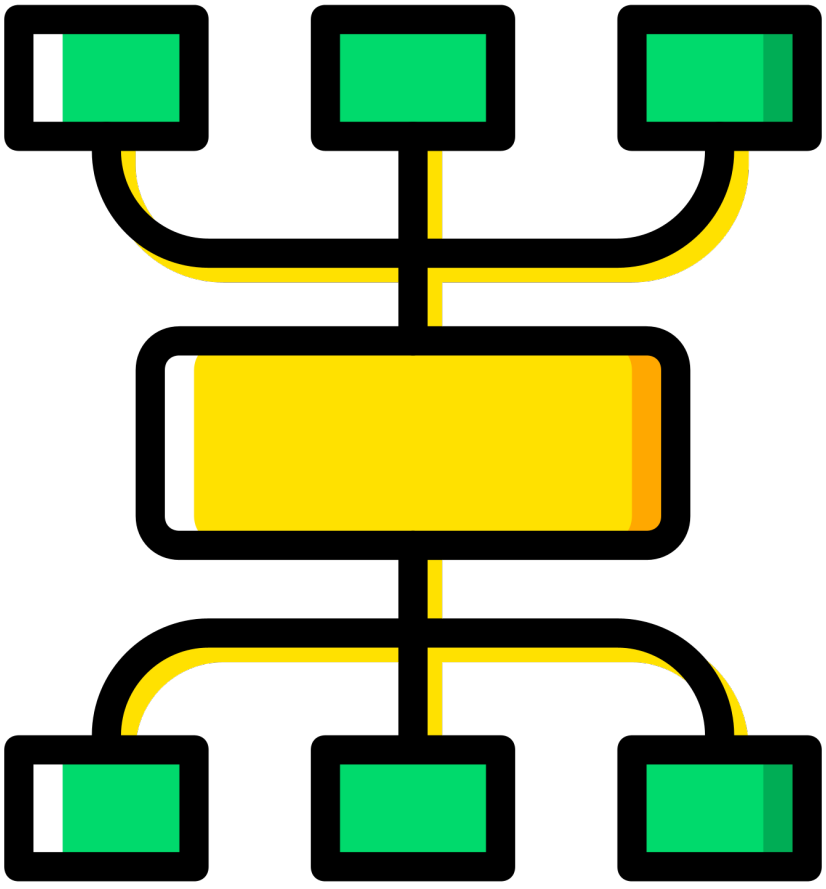
JavaScript DOM Programming



readbytes

JavaScript DOM Programming

From Basics to Advanced Techniques

readbytes.github.io

2025-07-12

This page is intentionally left blank.

Contents

Chapter 1.

Introduction to the DOM

1. What Is the DOM?
2. The DOM Tree Structure
3. How JavaScript Interacts with the DOM
4. Browser Developer Tools for DOM Inspection
5. Your First DOM Manipulation Example

1 Introduction to the DOM

1.1 What Is the DOM?

The **Document Object Model (DOM)** is a programming interface used by web browsers to represent and interact with HTML and XML documents. At its core, the DOM transforms the static HTML content of a web page into a **dynamic, tree-like structure of objects** that JavaScript can access and manipulate.

Think of the DOM as a Living Tree

Imagine your web page as a **family tree**. At the very top is the main ancestor—called the **document**. This document has children like `<html>`, which in turn has children like `<head>` and `<body>`. These elements have their own children—like `<p>`, `<div>`, `<h1>`, and so on.

Here's a basic HTML snippet:

```
<!DOCTYPE html>
<html>
  <head>
    <title>My Page</title>
  </head>
  <body>
    <h1>Hello, World!</h1>
    <p>This is my first DOM example.</p>
  </body>
</html>
```

The DOM represents this as a **tree structure**, where each HTML tag becomes a **node** in the tree.

```
document
+-- html
    +-- head
        +-- title
    +-- body
        +-- h1
        +-- p
```

Each node in this tree is an **object** that JavaScript can work with—read its content, change its attributes, insert new nodes, or remove them entirely.

