Day 3: JavaScript Fundamentals & DOM Manipulation

- **Goal:** Build interactive web applications with core JavaScript concepts and DOM manipulation.

Morning Session

1. JavaScript Basics - Variables, Data Types, and Operators

What are Variables?

Variables are like labeled boxes that store information. JavaScript has three ways to create them:

```
let name = "Alex";    // Can be changed later
const age = 25;       // Cannot be changed (constant)
var oldName = "John"; // Older way (avoid in modern JS)
```

Why use let **and** const**?**

- let: When values need to change (e.g., counter in a game)
- const: For values that shouldn't change (e.g., mathematical constants)

Data Types in JavaScript

JavaScript has 7 fundamental data types:

Type	Example	Description
String	"Hello"	Text data
Number	42, 3.14	Numeric values
Boolean	true, false	Logical values
Array	[1, 2, 3]	Ordered lists
Object	{name: "Alex"}	Key-value pairs
undefined	let x;	Declared but not assigned
null	let x = null;	Intentional empty value

Real-world analogy:

Think of a contact in your phone:

- **String:** Contact name ("Mom")
- **Number:** Phone number (5551234567)
- **Array:** List of their emails (["mom@gmail.com", "mom@work.com"])
- **Object:** All their details combined ({name: "Mom", phone: 5551234567})

2. Step-by-Step Code Examples

A. Working with Variables

```
// 1. Declaring variables
let message;        // undefined (no value yet)
message = "Hello";    // Now it's a string


// 2. Constants
const BIRTH_YEAR = 1990;  // Uppercase for constants
// BIRTH_YEAR = 2000;     // Error! Can't change const


// 3. Variable naming rules
let userName = "Alex";    // camelCase preferred
let $price = 9.99;       // Can start with $
let _internalValue = 42;  // Can start with _
```

B. Data Type Examples

Strings:

```
let singleQuotes = 'Hello';
let doubleQuotes = "World";
let backticks = `Hello ${doubleQuotes}`;  // Template literal (ES6)
console.log(backticks); // "Hello World"
```

Numbers:

```
let integer = 42;
let float = 3.14159;
let scientific = 2.998e8; // 299,800,000 (speed of light)
let hex = 0xFF;         // 255 in hexadecimal
```

Booleans:

```
let isLoggedIn = true;
let hasPermission = false;
// Common in conditionals
if (isLoggedIn) {
  console.log("Welcome back!");
}
```

Arrays:

```
let fruits = ["Apple", "Banana"];
fruits.push("Orange");    // Add to end
fruits[0] = "Mango";     // Change first item
console.log(fruits.length); // 3
```

Objects:

```
let person = {
  name: "Alex",
  age: 30,
  hobbies: ["Reading", "Hiking"]
};

// Accessing properties
console.log(person.name);     // "Alex"
console.log(person["age"]);   // 30 (bracket notation)
person.country = "Canada";    // Add new property
```

C. Type Coercion (Automatic Type Conversion)

JavaScript tries to be "helpful" by converting types automatically:

```
// String concatenation
console.log(10 + "5");      // "105" (number → string)
console.log("Total: " + 100); // "Total: 100"


// Numeric operations
console.log("10" - 5);      // 5 (string → number)
console.log("10"  "2");     // 20
console.log("10" / "2");     // 5


// Loose equality (==) vs strict equality (===)
console.log(10 == "10");     // true (coerces types)
console.log(10 === "10");    // false (different types)
```

Best Practice: Always use === unless you specifically need coercion.

