Example: Displaying a message when a button is clicked.
3. **Decouple Logic**:
   * Event listeners separate the logic of user interaction from the main application logic, making the code modular and maintainable.
4. **Enhance Interactivity**:
   * They allow developers to make dynamic, responsive applications that react to user actions in real-time.
5. **How does the addEventListener() method work in JavaScript? Provide an example.**

* The addEventListener() method in JavaScript is used to attach event listeners to HTML elements.
* This method allows you to listen for specific events (e.g., click, keydown, mouseover) and execute a callback function when the event occurs.

// Get the button element

const button = document.getElementById("myButton");

// Add an event listener for the "click" event

button.addEventListener("click", function() {

alert("Button clicked!");

});

* **DOM Manipulation**
  + 1. **What is the DOM (Document Object Model) in JavaScript? How does JavaScript interact with the DOM?**
* The Document Object Model (DOM) is a programming interface for HTML and XML documents.
* This allows programming languages like JavaScript to interact with, manipulate, and modify the structure, style, and content of a webpage dynamically.
* **Key Features of the DOM**

1. **Tree Structure**:
   * The DOM represents the document as a hierarchical tree.
   * Each HTML element (e.g., <div>, <p>) is a **node** in the tree.
2. **Dynamic Interaction**:
   * JavaScript can use the DOM to dynamically add, modify, or remove elements, attributes, and styles on a webpage.
3. **Standardized Interface**:
   * The DOM is standardized by the W3C, ensuring consistent behavior across browsers.

* **JavaScript provides methods and properties to interact with the DOM. Here's how:**

**1. Selecting Elements**

* JavaScript can find and reference elements in the DOM using selectors:
  + getElementById() – Selects an element by its id.
  + getElementsByClassName() – Selects elements by their class name.
  + getElementsByTagName() – Selects elements by their tag name.
  + querySelector() / querySelectorAll() – Selects elements using CSS-like selectors.

**2. Manipulating Elements**

* Modify content using properties like innerHTML, textContent, or value.
* Change attributes using methods like setAttribute() and getAttribute().
* Update styles using the style property.

**3. Adding/Removing Elements**

* Add elements: createElement(), appendChild(), insertBefore().
* Remove elements: removeChild(), replaceChild().

**4. Handling Events**

* Attach event listeners to DOM elements using addEventListener().
* React dynamically to user interactions (e.g., clicks, keyboard input).
  + 1. **Explain the methods getElementById(), getElementsByClassName(), and querySelector() used to select elements from the DOM.**
* JavaScript provides several methods to select elements from the DOM, enabling developers to manipulate and interact with webpage content. Here, we'll explain the commonly used methods: getElementById(), getElementsByClassName(), and querySelector().

1. **getElementById() - Purpose: Selects a single element by its unique id attribute.**

**Syntax**: const element= document.getElementById("id");

**Example -** <h1 id="mainTitle">Welcome!</h1>

<script>

const title = document.getElementById("mainTitle");

console.log(title.textContent); // Output: Welcome!

</script>

**2. getElementsByClassName()**

* **Purpose: Selects all elements with a specified class name.**
  1. **Syntax**: const elements = document.getElementsByClassName("className");
* **JavaScript Timing Events (setTimeout, setInterval)**
  1. **Explain the setTimeout() and setInterval() functions in JavaScript. How are they used for timing events?**
* setTimeout() and setInterval() in JavaScript
* Both setTimeout() and setInterval() are used to handle timing events in JavaScript. They enable developers to execute functions or code blocks at specific time intervals or after a delay.
  1. **setTimeout() –** 
     + Executes a function once after a specified delay (in milliseconds).
     + **Syntax-** setTimeout(function, delay, arg1, arg2, ...);
     + function: The function to execute after the delay.
     + delay: The time in milliseconds to wait before execution.
     + arg1, arg2, ...: Optional arguments to pass to the function.
       - * Example –

const timeoutId = setTimeout(() => {

console.log("This will never be logged!");

}, 5000);

// Cancel the timeout before it executes

clearTimeout(timeoutId);

* + **How It Works**
* The code continues executing while the timer runs in the background.
* After 3000 milliseconds (3 seconds), the callback function is executed.

**setInterval() –**

* + - Executes a function repeatedly at specified intervals (in milliseconds), until it is cleared.