1.1.1 Why the DOM Matters

Without the DOM, web pages would be static and unchangeable once loaded. The DOM gives us the power to:

- **Access elements** (e.g., get the text inside a `<p>` tag)
- **Change styles** (e.g., change background color when a button is clicked)
- **Add or remove content** dynamically
- **Respond to events** (e.g., mouse clicks or key presses)

In short, **the DOM is what allows JavaScript to make web pages interactive.**

1.1.2 The DOM Is an Abstraction

It's important to understand that the DOM is **not the HTML itself**. Rather, it's a **representation** of the HTML document created by the browser. When the browser loads a web page, it parses the HTML and builds a corresponding DOM in memory.

This means that the DOM can change—even if the original HTML stays the same—because JavaScript can modify the DOM after the page has loaded.

1.1.3 Real-World Analogy

Think of the HTML file as a **blueprint** for a house, and the DOM as the **actual house** built from that blueprint. While the blueprint is fixed, the real house can be changed—furniture can be moved, walls painted, and rooms added. JavaScript is like the crew that performs those changes.

1.1.4 Summary

- The DOM is a **tree-like structure** representing the content and structure of a web page.
- It is built by the **browser** when the HTML is parsed.
- JavaScript uses the DOM to **dynamically interact** with a web page: reading content, changing layout, handling user input, and more.
- The DOM is **not the HTML** itself but an abstraction of it—allowing for dynamic, programmatic control over the document.

1.2 The DOM Tree Structure

At the heart of the Document Object Model (DOM) lies its **tree-like structure**, which represents an HTML document as a hierarchy of **nodes**. This structure is essential for

understanding how JavaScript can navigate, query, and manipulate different parts of a web page.

1.2.1 What Is the DOM Tree?

The DOM organizes everything in an HTML document—elements, text, comments, and attributes—into a **tree of nodes**. Each piece of the document is a **node**, and nodes are connected through **parent**, **child**, and **sibling** relationships.

Let's look at a simple HTML document and see how it maps to the DOM tree:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Sample</title>
  </head>
  <body>
    <h1>Hello</h1>
    <p>Welcome to the DOM</p>
  </body>
</html>
```

This HTML becomes the following **DOM tree**:

```
document
+-- html
    +-- head
        +-- title
            +-- "Sample"
    +-- body
        +-- h1
            +-- "Hello"
        +-- p
            +-- "Welcome to the DOM"
```

1.2.2 Types of Nodes in the DOM

The DOM includes several **node types**, each representing a different kind of content:

Node Type	Description
`Element`	HTML tags like `<div>`, `<p>`, `<ul>`
`Text`	Text inside an element (e.g., `"Hello"`)
`Comment`	Content inside `<!-- -->`

Node Type	Description
`Attribute`	Properties inside tags (e.g., `class="box"`)

These nodes make up the building blocks of the DOM.

1.2.3 Relationships: Parent, Child, and Sibling

DOM nodes are **connected in a hierarchy**:

- **Parent node**: The node directly above (e.g., `<body>` is the parent of `<h1>`)
- **Child node**: A node directly under another (e.g., `<h1>` is a child of `<body>`)
- **Sibling node**: Nodes with the same parent (e.g., `<h1>` and `<p>` are siblings)

Understanding these relationships is key to navigating the DOM.

1.2.4 Mapping an HTML Example to the DOM Tree

Here's another example:

```
<ul>
  <li>First</li>
  <li>Second</li>
</ul>
```

DOM tree:

