Independent Skill Development Mission

Independent Skill Development Mission NXT
EMPOWERING FUTURES, SHAPING TOMORROW

ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION

INTRODUCTION TO JAVASCRIPT (WEEKS 10-12)

VARIABLES, DATA TYPES, AND OPERATORS

CHAPTER 1: INTRODUCTION TO VARIABLES, DATA TYPES, AND OPERATORS

1.1 Understanding Variables in Programming

A **variable** is a container used to store data in a program. It allows developers to **store, retrieve, and manipulate values dynamically**.

- **Why Are Variables Important?**
  - They **store user input, calculations, and database values**.
  - They help **reuse data efficiently without redundancy**.
  - They allow **dynamic operations**, making code more flexible.
- **Declaring Variables in JavaScript:**

```
var name = "John"; // Declares a variable with var (old method)

let age = 25; // Declares a variable using let (modern)

const PI = 3.1416; // Declares a constant (value cannot change)
```

1.2 Types of Variables in Different Languages

- **JavaScript:** Uses var, let, const.
- **Python:** Uses dynamic typing (name = "Alice", age = 30).
- **Java:** Requires explicit type declaration (String name = "Alice";).

CHAPTER 2: DATA TYPES IN PROGRAMMING

2.1 What Are Data Types?

Data types define **the kind of value** a variable can store.

- **Categories of Data Types:**

1️⃣ **Primitive Data Types** → Store simple values.

2️⃣ **Reference Data Types** → Store complex data structures.

2.2 Primitive Data Types

- **Common Primitive Data Types in JavaScript:**

Data Type	Description	Example
String	Text values	"Hello, world!"
Number	Integer & floating numbers	25, 3.14
Boolean	True/False values	true, false
Null	Empty value	null
Undefined	Variable with no value	let x;
Symbol	Unique identifiers (ES6+)	Symbol('id')

- **Example in JavaScript:**

```
let city = "New York"; // String

let population = 8419600; // Number
```

let isCapital = false; // Boolean

let unknownValue = null; // Null

let undefinedVar; // Undefined

2.3 Reference Data Types

Reference data types store **collections of values**.

- **Examples:**

1. **Arrays** → Ordered list of values.
2. **Objects** → Key-value pairs.

- **Example in JavaScript:**

let fruits = ["Apple", "Banana", "Mango"]; // Array

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

CHAPTER 3: OPERATORS IN PROGRAMMING

3.1 What Are Operators?

Operators are **symbols that perform operations on values**.

- **Types of Operators:**

Operator Type	Example	Description
Arithmetic	+, -, *, /	Perform mathematical calculations
Assignment	=, +=, -=	Assign and update values
Comparison	==, ===, !=, >	Compare values
Logical	&&, `	

Bitwise	&, `	, ^, <<`
Ternary	condition ? true : false	A shortcut for if-else

CHAPTER 4: UNDERSTANDING OPERATORS WITH EXAMPLES

4.1 Arithmetic Operators

- **Perform mathematical calculations**

```
let a = 10;

let b = 5;


console.log(a + b); // Addition: 15

console.log(a - b); // Subtraction: 5

console.log(a * b); // Multiplication: 50

console.log(a / b); // Division: 2

console.log(a % b); // Modulus: 0
```

4.2 Assignment Operators

- **Assign values to variables**

```
let x = 10;

x += 5; // Same as x = x + 5

console.log(x); // Output: 15
```

4.3 Comparison Operators

- **Compare values and return true/false**

```
let a = 5;

let b = "5";


console.log(a == b); // true (checks value only)

console.log(a === b); // false (checks value and type)
```

4.4 Logical Operators

- **Used for conditional checks**

```
let isMember = true;

let hasDiscount = isMember && age >= 18; // Both conditions must be true

console.log(hasDiscount);
```

Case Study: How E-Commerce Websites Use Variables, Data Types & Operators

Challenges Faced

- Managing **thousands of product listings**.
- Applying **discount calculations dynamically**.
- Ensuring **fast search and filtering**.

Solutions Implemented

- Used **arrays and objects** to store product details.
- Applied **arithmetic operators** for price calculations.
- Used **logical operators** for cart eligibility conditions.

- **Example:**

```
let product = { name: "Laptop", price: 1000, discount: 0.1 };

let finalPrice = product.price - (product.price * product.discount);

console.log(finalPrice); // Output: 900
```

📝 Exercise

✅ Declare a **string, number, boolean, and object** in JavaScript.
✅ Perform arithmetic calculations using **operators**.
✅ Write a **conditional statement** using comparison and logical operators.

Conclusion

- **Variables store data** for later use.
- **Data types define the kind of values variables can hold**.
- **Operators perform mathematical, comparison, and logical operations**.
- **Understanding these concepts is crucial for building dynamic applications**.