**Syntax** - setInterval(function, interval, arg1, arg2, ...);

* + - function: The function to execute repeatedly.
    - interval: The time in milliseconds between each execution.
    - arg1, arg2, ...: Optional arguments to pass to the function.

Example -

const intervalId = setInterval(() => {

console.log("This repeats every second.");

}, 1000);

// Stop the interval after 5 seconds

setTimeout(() => {

clearInterval(intervalId);

console.log("Interval stopped!");

}, 5000);

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

// Function to display a message

function showMessage() {

console.log("This message appears after a 2-second delay!");

}

// Use setTimeout to call the function after 2000 milliseconds (2 seconds)

setTimeout(showMessage, 2000);

* + **Explanation**

1. **Function Definition:**
   * The showMessage function contains the code to execute after the delay.
2. **setTimeout Call:**
   * The showMessage function is passed as the first argument to setTimeout**.**
   * The delay is set to 2000 milliseconds (2 seconds).

**Output**

**After running the code, the message "This message appears after a 2-second delay!" is logged to the console after 2 seconds.**

* **JavaScript Error Handling**
  1. **What is error handling in JavaScript? Explain the try, catch, and finally blocks with an example.**
     + Error handling in JavaScript is the process of catching and managing errors that occur during the execution of code.
     + This ensures the program doesn't crash unexpectedly and provides meaningful feedback to users or developers.
     + **Errors in JavaScript can occur due to:**
* **Syntax mistakes.**
* Runtime issues (e.g., invalid operations, accessing undefined variables).
* External factors (e.g., failed API requests).
  + - To manage such errors, JavaScript provides a structured mechanism using try, catch, and finally blocks**.**
    - **Components of Error Handling**

**1. try Block**

* Contains the code that may throw an error.
* If an error occurs in this block, the program skips the remaining code in try and jumps to the catch block.

**2. catch Block**

* Executes if an error is thrown in the try block.
* Provides a way to handle the error gracefully.

**3. finally Block (Optional)**

* Executes after the try and catch blocks, regardless of whether an error occurred.
* Commonly used for cleanup tasks (e.g., closing files, releasing resources).

**Example –**

try {

// Code that may throw an error

let num = 10;

let result = num / 0; // This won't throw an error (Infinity)

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

let undefinedVar = undefined;

console.log(undefinedVar.toUpperCase()); // This will throw an error

} catch (error) {

// Handling the error

console.log("An error occurred:", error.message);

} finally {

// Executes regardless of what happens

console.log("Execution of try-catch block is complete.");

}

**Output -**

**Result: Infinity**

**An error occurred: undefinedVar.toUpperCase is not a function**

**Execution of try-catch block is complete.**

* 1. **Why is error handling important in JavaScript applications?**
  + Error handling is crucial in JavaScript applications because it ensures the application remains robust, user-friendly, and maintainable even when unexpected issues occur.

**1. Prevent Application Crashes**

* + - Without Error Handling: Errors can cause the entire application to crash, making it unusable for users.
    - With Error Handling: The application can gracefully recover from errors, allowing other parts of the app to function properly.

try {

const result = riskyFunction(); // Might throw an error

} catch (error) {

console.error("An error occurred, but the app is still running!");

}

**2. Improve User Experience**

* Without Error Handling: Users may encounter cryptic error messages or broken functionality, leading to frustration.
* With Error Handling: Users are provided with meaningful feedback or alternative workflows, improving satisfaction.

try {

fetch("invalid\_url");

} catch (error) {

alert("Failed to load data. Please try again later.");

}

**3. Debugging and Maintenance**

* Errors with detailed stack traces and messages help developers identify and fix bugs more efficiently.
* Proper error handling allows developers to log issues for analysis and future prevention.

try {

JSON.parse("{invalidJson}");

} catch (error) {

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

}

**4. Ensures Data Integrity**

* Prevents corrupt or partial data updates in cases of errors during operations like form submissions, database writes, or API calls.
* Helps roll back changes or notify the user to retry.

**5. Protects Sensitive Operations**

* In applications with critical operations (e.g., financial transactions), error handling prevents incorrect actions like double payments or incorrect computations.

try {

processTransaction();

} catch (error) {

alert("Transaction failed. Your account was not charged.");

}