```
ul
+-- li
    +-- "First"
+-- li
    +-- "Second"
```

Each `<li>` is a **child** of `<ul>`, and the two `<li>` elements are **siblings**. The text inside each `<li>` is a **text node** child of its parent element.

1.2.5 Why This Tree Structure Matters

The tree structure allows you to:

- **Traverse** the document: move up, down, or sideways through the nodes

- **Select** nodes precisely using tools like `document.querySelector`
- **Modify** content by adding, removing, or updating nodes

This structure is **consistent across all HTML pages**, making it a reliable foundation for interacting with documents programmatically.

1.2.6 Summary

- The DOM is a **tree of nodes** representing the structure of an HTML document.
- Nodes have **parent**, **child**, and **sibling** relationships.
- **Element**, **text**, **comment**, and **attribute** nodes make up most of the DOM.
- Understanding the DOM tree helps you **select, traverse, and manipulate** HTML effectively using JavaScript.

1.3 How JavaScript Interacts with the DOM

JavaScript interacts with the DOM through a powerful set of APIs provided by the browser. These APIs allow scripts to **read**, **modify**, **add**, or **remove** elements, attributes, and content in real-time. This interaction is at the heart of dynamic, interactive web applications.

1.3.1 The DOM Is a Live Interface

The DOM is not a static snapshot of the HTML document—**it's a live, dynamic structure**. When JavaScript changes the DOM (e.g., updates an element's text or style), the browser immediately reflects these changes in the visible page. This makes JavaScript a core tool for creating responsive user interfaces.

1.3.2 DOM Manipulation: The Basics

Here are some of the key operations JavaScript can perform on the DOM:

Action	Description
Select	Find elements with `document.querySelector()` or `getElementById()`
Read/Write Content	Get or set text and HTML with `.textContent`, `.innerHTML`, or `.value`

Action	Description
Change Attributes	Use `.setAttribute()`, `.getAttribute()`, or access `.id`, `.className`, etc.
Modify Styles	Access `.style` to change visual appearance
Create/Remove Elements	Use `document.createElement()` and `appendChild()` or `removeChild()`
Attach Events	Register user interaction handlers with `.addEventListener()`

These capabilities will be explored in depth in later chapters.

1.3.3 Example: Reading and Changing DOM Content

Let's walk through a simple example to see JavaScript interacting with the DOM:

HTML:

```
<p id="greeting">Hello, world!</p>
<button id="changeText">Change Greeting</button>
```

JavaScript:

```
// Select elements
const greeting = document.getElementById("greeting");
const button = document.getElementById("changeText");

// Attach an event to update the DOM
button.addEventListener("click", () => {
  greeting.textContent = "Hi there, JavaScript!";
});
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>DOM Interaction Example</title>
  <style>
    body {
      font-family: sans-serif;
      padding: 40px;
      text-align: center;
    }
    button {
      padding: 10px 20px;
      font-size: 16px;
    }
  </style>
</head>
<body>
```

```
  <p id="greeting">Hello, world!</p>
  <button id="changeText">Change Greeting</button>

  <script>
    // Select elements
    const greeting = document.getElementById("greeting");
    const button = document.getElementById("changeText");

    // Attach an event to update the DOM
    button.addEventListener("click", () => {
      greeting.textContent = "Hi there, JavaScript!";
    });
  </script>

</body>
</html>
```

How it works:

1. The `getElementById` method locates the elements in the DOM.
2. An event listener waits for the button to be clicked.
3. When clicked, JavaScript changes the `textContent` of the paragraph element.
4. The browser immediately updates the display on the page—no reload required!

1.3.4 The DOM and JavaScript: A Dynamic Relationship

Because the DOM is live:

- DOM queries always reflect the current document structure.
- Changes made through JavaScript persist until changed again or the page reloads.
- DOM elements can be reused, cached, or queried as needed.

This live relationship is what enables animations, form validation, content updates, and other interactive behaviors.

1.3.5 Summary

- JavaScript uses browser-provided DOM APIs to read and modify web page structure and content.
- The DOM is live—changes made by JavaScript are reflected immediately.
- Key actions include selecting elements, modifying content or attributes, adding/removing elements, and responding to user events.
- These interactions form the backbone of modern web interactivity.

1.4 Browser Developer Tools for DOM Inspection

Modern browsers come equipped with powerful **Developer Tools** that allow you to inspect, edit, and debug your HTML, CSS, and JavaScript code directly in the browser. These tools are essential for working with the DOM because they let you **visualize the DOM structure**, **modify elements in real time**, and **interact with it using JavaScript**.

1.4.1 What Are Developer Tools?

Browser Developer Tools (often opened with `F12` or `Right Click → Inspect`) are built into browsers like:

- **Google Chrome** (Chrome DevTools)
- **Mozilla Firefox** (Firefox Developer Tools)
- **Microsoft Edge** (DevTools)
- **Safari** (Web Inspector)

They include features like:

- DOM inspection
- CSS editing
- JavaScript debugging
- Network monitoring
- Console for scripting

1.4.2 Exploring the DOM in the Elements Panel

The **Elements panel** shows the live DOM tree. It's like peeking into how the browser has parsed and built the HTML into nodes.

Try This:

1. **Open DevTools:**
    - Press `F12`, or right-click on any webpage element and choose **Inspect**.
2. **View the Elements panel:**
    - You'll see the current DOM structure of the page.
3. **Hover and Click:**
    - Hover over elements in the tree to highlight them on the page.
4. **Edit on the fly:**

- Double-click an element or right-click → "Edit as HTML" to change content live.

This is extremely useful for testing changes before implementing them in your source code.

1.4.3 Live Style Editing

The **Styles pane** beside the Elements panel shows the CSS rules applied to the selected element. You can:

- **Edit styles live** by clicking any property.
- **Toggle CSS rules** on/off.
- See **inherited styles**, box models, and computed values.

This real-time feedback loop is crucial for fine-tuning appearance and layout.

1.4.4 Using the Console to Interact with the DOM

The **Console panel** is where you can run JavaScript and see output immediately. It gives you full programmatic access to the DOM.

Example:

Let's say you have this element on a page:

```
<p id="message">Hello!</p>
```

You can modify it in the console:

```
document.getElementById("message").textContent = "Changed via Console!";
```

You'll see the page update instantly. This is the same way JavaScript changes the DOM from scripts—only now you're doing it manually for testing.

1.4.5 Tip: \0 and \1 in Chrome

When you inspect an element in the Elements panel, Chrome gives you access to it in the Console as `$0`.

```
$0.textContent = "Updated text";
```

This is super handy for experimentation without needing to write `getElementById()` or

`querySelector()` every time.

1.4.6 Screenshot Walkthrough (If Using Images)

You may include visuals like:

1. **DevTools Open with Elements Panel Highlighted**
2. **Editing an Element's Text Live**
3. **Console Showing `document.getElementById()` Use**

These help beginners orient themselves in the interface.

1.4.7 Summary

- Browser developer tools let you inspect and interact with the DOM directly.
- The **Elements panel** visualizes the live DOM tree and allows HTML/CSS edits.
- The **Console panel** is a JavaScript playground for testing DOM manipulations.
- Tools like `$0`, live-editing, and CSS inspection are invaluable for debugging and learning.

1.5 Your First DOM Manipulation Example

Now that you've learned what the DOM is and how to inspect it in your browser, let's write your very first **DOM manipulation script**. This simple example will show you how to use JavaScript to **select a DOM element** and **change its content and style**—a fundamental skill in web development.

1.5.1 Step 1: Create a Simple HTML Page

Here's a minimal HTML file you can open in your browser. Copy and paste this into a file named `index.html`:

Html:

```
<p id="greeting">Hello, world!</p>
<button id="changeBtn">Change Greeting</button>
```

```javascript
// Select the paragraph element by ID
const greeting = document.getElementById("greeting");

// Select the button element
    const button = document.getElementById("changeBtn");

// Add an event listener to the button
button.addEventListener("click", () => {
      // Change the text content
      greeting.textContent = "You clicked the button!";

      // Change the style
      greeting.style.color = "green";
      greeting.style.fontWeight = "bold";
});
```

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>DOM Manipulation Example</title>
  <style>
    #greeting {
      font-size: 20px;
      color: darkblue;
    }
  </style>
</head>
<body>
  <p id="greeting">Hello, world!</p>
  <button id="changeBtn">Change Greeting</button>

  <script>
    // Select the paragraph element by ID
    const greeting = document.getElementById("greeting");

    // Select the button element
    const button = document.getElementById("changeBtn");

    // Add an event listener to the button
    button.addEventListener("click", () => {
      // Change the text content
      greeting.textContent = "You clicked the button!";

      // Change the style
      greeting.style.color = "green";
      greeting.style.fontWeight = "bold";
    });
  </script>
</body>
</html>
```

1.5.2 Step-by-Step Explanation

Let's break down what's happening in the JavaScript part:

```
const greeting = document.getElementById("greeting");
```

- This **selects the <p> element** with the ID `"greeting"`.
- The result is stored in a variable so we can use it later.

```
const button = document.getElementById("changeBtn");
```

- We also select the `<button>` element that users will click.

```
button.addEventListener("click", () => {
  // code inside runs when the button is clicked
});
```

- This sets up an **event listener**. When the button is clicked, the function inside is triggered.

Inside the event handler:

```
greeting.textContent = "You clicked the button!";
```

- We **change the content** of the `<p>` element using `.textContent`.

```
greeting.style.color = "green";
greeting.style.fontWeight = "bold";
```

- These lines **dynamically update the styles** of the element using `.style`.