FUNCTIONS, LOOPS, AND CONDITIONAL STATEMENTS

CHAPTER 1: UNDERSTANDING FUNCTIONS IN PROGRAMMING

1.1 What Are Functions?

Functions are **reusable blocks of code** that perform a specific task. Instead of writing the same code multiple times, functions allow programmers to **define a task once and reuse it throughout the program**.

- **Why Use Functions?**
  - **Code Reusability:** Write once, use multiple times.
  - **Improved Readability:** Breaks code into logical sections.
  - **Easier Debugging:** Fixing errors is simpler when code is modular.
- **Example of a Basic Function in JavaScript:**

```
function greet() {
  console.log("Hello, World!");
}
greet(); // Output: Hello, World!
```

Here, greet() is a function that **prints a message** when called.

1.2 Function Parameters and Return Values

Functions can **accept parameters (inputs)** and **return values (outputs)**.

- **Example: Function with Parameters and Return Value**

```
function addNumbers(a, b) {
  return a + b;
}
console.log(addNumbers(5, 3)); // Output: 8
```

Here, a and b are **parameters**, and the function **returns their sum**.

1.3 Different Types of Functions

- **Named Functions:**

```
function multiply(x, y) {
  return x * y;
}
```

- **Anonymous Functions (Function Expressions):**

```
const subtract = function(a, b) {
  return a - b;
};
console.log(subtract(10, 5)); // Output: 5
```

- **Arrow Functions (ES6 Syntax in JavaScript):**

```
const divide = (x, y) => x / y;
console.log(divide(10, 2)); // Output: 5
```

Arrow functions provide **a concise way to write functions** in JavaScript.

CHAPTER 2: LOOPS IN PROGRAMMING

2.1 What Are Loops?

Loops allow code to be **executed repeatedly** until a condition is met.

- **Why Use Loops?**
  - Automates repetitive tasks.
  - Reduces the amount of code.
  - Improves efficiency in handling large datasets.

2.2 Types of Loops

2.2.1 For Loop

Used when **the number of iterations is known**.

- **Example:** Print numbers from 1 to 5:

```
for (let i = 1; i <= 5; i++) {
  console.log(i);
}
```

2.2.2 While Loop

Executes code **as long as a condition is true**.

- **Example:** Print numbers from 1 to 5 using a while loop:

```
let i = 1;
while (i <= 5) {
  console.log(i);
```

```
  i++;
}
```

2.2.3 Do-While Loop

Executes code **at least once,** even if the condition is false.

- **Example:**

```
let i = 1;
do {
  console.log(i);
  i++;
} while (i <= 5);
```

2.3 Loop Control Statements

- **Break Statement:** Exits a loop when a condition is met.

```
for (let i = 1; i <= 10; i++) {
  if (i === 5) break;
  console.log(i); // Stops at 4
}
```

- **Continue Statement:** Skips the current iteration and moves to the next one.

```
for (let i = 1; i <= 5; i++) {
  if (i === 3) continue;
  console.log(i); // Skips 3
```

```
}
```

CHAPTER 3: CONDITIONAL STATEMENTS

3.1 What Are Conditional Statements?

Conditional statements **control program flow** by executing different code blocks based on conditions.

- **Why Use Conditional Statements?**
  - They allow **decision-making** in programs.
  - Different **actions are executed based on user input or conditions**.

3.2 Types of Conditional Statements

3.2.1 If Statement

Executes code **only if a condition is true**.

```
let age = 18;

if (age >= 18) {
   console.log("You are eligible to vote.");
}
```

3.2.2 If-Else Statement

Executes one block if the condition is true, otherwise another block runs.

```
let age = 16;

if (age >= 18) {
```

```
  console.log("You can vote.");
} else {
  console.log("You cannot vote.");
}
```

3.2.3 Else-If Ladder

Used when there are **multiple conditions**.

```
let marks = 85;
if (marks >= 90) {
  console.log("Grade: A");
} else if (marks >= 75) {
  console.log("Grade: B");
} else {
  console.log("Grade: C");
}
```

3.2.4 Switch Statement

Used when multiple conditions depend on the **same variable**.

```
let day = "Monday";
switch (day) {
  case "Monday":
    console.log("Start of the workweek.");
    break;
  case "Friday":
```

```
    console.log("Weekend is coming!");
    break;
  default:
    console.log("It's a normal day.");
}
```

The switch statement is **faster** than multiple if-else checks for handling **multiple choices**.

Case Study: How Functions, Loops, and Conditionals Power E-Commerce Websites

Challenges Faced in E-Commerce Websites

- **Processing user inputs efficiently** (e.g., login authentication).
- **Displaying product listings dynamically**.
- **Handling checkout logic (e.g., discount conditions, cart loops).**

Solutions Implemented

- **Functions** to handle reusable tasks like calculating totals.
- **Loops** to display **multiple products** dynamically.
- **Conditional statements** for checkout logic (e.g., if a discount applies).