3. Practical Labs with Real-World Context

Lab 1: User Profile Generator (Enhanced)

```
<!DOCTYPE html>
<html>
<body>
 <script>
  // 1. Collect user data
  const user = {
   name: prompt("Enter your name:"),
   age: parseInt(prompt("Enter your age:")),
   isDeveloper: confirm("Are you a developer?")
  };

  // 2. Generate profile HTML
  const profileHTML = `
   <div class="profile">
    <h1>${user.name}</h1>
    <p>Age: ${user.age}</p>
    <p>Status: ${user.isDeveloper ? "Developer" : "Learner"}</p>
    <p>Birth Year: ${new Date().getFullYear() - user.age}</p>
   </div>
  `;

  // 3. Display the profile
  document.body.innerHTML = profileHTML;
 </script>
</body>
</html>
```

Key Features:

- Uses template literals for clean HTML generation
- Incorporates a ternary operator for conditional text
- Calculates birth year dynamically

Lab 2: Shopping Cart Calculator

```
// Product prices
const products = [
 { name: "Laptop", price: 999 },
 { name: "Phone", price: 699 },
 { name: "Tablet", price: 399 }
];

// User's cart
const cart = [
 { product: products[0], quantity: 1 },
 { product: products[1], quantity: 2 }
];

// Calculate total
let subtotal = 0;
cart.forEach(item => {
 subtotal += item.product.price  item.quantity;
});

const tax = subtotal  0.08;
const total = subtotal + tax;

console.log(`
 Subtotal: $${subtotal.toFixed(2)}
 Tax (8%): $${tax.toFixed(2)}
 Total: $${total.toFixed(2)}
`);
```

Real-World Application:

- Demonstrates working with arrays of objects
- Shows practical calculations with decimal formatting
- Models an e-commerce scenario

4. Common Mistakes & Debugging Tips

Mistake 1: Reassigning Constants

```
const PI = 3.14;
PI = 3.14159; // TypeError: Assignment to constant variable
```

Fix: Use let if the value needs to change.

Mistake 2: Implicit Global Variables

```
function createUser() {
  username = "Alex"; // Missing 'let'/'const' → global variable!
}
```

Fix: Always declare variables with let/const.

Mistake 3: NaN (Not a Number)

```
const result = "abc" / 2; // NaN
console.log(result === NaN); // false! (NaN is special)
console.log(isNaN(result)); // true (correct check)
```

Better Alternative:

```
console.log(Number.isNaN(result)); // More reliable
```

5. Key Takeaways Cheat Sheet

```
# JavaScript Basics Cheat Sheet
## Variables
- `let`: Reassignable variables
- `const`: Unchangeable constants
- `var`: Legacy (avoid in modern JS)

## Data Types
1. Primitives:
  - `String`: text
  - `Number`: integers/decimals
  - `Boolean`: true/false
  - `null`/`undefined`: empty values

2. Structural:
  - `Array`: ordered lists `[1, 2, 3]`
  - `Object`: key-value pairs `{key: value}`

## Operators
- Arithmetic: `+`, `-`, ``, `/`
- Comparison: `==` (loose), `===` (strict)
- Logical: `&&` (AND), `||` (OR), `!` (NOT)

## Best Practices
1. Use `const` by default, `let` when needed
2. Prefer `===` over `==`
3. Use template literals (`${variable}`) for strings
4. Declare variables at the top of their scope
```

6. Real-World Use Cases

1. Form Input Handling

```
// Get form values
const email = document.getElementById("email").value;
const password = document.getElementById("password").value;

// Validate
```

```
if (email.includes("@") && password.length >= 8) {
  loginUser(email, password);
}
```

2. Game Development

```
// Player stats
const player = {
  name: "Hero",
  health: 100,
  inventory: ["Sword", "Potion"]
};

// Damage calculation
function takeDamage(amount) {
  player.health -= amount;
  if (player.health <= 0) gameOver();
}
```

3. Dynamic UI Updates

```
// Update online user count
const onlineUsers = 42;
document.getElementById("user-count").textContent =
  `${onlineUsers} users online`;
```

Functions & Conditionals in JavaScript

1. Core Concepts Explained Simply

What Are Functions?