1.5.3 Try This Yourself

Open `index.html` in your browser and click the button. You should see the text and color of the paragraph change instantly!

Then try modifying the script:

- Change the text to something else.
- Use `greeting.style.fontSize = "30px"` to make the text larger.
- Add a second button to reverse the change.

1.5.4 Summary

In this section, you:

- Selected DOM elements using `getElementById`
- Attached an event listener with `addEventListener`
- Changed an element's `textContent` and `style`

This small example is the gateway to building **interactive, dynamic websites**. In the next chapters, you'll explore more powerful techniques to read from, write to, and react to DOM changes with confidence and clarity.

Chapter 2.

Selecting and Accessing DOM Elements

1. Basic Selectors: `getElementById()`, `getElementsByClassName()`, `getElementsByTagName()`
2. Modern Selectors: `querySelector()`, `querySelectorAll()`
3. Understanding NodeLists and HTMLCollections
4. Accessing Element Properties and Attributes
5. Practical Examples: Selecting Elements in Various Ways

2 Selecting and Accessing DOM Elements

2.1 Basic Selectors: `getElementById()`, `getElementsByClassName()`, `getElementsByTagName()`

Before we can manipulate elements on a web page, we need to **select them** using JavaScript. The DOM provides several **basic selector methods** to help you target elements efficiently. In this section, you'll learn how to use three foundational methods:

- `getElementById()`
- `getElementsByClassName()`
- `getElementsByTagName()`

2.1.1 `getElementById()`

This method selects a **single element** by its unique `id` attribute.

```
const title = document.getElementById("main-title");
title.textContent = "Welcome!";
```

HTML Example:

```
<h1 id="main-title">Hello</h1>
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Title Update Example</title>
</head>
<body>
  <h1 id="main-title">Hello</h1>

  <script>
    const title = document.getElementById("main-title");
    title.textContent = "Welcome!";
  </script>
</body>
</html>
```

Key Points:

- Returns **one element** (or `null` if none found).
- Fastest DOM lookup method.
- Best used when targeting a unique element.

2.1.2 `getElementsByClassName()`

This method selects **all elements** with a specific class name. It returns a **live HTMLCollection** (not an array!).

```
const items = document.getElementsByClassName("item");

for (let i = 0; i < items.length; i++) {
  items[i].style.color = "blue";
}
```

HTML Example:

```
<ul>
  <li class="item">First</li>
  <li class="item">Second</li>
  <li class="item">Third</li>
</ul>
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Style Items Example</title>
</head>
<body>
  <ul>
    <li class="item">First</li>
    <li class="item">Second</li>
    <li class="item">Third</li>
  </ul>

  <script>
    const items = document.getElementsByClassName("item");

    for (let i = 0; i < items.length; i++) {
      items[i].style.color = "blue";
    }
  </script>
</body>
</html>
```

Key Points:

- Returns a **live collection** that updates if the DOM changes.
- Result is **array-like**, so you can loop through it.
- Commonly used for applying styles or logic to groups of elements.

2.1.3 `getElementsByTagName()`

This method selects **all elements** with a specific tag name like `"p"`, `"div"`, or `"button"`.

```
const paragraphs = document.getElementsByTagName("p");

for (let p of paragraphs) {
  p.style.fontWeight = "bold";
}
```

HTML Example:

```
<p>Intro</p>
<p>Details</p>
<p>Summary</p>
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Bold Paragraphs Example</title>
</head>
<body>
  <p>Intro</p>
  <p>Details</p>
  <p>Summary</p>

  <script>
    const paragraphs = document.getElementsByTagName("p");

    for (let p of paragraphs) {
      p.style.fontWeight = "bold";
    }
  </script>
</body>
</html>
```

Key Points:

- Also returns a **live HTMLCollection**.
- Useful when you want to target all elements of a certain type.