◆ **Example: Function to Apply Discount Based on Cart Value**

```
function applyDiscount(cartValue) {
  if (cartValue >= 100) {
    return cartValue * 0.9; // 10% discount
```

```
  } else {
    return cartValue;
  }
}
console.log(applyDiscount(150)); // Output: 135
```

This function **applies a discount** only if the cart value is above $100.

🔹 **Key Takeaways from E-Commerce Solutions:**

- **Functions improve modularity and reusability.**
- **Loops help iterate through product listings efficiently.**
- **Conditionals make dynamic decision-making possible.**

📝 **Exercise**

✅ Write a function that **calculates the square of a number**.
✅ Use a loop to **print numbers from 1 to 10**.
✅ Write a conditional statement to **check if a number is even or odd**.
✅ Create a **switch statement** to display different messages for different weekdays.

Conclusion

- **Functions make code reusable and modular.**
- **Loops automate repetitive tasks efficiently.**
- **Conditional statements control program flow and decision-making.**

- **Using all three concepts together makes programming more powerful and flexible.**

SELECTING & MODIFYING ELEMENTS

CHAPTER 1: INTRODUCTION TO SELECTING & MODIFYING ELEMENTS

1.1 What Does Selecting & Modifying Elements Mean?

Selecting and modifying elements refers to the process of using **CSS and JavaScript** to **target, style,** and **manipulate** elements in a webpage. This enables **dynamic interactivity** and **styling changes** based on user actions or predefined conditions.

- **Why is it Important?**
  - Allows **efficient styling** and structure control in CSS.
  - Enables **interactive elements** using JavaScript (e.g., changing text, hiding/showing elements).
  - Enhances **user experience (UX)** by allowing real-time updates to content.
- **Example:** Changing the **background color of a button** when clicked.

```
<button id="changeBtn">Click Me</button>


<script>


document.getElementById("changeBtn").addEventListener("click", function() {
    this.style.backgroundColor = "blue";
});
</script>
```

📌 **This changes the button's background color when clicked.**

CHAPTER 2: SELECTING ELEMENTS USING CSS

2.1 Selecting Elements with CSS Selectors

CSS allows selecting elements based on **tags, classes, IDs, attributes, and relationships**.

- **Common CSS Selectors:**
  - **Element Selector (h1, p, div)** → Targets all elements of a type.
  - **Class Selector (.class-name)** → Targets all elements with a class.
  - **ID Selector (#id-name)** → Targets a specific element with an ID.
  - **Attribute Selector ([type="text"])** → Targets elements based on attributes.
- **Example:**

```
h1 {
  color: red; /* Targets all h1 elements */
}

.box {
  border: 2px solid black; /* Targets elements with class "box" */
}
```

```
#main-header {
  font-size: 24px; /* Targets element with ID "main-header" */
}
```

2.2 Advanced CSS Selectors

- **Pseudo-classes & Combinators:**
  - :hover ⟶ Styles an element when hovered.
  - :nth-child(n) ⟶ Selects the nth child of a parent.
  - div > p ⟶ Selects p elements that are direct children of a div.
- **Example:**

```
button:hover {
  background-color: green;
}

p:nth-child(2) {
  font-weight: bold;
}
```

📌 **Hovering over the button changes its background.**

CHAPTER 3: SELECTING ELEMENTS USING JAVASCRIPT

3.1 JavaScript Methods for Selecting Elements

JavaScript provides multiple ways to **select and modify elements dynamically**.

- **Methods to Select Elements:**

Method	Description	Example
document.getElementById()	Selects an element by ID.	document.getElementById("header")
document.getElementsByClassName()	Selects elements by class.	document.getElementsByClassName("box")
document.getElementsByTagName()	Selects elements by tag.	document.getElementsByTagName("p")
document.querySelector()	Selects the **first** matching element.	document.querySelector(".btn")
document.querySelectorAll()	Selects **all** matching elements.	document.querySelectorAll("li")

- **Example: Changing an Element's Text Using JavaScript**

```
<h1 id="title">Hello World</h1>

<button onclick="changeText()">Click Me</button>


<script>

  function changeText() {

    document.getElementById("title").innerText = "Hello, JavaScript!";

  }

</script>
```

📌 **Clicking the button changes the text inside the <h1> tag.**

CHAPTER 4: MODIFYING ELEMENTS USING JAVASCRIPT

4.1 Changing Styles Dynamically

JavaScript allows changing an element's **styles dynamically**.