Functions are reusable blocks of code that:

- Perform specific tasks
- Can accept inputs (parameters)
- Can return outputs
- Help organize and modularize code

Real-world analogy:

Think of a function like a coffee machine:

- **Input (parameters):** Coffee beans, water
- **Process:** Brewing
- **Output (return value):** Cup of coffee

Types of Functions

1. **Function Declarations**

```
function greet(name) {
 return `Hello ${name}`;
}
```

    a. Hoisted (can be called before declaration)

    b. Good for general-purpose functions

2. **Arrow Functions (ES6)**

```
const greet = name => `Hello ${name}`;
```

   a. Concise syntax

   b. Lexical this binding (better for callbacks)

   c. Must be defined before use

Conditionals

Ways to make decisions in code:

1. **if/else statements** - For complex conditions
2. **Ternary operator** - For simple true/false decisions
3. **switch statements** - For multiple fixed cases

2. Step-by-Step Code Examples

A. Function Variations

Basic Function:

```
// Declaration
function calculateArea(width, height) {
  return width  height;
}
// Arrow equivalent
const calculateArea = (width, height) => width  height;
```

Default Parameters:

```
function createUser(name, status = "active") {
  return { name, status };
}
console.log(createUser("Alex")); // {name: "Alex", status: "active"}
```

Rest Parameters (Variable Arguments):

```
function sum(...numbers) {
  return numbers.reduce((total, num) => total + num, 0);
}
console.log(sum(1, 2, 3)); // 6
```

B. Conditional Statements

if/else:

```
function getTicketPrice(age) {
  if (age < 5) {
    return "Free";
  } else if (age < 18) {
    return "$10";
  } else if (age < 65) {
    return "$20";
  } else {
    return "$15";
  }
}
```

Ternary Operator:

```
const isMember = true;
const fee = isMember ? "$5" : "$20";
```

Switch Statement:

```
function getDayName(dayNum) {
 switch(dayNum) {
  case 1: return "Monday";
  case 2: return "Tuesday";
  // ...
  default: return "Invalid day";
 }
}
```

3. Practical Labs with Real-World Context

Lab 1: Enhanced Age Verification App

```
<!DOCTYPE html>
<html>
<head>
 <title>Age Verification</title>
 <style>
  body { font-family: Arial; text-align: center; margin-top: 50px; }
  .granted { color: green; }
  .denied { color: red; }
 </style>
</head>
<body>
 <h1>Age Verification Portal</h1>
 <input id="ageInput" type="number" placeholder="Enter your age">
 <button onclick="verifyAge()">Check Access</button>
 <p id="result"></p>
 <p id="message"></p>

 <script>
  function verifyAge() {
   const age = parseInt(document.getElementById("ageInput").value);
   const resultElement = document.getElementById("result");
   const messageElement = document.getElementById("message");

   if (isNaN(age)) {
    resultElement.textContent = "Please enter a valid age";
```

```
    resultElement.className = "denied";
    messageElement.textContent = "";
    return;
  }

  const status = age >= 18 ? "granted" : "denied";
  resultElement.textContent = `Access ${status}`;
  resultElement.className = status;

  // Additional messages
  if (status === "granted") {
    messageElement.textContent = "Welcome to our service!";
  } else {
    messageElement.textContent = "You must be 18+ to access this content.";
  }
 }
 </script>
</body>
</html>
```

Key Features:

- Input validation
- Dynamic styling based on result
- Additional contextual messages
- Clean separation of concerns

Lab 2: Shopping Cart Discount Calculator

```
function calculateTotal(items, isMember = false) {
 const subtotal = items.reduce((sum, item) => sum + item.price, 0);

 let discount = 0;
 if (subtotal > 100 && isMember) {
  discount = 0.2; // 20% discount
 } else if (subtotal > 50) {
  discount = 0.1; // 10% discount
 }

 const discountAmount = subtotal  discount;
 const total = subtotal - discountAmount;

 return {
  subtotal,
  discount: `${discount  100}% `,
  discountAmount,
  total
 };
}

// Example Usage:
const cartItems = [
 { name: "Shirt", price: 25 },
 { name: "Jeans", price: 50 },
 { name: "Hat", price: 15 }
];

const receipt = calculateTotal(cartItems, true);
console.log(receipt);
/
{
 subtotal: 90,
 discount: "20%",
 discountAmount: 18,
 total: 72
}
/
```