2.1.4 Return Types Summary

Selector	Return Type	Live?	Unique?
`getElementById()`	`Element` or `null`	NO	YES
`getElementsByClassName()`	`HTMLCollection`	YES	NO
`getElementsByTagName()`	`HTMLCollection`	YES	NO

2.1.5 Performance and Specificity

- `getElementById()` is the **most performant** and should be preferred when selecting a unique element.
- `getElementsByClassName()` and `getElementsByTagName()` are slightly **slower**, especially on large documents, since they search the entire DOM unless scoped.
- All three selectors can be **scoped to a parent element** for more efficient queries:

```
const section = document.getElementById("content");
const buttons = section.getElementsByTagName("button");
```

2.1.6 Recap

- Use `getElementById()` for single, unique elements.
- Use `getElementsByClassName()` to select multiple elements sharing a class.
- Use `getElementsByTagName()` to find all elements of a specific type.

These basic selectors are a solid foundation for DOM scripting. In the next section, you'll learn about more flexible and powerful **CSS-style selectors** with `querySelector()` and `querySelectorAll()`.

2.2 Modern Selectors: `querySelector()`, `querySelectorAll()`

JavaScript's modern DOM selection methods — `querySelector()` and `querySelectorAll()` — offer a powerful and flexible way to select elements using **CSS selector syntax**. These methods are more expressive than the older, basic selectors, making them the preferred choice in most modern web development.

2.2.1 `querySelector()`

`querySelector()` returns the **first element** in the document that matches a given **CSS selector**.

```
const header = document.querySelector("h1");
const activeButton = document.querySelector(".btn.active");
const inputByAttr = document.querySelector("input[type='email']");
```

Usage Examples:

```
<h1>Main Title</h1>
<button class="btn active">Click Me</button>
<input type="email" placeholder="Your email" />
```

- Selects by **tag name**: `"h1"`, `"p"`, `"div"`
- By **class**: `".btn"`
- By **id**: `"#main"`
- By **attribute**: `"[type='email']"`
- By **combination**: `".form input[type='text']"`

Returns:

- The **first matching element**, or `null` if none found.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>QuerySelector Example</title>
</head>
<body>
  <h1>Main Title</h1>
  <button class="btn active">Click Me</button>
  <input type="email" placeholder="Your email" />

  <script>
    const header = document.querySelector("h1");
    const activeButton = document.querySelector(".btn.active");
    const inputByAttr = document.querySelector("input[type='email']");

    // Example actions to show they're selected
    header.style.color = "green";
    activeButton.textContent = "I'm Active!";
    inputByAttr.style.border = "2px solid red";
  </script>
</body>
</html>
```

2.2.2 `querySelectorAll()`

`querySelectorAll()` returns **all matching elements** as a **static NodeList**.

```
const buttons = document.querySelectorAll(".btn");
buttons.forEach((btn) => (btn.disabled = true));
```

Example HTML:

```
<button class="btn">Save</button>
<button class="btn">Cancel</button>
```

- Returns a **NodeList** (not live like `getElementsByClassName()`).
- Can be iterated using `forEach()` or converted to an array with `Array.from()` or spread syntax.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Disable Buttons Example</title>
</head>
<body>
  <button class="btn">Save</button>
  <button class="btn">Cancel</button>

  <script>
    const buttons = document.querySelectorAll(".btn");
    buttons.forEach((btn) => (btn.disabled = true));
  </script>
</body>
</html>
```

2.2.3 Complex Selector Capabilities

`querySelector()` and `querySelectorAll()` support the full range of CSS selectors:

Selector Type	Example	Meaning
Class	`.highlight`	Elements with class `highlight`
ID	`#header`	Element with ID `header`
Descendant	`div p`	All `<p>` inside `<div>`
Child	`ul > li`	Immediate children `<li>` of `<ul>`
Attribute	`input[type='text']`	Inputs with `type="text"`
Pseudo-class	`li:first-child`	First `<li>` among siblings
Grouping	`h1, h2, h3`	All headers (h1 through h3)

2.2.4 Advantages Over Basic Selectors

Feature	`getElementById()` etc.	`querySelector()` / `querySelectorAll()`
CSS-style syntax	NO	YES
Select multiple types	NO	YES
Flexible nesting/querying	NO	YES

Feature	`getElementById()` etc.	`querySelector()` / `querySelectorAll()`
Return all matches	`getElementsBy*` only	YES
Returns static results	NO (live)	YES (static NodeList)

2.2.5 Browser Support

- Fully supported in **all modern browsers**, including mobile.
- Safe to use in production environments with no polyfills needed.