- **Example:**

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

<button onclick="changeColor()">Change Color</button>


<script>

  function changeColor() {

    document.getElementById("text").style.color = "blue";

  }

</script>
```

📌 **Clicking the button changes the text color to blue.**

4.2 Adding and Removing Classes

Instead of modifying styles directly, it's better to **add or remove CSS classes** dynamically.

🔹 **Example:**

```
<p id="paragraph" class="normal">This is a paragraph.</p>

<button onclick="toggleStyle()">Toggle Style</button>


<style>
  .normal { color: black; }
  .highlight { color: red; font-weight: bold; }
</style>


<script>
  function toggleStyle() {

document.getElementById("paragraph").classList.toggle("highlight");
  }
</script>
```

📌 **Clicking the button toggles between normal and highlighted text styles.**

4.3 Adding and Removing Elements

- **Example: Creating a New Element Dynamically**

```
<button onclick="addElement()">Add Element</button>
<div id="container"></div>

<script>
  function addElement() {
    let newPara = document.createElement("p");
    newPara.innerText = "New paragraph added!";
    document.getElementById("container").appendChild(newPara);
  }
</script>
```

📌 **Clicking the button adds a new paragraph inside the container.**

- **Example: Removing an Element Dynamically**

```
<p id="removeMe">This paragraph will be removed.</p>
<button onclick="removeElement()">Remove Paragraph</button>

<script>
  function removeElement() {
    document.getElementById("removeMe").remove();
```

```
}
</script>
```

📌 **Clicking the button removes the paragraph.**

Case Study: How Google Uses JavaScript for Dynamic UI Modifications

Challenges Faced

- Improving **user interactions** without reloading the page.
- **Dynamically updating content** based on user actions.

Solutions Implemented

- Used **event listeners** to modify content dynamically.
- Applied **class toggling** for responsive UI elements.

🔹 **Key Takeaways from Google's Success:**

- **JavaScript enhances user experience by modifying elements dynamically.**
- **Efficient selection methods improve performance and responsiveness.**

📝 Exercise

✅ Use **CSS selectors** to change the background of every odd-numbered list item.
✅ Write JavaScript code to **change the text color of a paragraph when clicked**.
✅ Create a button that **adds and removes a new element dynamically** when clicked.

Conclusion

- **Selecting elements is essential for styling and interactivity.**
- **CSS selectors enable efficient targeting and modifications.**
- **JavaScript provides powerful ways to dynamically modify elements.**
- **Efficiently selecting and modifying elements improves website usability.**

HANDLING USER EVENTS & INTERACTIONS

CHAPTER 1: INTRODUCTION TO USER EVENTS & INTERACTIONS

1.1 What Are User Events & Interactions?

User events and interactions refer to actions performed by users on a webpage, such as **clicking a button, hovering over an element, typing in a form, or scrolling down a page**. Handling these interactions allows developers to create **dynamic and interactive web experiences**.

- **Why Are User Events Important?**
  - Improve **user engagement** with real-time feedback.
  - Enable **interactive components** such as modals, sliders, and dropdowns.
  - Enhance **usability** by providing better navigation and responses.
- **Examples of Common User Events:**

Event Type	Description	Example
click	When an element is clicked	Clicking a button
mouseover	When the mouse hovers over an element	Hover effects
keydown	When a key is pressed	Typing in a form
scroll	When the user scrolls	Lazy loading images

submit	When a form is submitted	Login or signup form

CHAPTER 2: HANDLING EVENTS USING JAVASCRIPT

2.1 Using Event Listeners

An **event listener** allows a webpage to detect when a user interacts with an element and execute a function in response.

- **Example: Handling a Click Event on a Button**

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


<script>
document.getElementById("myButton").addEventListener("click", function() {
  alert("Button Clicked!");
});
</script>
```

This triggers an **alert box** whenever the button is clicked.

2.2 Handling Hover Events (mouseover & mouseout)

Hover events enhance **interactive UI elements**, such as **tooltips or dropdowns**.

- **Example: Changing Background Color on Hover**

```
<div id="hoverBox" style="width: 200px; height: 100px; background: lightgray;">
  Hover Over Me!
```

```
</div>


<script>
let box = document.getElementById("hoverBox");


box.addEventListener("mouseover", function() {
   box.style.backgroundColor = "blue";
});


box.addEventListener("mouseout", function() {
   box.style.backgroundColor = "lightgray";
});
</script>
```

This changes the **background color** when hovering over the box.

CHAPTER 3: ADVANCED EVENT HANDLING TECHNIQUES

3.1 Event Delegation

Event delegation allows you to **attach a single event listener** to a parent element instead of adding multiple listeners to each child element. This improves **performance** and makes handling dynamic elements easier.

- **Example: Handling Click Events for Multiple List Items**

```
<ul id="parentList">
```

```
  <li>Item 1</li>
  <li>Item 2</li>
  <li>Item 3</li>
</ul>

<script>
document.getElementById("parentList").addEventListener("click", function(event) {
  if (event.target.tagName === "LI") {
    alert("You clicked on " + event.target.innerText);
  }
});
</script>
```

Instead of adding a click event to **each <li> element,** we use **event delegation** to detect which item was clicked.

3.2 Preventing Default Actions

Some user interactions, like form submissions and links, trigger **default browser actions** that might not be desirable.