Real-World Application:

- Implements tiered discount logic
- Uses object return for multiple values
- Demonstrates array reduction
- Shows conditional discount application

4. Common Mistakes & Debugging Tips

Mistake 1: Missing Return Statement

```
function add(a, b) {
  a + b; // Oops! No return
}
console.log(add(2, 3)); // undefined
```

Fix: Always include return when needed.

Mistake 2: Arrow Function Braces

```
const multiply = (a, b) => { a  b }; // Needs return or no braces
const multiply = (a, b) => a  b; // Correct
```

Mistake 3: Loose Equality in Conditionals

```
if (user.age == "21") { ... } // Works but dangerous
if (user.age === 21) { ... } // Better
```

Debugging Tip:

```
function complexCalculation(a, b) {
  console.log("Inputs:", a, b); // Debug logging
  const step1 = a  2;
  console.log("Step 1:", step1);
}
```

5. Key Takeaways Cheat Sheet

```
# Functions & Conditionals Cheat Sheet


## Function Types
1. Declaration:
  function name(params) { ... }
```

2. Arrow:

```
const name = (params) => { ... }
```

Parameters

- Defaults: (param = defaultValue)
- Rest: (...args) for variable arguments

Conditionals

1. if/else:

```
if (condition) { ... }
else if { ... }
else { ... }
```

2. Ternary:

```
condition ? trueExpr : falseExpr
```

3. Switch:

```
switch(value) {
  case x: ... break;
  default: ...
}
```

Best Practices

1. Use clear, descriptive function names
2. Keep functions small/single-purpose
3. Prefer strict equality (===)
4. Document complex logic with comments

```
## 6. Real-World Use Cases
1. Form Validation
  ```javascript
  function validateEmail(email) {
    return email.includes("@") &&
          email.length > 5 &&
          email.endsWith(".com");
  }
```

2. **User Authentication**

```
function login(username, password) {
  if (!username || !password) return false;

  const user = findUser(username);
  return user?.password === hash(password);
}
```

3. **Dynamic UI Components**

```
function createNotification(message, type = "info") {
  const element = document.createElement("div");
  element.className = `notification ${type}`;
  element.textContent = message;
  document.body.appendChild(element);

  setTimeout(() => element.remove(), 3000);
}
```
```

Afternoon Session

3. DOM Manipulation & Event Handling

1. Core Concepts Explained Simply

What is the DOM?

The Document Object Model (DOM) is:

- A tree-like representation of your HTML document
- A programming interface for web documents
- What allows JavaScript to interact with your webpage

Real-world analogy:

Think of the DOM as a live construction blueprint of your house (webpage).

JavaScript is like the contractor who can:

- View the blueprint (querySelector)
- Make changes (textContent, style)
- Respond to events (addEventListener)

DOM Manipulation Basics

1. **Selecting Elements** - Finding elements to work with
2. **Modifying Elements** - Changing content, styles, attributes
3. **Creating/Removing Elements** - Adding or deleting nodes

4. **Event Handling** - Making pages interactive

2. Step-by-Step Code Examples

A. Selecting Elements

Single Elements:

```
// By ID (returns single element)
const header = document.getElementById("header");

// By CSS selector (first match)
const btn = document.querySelector(".btn-primary");

// By element type (first match)
const firstPara = document.querySelector("p");
```

Multiple Elements:

```
// All elements with class
const buttons = document.querySelectorAll(".btn");

// All paragraphs
const allParas = document.getElementsByTagName("p");

// HTMLCollection vs NodeList
console.log(buttons.forEach); // Works (NodeList)
console.log(allParas.forEach); // Undefined (HTMLCollection)
```

B. Modifying Elements

Content Modification:

```
// Text content (safer)
element.textContent = "New text";

// HTML content (potential security risk)
```

```
element.innerHTML = "<strong>Bold</strong> text";

// Value for form elements
input.value = "default@email.com";
```

Style Changes:

```
// Individual properties
element.style.color = "red";
element.style.fontSize = "20px"; // Note camelCase

// Multiple properties
Object.assign(element.style, {
  backgroundColor: "black",
  padding: "10px"
});
```

Attributes & Classes:

```
// Get/set attributes
const link = document.querySelector("a");
console.log(link.getAttribute("href"));
link.setAttribute("target", "_blank");

// Class manipulation
element.classList.add("active");
element.classList.remove("inactive");
element.classList.toggle("hidden");

// Data attributes
element.dataset.userId = "123"; // data-user-id
```

C. Creating/Removing Elements

```
// Create new element
const newDiv = document.createElement("div");
newDiv.textContent = "Hello World!";

// Add to DOM
document.body.appendChild(newDiv);
```

```
// Clone existing element
const clonedBtn = btn.cloneNode(true);


// Remove elements
element.remove();
// OR
parent.removeChild(childElement);
```

3. Practical Labs with Real-World Context

Lab 1: Enhanced Dark Mode Toggle

```
<!DOCTYPE html>
<html>
<head>
 <style>
  body {
   transition: background 0.3s, color 0.3s;
  }
  .dark-mode {
   background: #333;
   color: white;
  }
  .dark-mode button {
   background: #555;
   color: white;
  }
 </style>
</head>
<body>
 <button id="darkModeToggle">Toggle Dark Mode</button>
 <h1>Welcome to our Site</h1>
 <p>This is sample content that will change with the theme.</p>


 <script>
  const toggleBtn = document.getElementById("darkModeToggle");
  const prefersDark = window.matchMedia("(prefers-color-scheme: dark)");


  // Initialize based on system preference
  if (prefersDark.matches) {
```

```
    document.body.classList.add("dark-mode");
  }

  // Toggle function
  function toggleDarkMode() {
    document.body.classList.toggle("dark-mode");

    // Save preference
    const isDark = document.body.classList.contains("dark-mode");
    localStorage.setItem("darkMode", isDark);
  }

  // Load saved preference
  if (localStorage.getItem("darkMode") === "true") {
    document.body.classList.add("dark-mode");
  }

  // Event listeners
  toggleBtn.addEventListener("click", toggleDarkMode);

  // Watch for system changes
  prefersDark.addListener(e => {
    document.body.classList.toggle("dark-mode", e.matches);
  });
 </script>
</body>
</html>
```

Key Features:

- Respects system color scheme preference
- Persists user choice with localStorage
- Smooth transitions between modes
- Media query listener for system changes

Lab 2: Interactive Shopping List

```
<div id="app">
 <h1>Shopping List</h1>
 <form id="itemForm">
  <input type="text" id="itemInput" placeholder="Add an item..." required>
  <button type="submit">Add</button>
 </form>
 <ul id="itemList"></ul>
</div>

<script>
 const form = document.getElementById("itemForm");
 const input = document.getElementById("itemInput");
 const list = document.getElementById("itemList");
 let items = JSON.parse(localStorage.getItem("shoppingList")) || [];

 // Render existing items
 function renderItems() {
  list.innerHTML = items.map(item => `
   <li>
    ${item.text}
    <button data-id="${item.id}" class="delete">×</button>
   </li>
  `).join("");
 }

 // Add new item
 form.addEventListener("submit", e => {
  e.preventDefault();

  const newItem = {
   id: Date.now(),
   text: input.value
  };

  items.push(newItem);
  saveItems();
  renderItems();
  input.value = "";
```

```
});

// Delete item
list.addEventListener("click", e => {
  if (e.target.classList.contains("delete")) {
    const id = Number(e.target.dataset.id);
    items = items.filter(item => item.id !== id);
    saveItems();
    renderItems();
  }
});

// Save to localStorage
function saveItems() {
  localStorage.setItem("shoppingList", JSON.stringify(items));
}

// Initial render
renderItems();
</script>
```