- **Example: Preventing Form Submission**

```
<form id="myForm">
  <input type="text" placeholder="Enter something">
  <button type="submit">Submit</button>
</form>
```

```
<script>
document.getElementById("myForm").addEventListener("submit", function(event) {
  event.preventDefault();
  alert("Form submission prevented!");
});
</script>
```

This prevents the page from **refreshing** when the form is submitted.

CHAPTER 4: HANDLING KEYBOARD AND SCROLL EVENTS

4.1 Detecting Keyboard Input (keydown, keyup)

Keyboard events allow you to **capture user input in real-time,** which is useful for **search fields, hotkeys, and shortcuts**.

- **Example: Detecting Key Presses**

```
<input type="text" id="textInput" placeholder="Type something">

<script>
document.getElementById("textInput").addEventListener("keydown", function(event) {
  console.log("Key Pressed: " + event.key);
});
</script>
```

This logs the **key pressed** in the console.

4.2 Detecting Scroll Events

Scroll events help implement features like **sticky headers, lazy loading, and infinite scrolling**.

- **Example: Detecting Page Scroll**

```
<script>
window.addEventListener("scroll", function() {
  console.log("Scrolled to: " + window.scrollY);
});
</script>
```

This logs the **current scroll position** whenever the user scrolls.

Case Study: How Instagram Enhances User Interactions

Challenges Faced

- Keeping users engaged with **interactive UI elements**.
- Reducing **manual clicks** by making actions intuitive.

Solutions Implemented

- **Scroll events** to **auto-load** new posts when reaching the bottom.
- **Click events** to toggle **likes and comments** dynamically.
- **Hover interactions** for **smooth animations** on elements.

- **Key Takeaways from Instagram's Interaction Strategy:**
  - **Seamless interactions improve user retention.**

- **Real-time event handling enhances the overall experience.**

📝 Exercise

✅ Create a **click event** to toggle the visibility of a paragraph.
✅ Implement an **input field** that detects keystrokes and displays them on the page.
✅ Design a **scroll event** that hides the navbar when scrolling down and shows it when scrolling up.

Conclusion

- **User event handling** makes web pages dynamic and interactive.
- **Event listeners (click, keydown, scroll) allow real-time feedback.**
- **Advanced techniques like event delegation improve performance.**
- **Preventing default actions helps control user interactions better.**

VALIDATING FORM INPUTS USING JAVASCRIPT

CHAPTER 1: INTRODUCTION TO FORM VALIDATION

1.1 What is Form Validation?

Form validation is the process of **checking user input** before submitting it to the server. It ensures that data entered is **correct, complete, and secure**.

- **Why Is Form Validation Important?**
  - Prevents **submission of incorrect or incomplete data**.
  - Improves **user experience** by providing instant feedback.
  - Enhances **security** by preventing **malicious inputs (SQL Injection, XSS attacks, etc.)**.
- **Types of Form Validation:**

1. **Client-side validation** (Using JavaScript) → Fast, reduces server load.
2. **Server-side validation** (Using PHP, Python, Node.js) → Ensures security.

- **Basic Example: Preventing Empty Input Fields**

```
<input type="text" id="name">

<button onclick="validate()">Submit</button>


<script>

function validate() {
```

```
  let name = document.getElementById("name").value;
  if (name === "") {
    alert("Name field cannot be empty!");
  }
}
</script>
```

This prevents the user from submitting an **empty name field**.

CHAPTER 2: METHODS OF JAVASCRIPT FORM VALIDATION

2.1 Real-Time vs. On-Submit Validation

- **Real-time validation** → Validates input as the user types.
- **On-submit validation** → Validates when the user clicks the submit button.
- **Example: Real-Time Email Validation**

```
<input type="email" id="email" oninput="validateEmail()">
<p id="emailError"></p>

<script>
function validateEmail() {
  let email = document.getElementById("email").value;
  let errorMsg = document.getElementById("emailError");
  if (!email.includes("@")) {
    errorMsg.innerText = "Invalid email format!";
```

```
    errorMsg.style.color = "red";
  } else {
    errorMsg.innerText = "";
  }
}
</script>
```

This gives **instant feedback** if the email format is incorrect.

CHAPTER 3: IMPLEMENTING COMMON INPUT VALIDATIONS

3.1 Required Field Validation

Ensures that users do not leave required fields blank.

- **Example: Checking Empty Fields**

```
function validateForm() {
  let username = document.getElementById("username").value;
  if (username.trim() === "") {
    alert("Username is required!");
    return false;
  }
  return true;
}
```

3.2 Email Validation

Ensures email follows **correct format (example@domain.com)**.

- **Example: Using Regular Expressions (Regex) for Email Validation**

```
function validateEmail(email) {
  let regex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
  return regex.test(email);
}

console.log(validateEmail("test@example.com")); // Output: true
console.log(validateEmail("invalid-email")); // Output: false
```

3.3 Password Strength Validation

Ensures password contains **uppercase, lowercase, numbers, and special characters**.

- **Example: Password Strength Validation**

```
function validatePassword(password) {
  let regex = /^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[@$!%*?&])[A-Za-z\d@$!%*?&]{8,}$/;
  return regex.test(password);
}

console.log(validatePassword("Strong@123")); // Output: true
console.log(validatePassword("weakpass")); // Output: false
```

3.4 Phone Number Validation

Checks if the phone number is **exactly 10 digits**.

- **Example:**

```
function validatePhone(phone) {
  let regex = /^[0-9]{10}$/;
  return regex.test(phone);
}

console.log(validatePhone("1234567890")); // Output: true
console.log(validatePhone("12345abc90")); // Output: false
```

CHAPTER 4: VALIDATING ENTIRE FORM ON SUBMISSION

4.1 Preventing Form Submission if Errors Exist

- **Example: Validating Multiple Fields Before Submission**

```
<form onsubmit="return validateForm()">
  <input type="text" id="username" placeholder="Enter username">
  <input type="email" id="email" placeholder="Enter email">
  <input type="password" id="password" placeholder="Enter password">
  <button type="submit">Submit</button>
</form>

<script>
function validateForm() {
  let username = document.getElementById("username").value;
```

```
    let email = document.getElementById("email").value;
    let password = document.getElementById("password").value;

    if (username === "" || email === "" || password === "") {
        alert("All fields are required!");
        return false; // Prevent form submission
    }

    if (!validateEmail(email)) {
        alert("Invalid email format!");
        return false;
    }

    if (!validatePassword(password)) {
        alert("Weak password! It must include uppercase, lowercase, numbers, and special characters.");
        return false;
    }

    return true; // Allow form submission
}

function validateEmail(email) {
```

```
  let regex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
  return regex.test(email);
}

function validatePassword(password) {
  let regex = /^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[@$!%*?&])[A-Za-z\d@$!%*?&]{8,}$/;
  return regex.test(password);
}
</script>
```

This **validates multiple fields** and prevents submission **if any input is invalid**.

Case Study: How Amazon Implements Form Validation to Prevent Errors

Challenges Faced

- Users **enter incorrect email formats or weak passwords** during registration.
- Preventing **fake or duplicate accounts**.

Solutions Implemented

- Used **real-time validation** to highlight errors while typing.
- Applied **strong password policies** to improve security.
- Displayed **error messages in real-time** without refreshing the page.

🔹 **Key Takeaways:**

- **Instant feedback prevents frustration and improves user experience.**
- **Strict validation rules enhance security.**

📝 **Exercise**

✅ Create a **login form with JavaScript validation** for email and password.
✅ Implement **real-time validation** for a phone number field.
✅ Use JavaScript **to highlight empty fields in red when the form is submitted**.

Conclusion

- **Form validation ensures accurate and secure user input.**
- **JavaScript provides real-time and on-submit validation.**
- **Regular expressions (Regex) are useful for complex validations like email and passwords.**
- **Implementing validation improves user experience and prevents security vulnerabilities.**

STORING AND RETRIEVING DATA FROM LOCAL STORAGE

CHAPTER 1: INTRODUCTION TO LOCAL STORAGE

1.1 What is Local Storage?

Local Storage is a **web storage API** that allows websites to store key-value pairs in a **user's browser** with no expiration time. This means that the data remains **available even after the browser is closed and reopened**.

- **Why Use Local Storage?**
  - Stores **persistent data** without requiring a database.
  - Speeds up website loading by storing frequently used data.
  - Enhances **user experience** by saving preferences, login states, or cart items.
- **How Local Storage Works:**
  1. Data is stored in the **browser** instead of the server.
  2. Unlike cookies, **local storage does not expire** unless manually removed.
  3. Data can only be accessed by the **same origin (same domain and protocol)**.
- **Example: Checking Available Local Storage in Browser**
  1. Open **Google Chrome** → Right-click → **Inspect (DevTools)**.
  2. Go to **Application Tab** → **Storage** → **Local Storage**.

CHAPTER 2: STORING DATA IN LOCAL STORAGE

2.1 Adding Data to Local Storage

The localStorage.setItem() method **stores data** in the browser.

- **Example: Storing User Data**

```
localStorage.setItem("username", "JohnDoe");

console.log("User saved in local storage.");
```

This saves the **key-value pair** (username: JohnDoe) in local storage.

- **How to Verify Stored Data:**

1. Open **Chrome DevTools (F12)**.
2. Navigate to **Application → Local Storage**.
3. Look for username: "JohnDoe" under the storage section.

2.2 Storing Complex Data (Objects & Arrays)

Local Storage only supports **strings**, so objects and arrays must be converted into **JSON format** using JSON.stringify().