Real-World Features:

- Form submission handling
- Dynamic list rendering
- Event delegation for dynamic elements
- Local persistence
- Unique IDs for items

4. Event Handling

Event Types and Patterns

Common Event Types:

```
// Mouse events
element.addEventListener("click", handleClick);
element.addEventListener("mouseenter", showTooltip);
element.addEventListener("mouseleave", hideTooltip);

// Keyboard events
document.addEventListener("keydown", e => {
  if (e.key === "Escape") closeModal();
});

// Form events
form.addEventListener("submit", handleSubmit);
input.addEventListener("input", validateField);

// Window events
window.addEventListener("resize", handleResize);
window.addEventListener("load", initApp);
```

Event Object Properties:

```
button.addEventListener("click", e => {
  console.log(e.target); // The clicked element
  console.log(e.currentTarget); // The element with listener
  console.log(e.clientX, e.clientY); // Mouse position
});
```

Event Delegation Pattern:

```
// Instead of adding listeners to each item:
document.querySelectorAll(".item").forEach(item => {
  item.addEventListener("click", handleClick);
});
```

```
// Add one listener to parent:
list.addEventListener("click", e => {
 if (e.target.matches(".item")) {
  handleClick(e);
 }
});
```

5. Key Takeaways Cheat Sheet

```
# DOM Manipulation Cheat Sheet

## Selecting Elements
- `getElementById()` - Single element by ID
- `querySelector()` - First match of CSS selector
- `querySelectorAll()` - All matches (NodeList)

## Modifying Elements
- `textContent` - Safe text insertion
- `innerHTML` - HTML insertion (caution: XSS risk)
- `classList` - Add/remove/toggle classes
- `style` - Modify CSS properties

## Event Handling
- `addEventListener(type, callback)`
- Common types: `click`, `submit`, `keydown`
- Event object contains:
 - `target` - Originating element
 - `preventDefault()` - Stop default behavior
 - `stopPropagation()` - Stop bubbling

## Best Practices
1. Cache DOM references (don't requery)
2. Use event delegation for dynamic content
3. Prefer `textContent` over `innerHTML` for text
4. Clean up event listeners when needed
```

6. Real-World Use Cases

1. Form Validation

```
const emailInput = document.getElementById("email");

emailInput.addEventListener("input", () => {
  const isValid = emailInput.value.includes("@");
  emailInput.style.borderColor = isValid ? "green" : "red";
});
```

2. Modal Window

```
function openModal() {
  const modal = document.createElement("div");
  modal.className = "modal";
  modal.innerHTML = `
    <div class="modal-content">
      <span class="close">&times;</span>
      <h2>Important Message</h2>
      <p>This is a modal dialog.</p>
    </div>
  `;

  modal.querySelector(".close").addEventListener("click", () => {
    modal.remove();
  });

  document.body.appendChild(modal);
}
```

3. Infinite Scroll

```
window.addEventListener("scroll", () => {
  if (window.innerHeight + window.scrollY >= document.body.offsetHeight - 500) {
    loadMoreContent();
  }
});
```

Project (1):

- ## Interactive Task Manager

```
<div id="app">
 <input id="taskInput" placeholder="New task">
 <button id="addBtn">Add Task</button>
 <ul id="taskList"></ul>
</div>
<script>
 const tasks = [];
 document.getElementById("addBtn").addEventListener("click", () => {
  const input = document.getElementById("taskInput");
  tasks.push(input.value);
  input.value = "";
  renderTasks();
 });

 function renderTasks() {
  const list = document.getElementById("taskList");
  list.innerHTML = tasks.map(task => `<li>${task}</li>`).join("");
 }
</script>
```