- **Example: Storing an Object in Local Storage**

```
const user = {
  name: "Alice",
  age: 25,
  country: "USA"
};

localStorage.setItem("userData", JSON.stringify(user));
```

This stores the object as a **string** in local storage.

CHAPTER 3: RETRIEVING DATA FROM LOCAL STORAGE

3.1 Retrieving Simple Data

The localStorage.getItem() method **fetches data** from local storage.

- **Example: Retrieving User Data**

```
let username = localStorage.getItem("username");

console.log("Hello, " + username);
```

If the value exists, it **displays: "Hello, JohnDoe"**.

3.2 Retrieving Objects & Converting from JSON

Since data is stored as a string, **parse it back into an object** using JSON.parse().

- **Example: Retrieving a Stored Object**

```
let userData = JSON.parse(localStorage.getItem("userData"));

console.log(userData.name); // Output: Alice
```

CHAPTER 4: REMOVING DATA FROM LOCAL STORAGE

4.1 Deleting a Specific Item

The localStorage.removeItem() method removes **a single key-value pair**.

- **Example:**

```
localStorage.removeItem("username");

console.log("User removed from local storage.");
```

4.2 Clearing All Data

To **delete everything** from local storage, use clear().

- **Example:**

```
localStorage.clear();

console.log("All local storage data cleared.");
```

CHAPTER 5: USE CASES OF LOCAL STORAGE

5.1 Storing User Preferences

Local storage can **remember user settings** like dark mode or language preferences.

- **Example: Saving Dark Mode Preference**

```
localStorage.setItem("theme", "dark");

let theme = localStorage.getItem("theme");

if (theme === "dark") {

  document.body.style.backgroundColor = "#333";

}
```

This **automatically applies dark mode** based on saved preferences.

5.2 Storing Shopping Cart Data

E-commerce sites use local storage to **store cart items** for users who haven't checked out yet.

- **Example: Adding Items to a Cart**

```
let cart = ["Laptop", "Mouse", "Keyboard"];
```

localStorage.setItem("shoppingCart", JSON.stringify(cart));

Even if the page is refreshed, the **cart items remain stored**.

Case Study: How Amazon Uses Local Storage for User Experience

Challenges Faced

- Users abandoning shopping carts before completing a purchase.
- Preference settings lost upon **page reload**.

Solutions Implemented

- Amazon **stores cart items in local storage** so they persist across sessions.
- User preferences like **recent searches and dark mode** are saved.

🔹 **Key Takeaways from Amazon's Strategy:**

- **Local Storage enhances the user experience by keeping session data.**
- **Persistent storage prevents loss of shopping cart data.**

📝 Exercise

✅ Store a **user's favorite color** in local storage and apply it as the background color.
✅ Save and retrieve **a to-do list** using local storage.
✅ Implement a **"Remember Me" login feature** using local storage.

Conclusion

- **Local Storage provides persistent storage** in the browser without expiry.
- **Data is stored in key-value pairs** and remains available across page reloads.
- **JSON methods (stringify() & parse())** help store and retrieve complex data.
- **Useful for storing user settings, form inputs, and cart data.**

ASSIGNMENT:

CREATE AN INTERACTIVE FORM WITH VALIDATION AND LOCAL STORAGE IMPLEMENTATION.

Step-by-Step Guide to Creating an Interactive Form with Validation and Local Storage Implementation

📌 Step 1: Setting Up the Project

1.1 Create Project Files

1. **Create a project folder** (e.g., interactive-form).
2. **Inside the folder, create the following files:**
   - index.html → Main HTML file.
   - style.css → CSS file for styling.
   - script.js → JavaScript file for validation and Local Storage handling.

📌 Step 2: Creating the HTML Form

🛠 Define the Form Structure in index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Interactive Form</title>
```

```
  <link rel="stylesheet" href="style.css">
</head>
<body>

  <div class="container">
    <h2>Register Here</h2>
    <form id="userForm">
      <label for="name">Name:</label>
      <input type="text" id="name" placeholder="Enter your name">
      <small class="error-message" id="nameError"></small>

      <label for="email">Email:</label>
      <input type="email" id="email" placeholder="Enter your
email">
      <small class="error-message" id="emailError"></small>

      <label for="password">Password:</label>
      <input type="password" id="password" placeholder="Enter
your password">
      <small class="error-message" id="passwordError"></small>

      <button type="submit">Submit</button>
    </form>
```

```
    <h3>Saved Data:</h3>
    <div id="savedData"></div>
  </div>

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

</body>
</html>
```

📌 **This form contains fields for Name, Email, and Password along with placeholders for error messages.**

📌 **Step 3: Adding Styles for Better UI**

🛠 **Styling the Form in style.css**

```
body {
  font-family: Arial, sans-serif;
  background-color: #f8f9fa;
  text-align: center;
}

.container {
  width: 50%;
```

```
    margin: 20px auto;
    padding: 20px;
    background: white;
    box-shadow: 0px 0px 10px rgba(0, 0, 0, 0.1);
    border-radius: 5px;
}

h2, h3 {
    color: #333;
}

form {
    display: flex;
    flex-direction: column;
    align-items: center;
}

input {
    width: 80%;
    padding: 8px;
    margin: 8px 0;
    border: 1px solid #ccc;
```

```
    border-radius: 5px;
}

button {
    background-color: #28a745;
    color: white;
    border: none;
    padding: 10px 15px;
    margin-top: 10px;
    cursor: pointer;
    border-radius: 5px;
}

button:hover {
    background-color: #218838;
}

.error-message {
    color: red;
    font-size: 12px;
}
```

📌 **This improves form styling and provides a clean layout.**

📌 Step 4: Implementing JavaScript for Form Validation

🛠 Add Validation Logic in script.js

```
document.addEventListener("DOMContentLoaded", function () {
    const form = document.getElementById("userForm");
    const nameInput = document.getElementById("name");
    const emailInput = document.getElementById("email");
    const passwordInput = document.getElementById("password");
    const nameError = document.getElementById("nameError");
    const emailError = document.getElementById("emailError");
    const passwordError =
document.getElementById("passwordError");
    const savedDataDiv = document.getElementById("savedData");

    // Load stored data on page load
    loadSavedData();

    form.addEventListener("submit", function (event) {
        event.preventDefault(); // Prevent form submission

        let isValid = true;

        // Validate Name
```

```
    if (nameInput.value.trim() === "") {
      nameError.textContent = "Name is required!";
      isValid = false;
    } else {
      nameError.textContent = "";
    }

    // Validate Email
    if (!validateEmail(emailInput.value)) {
      emailError.textContent = "Enter a valid email!";
      isValid = false;
    } else {
      emailError.textContent = "";
    }

    // Validate Password
    if (passwordInput.value.length < 6) {
      passwordError.textContent = "Password must be at least 6 characters!";
      isValid = false;
    } else {
      passwordError.textContent = "";
    }
```

```
    // If form is valid, save data
    if (isValid) {
      saveToLocalStorage(nameInput.value, emailInput.value,
passwordInput.value);
      loadSavedData();
      form.reset();
    }
  });

  function validateEmail(email) {
    return /^[^\s@]+@[^\s@]+\.[^\s@]+$/.test(email);
  }

  function saveToLocalStorage(name, email, password) {
    const userData = { name, email, password };
    localStorage.setItem("userData", JSON.stringify(userData));
  }

  function loadSavedData() {
    const storedData =
JSON.parse(localStorage.getItem("userData"));
    if (storedData) {
```

```
        savedDataDiv.innerHTML = `
          <p><strong>Name:</strong> ${storedData.name}</p>
          <p><strong>Email:</strong> ${storedData.email}</p>
          <p><strong>Password:</strong> ${"*".repeat(storedData.password.length)}</p>
        `;
    } else {
        savedDataDiv.innerHTML = "<p>No data saved yet.</p>";
    }
  }
});
```

📌 **This JavaScript script performs:**

✓ **Form validation** before submission.

✓ **Local Storage implementation** to save form data.

✓ **Auto-load of saved data** upon page reload.

📌 Step 5: Testing & Debugging

✅ Steps to Test the Form

1. **Enter invalid details** (e.g., an empty name or short password) → Check error messages.
2. **Enter valid details and submit the form** → Data should be saved.
3. **Reload the page** → Previously entered data should appear.

✅ Expected Behaviors

✓ If fields are empty or incorrect, error messages should appear.
✓ If all fields are valid, the data is stored in **Local Storage**.
✓ Reloading the page should retrieve and display the saved data.

🎯 Final Outcome

✓ A fully interactive form with validation.
✓ Saved user data using Local Storage.
✓ Displayed stored data upon page reload.
✓ Ensured proper error handling and usability.

📌 Bonus: Enhancing the Form with Additional Features

1️⃣ Add a "Clear Data" Button

Modify index.html:

```
<button id="clearData">Clear Data</button>
```

Modify script.js:

```
document.getElementById("clearData").addEventListener("click", function () {
    localStorage.removeItem("userData");
    savedDataDiv.innerHTML = "<p>No data saved yet.</p>";
});
```

📌 **This allows users to reset saved data.**

2️⃣ Add Password Toggle Visibility

Modify index.html:

<input type="password" id="password" placeholder="Enter your password">

<input type="checkbox" id="showPassword"> Show Password

Modify script.js:

```
document.getElementById("showPassword").addEventListener("change", function () {
    passwordInput.type = this.checked ? "text" : "password";
});
```

📌 **This enables toggling password visibility.**

Conclusion

✓ Created a fully functional interactive form.
✓ Implemented real-time form validation using JavaScript.
✓ Used Local Storage to save and retrieve form data.
✓ Ensured a responsive and user-friendly experience.