2.2.6 Quick Tip: Use `querySelector` for Simplicity

In modern practice, `querySelector()` and `querySelectorAll()` often **replace** older methods like `getElementById()` or `getElementsByClassName()` because they offer **greater flexibility** with cleaner, CSS-like syntax.

```
// Instead of this:
const logo = document.getElementById("site-logo");

// Use this:
const logo = document.querySelector("#site-logo");
```

2.2.7 Practice Exercise

Try selecting the following elements using `querySelector()` or `querySelectorAll()`:

HTML:

```
<div class="card" id="user-card">
  <h2>User Info</h2>
  <p class="name">Alice</p>
  <p class="email">alice@example.com</p>
</div>
```

Tasks:

1. Select the card container.
2. Select the name paragraph using a class.
3. Select the email paragraph using a descendant selector.
4. Select all `<p>` elements inside `.card`.

2.2.8 Recap

- Use `querySelector()` for the **first matching element**.
- Use `querySelectorAll()` for **all matching elements**.
- Leverage **CSS selectors** for powerful, readable queries.
- Prefer these methods in modern JavaScript for flexibility and simplicity.

Next, we'll explore what exactly `NodeList` and `HTMLCollection` are — and how to work with them effectively.

2.3 Understanding NodeLists and HTMLCollections

When you select multiple DOM elements in JavaScript, you often receive a special kind of list — either a **NodeList** or an **HTMLCollection**. These are *array-like* objects, but they are **not true arrays**, and understanding the differences between them is important for writing clean, effective DOM code.

2.3.1 What Are NodeLists and HTMLCollections?

Collection Type	Returned By
HTMLCollection	`getElementsByClassName()`, `getElementsByTagName()`
NodeList	`querySelectorAll()`

Both are *array-like*:

- They have a `length` property
- You can access items with bracket notation (`list[0]`)
- But they lack full array methods like `map()`, `filter()`, or `reduce()`

2.3.2 Live vs. Static Collections

Collection Type	Live or Static?
HTMLCollection	YES *Live*
NodeList	NO *Static* (usually)

Live collections reflect real-time changes in the DOM. **Static collections** do not update

after they're created.

Example:

```
<ul id="myList">
  <li>Item 1</li>
</ul>
```

```
const liveList = document.getElementsByTagName("li"); // HTMLCollection (Live)
const staticList = document.querySelectorAll("li");   // NodeList (Static)

const ul = document.getElementById("myList");
const newItem = document.createElement("li");
newItem.textContent = "Item 2";
ul.appendChild(newItem);

console.log(liveList.length);  // 2 (updated)
console.log(staticList.length); // 1 (unchanged)
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Live vs Static NodeList Example</title>
</head>
<body>
  <ul id="myList">
    <li>Item 1</li>
  </ul>

  <script>
    const liveList = document.getElementsByTagName("li"); // Live HTMLCollection
    const staticList = document.querySelectorAll("li");   // Static NodeList

    const ul = document.getElementById("myList");
    const newItem = document.createElement("li");
    newItem.textContent = "Item 2";
    ul.appendChild(newItem);

    console.log("Live HTMLCollection length:", liveList.length);  // 2 (updated)
    console.log("Static NodeList length:", staticList.length);    // 1 (unchanged)
  </script>
</body>
</html>
```

2.3.3 Iterating Over Collections

NodeList (has forEach):

```
const items = document.querySelectorAll("li");
items.forEach((item) => {
  console.log(item.textContent);
});
```

HTMLCollection (no forEach use `for` loop or convert):

```
const items = document.getElementsByTagName("li");
for (let i = 0; i < items.length; i++) {
  console.log(items[i].textContent);
}
```

2.3.4 Converting to Arrays

To use array methods like `map()` or `filter()`, convert the collection:

```
// Convert NodeList or HTMLCollection to array
const itemsArray = Array.from(document.getElementsByTagName("li"));
// Or using spread syntax:
const itemsArray2 = [...document.querySelectorAll("li")];

itemsArray.map((el) => el.textContent.toUpperCase());
```

2.3.5 Key Differences Recap

Feature	**NodeList**	**HTMLCollection**
Returned by	`querySelectorAll()`	`getElementsByClassName()`, `getElementsByTagName()`
Can use `forEach()`	YES (modern browsers)	NO (must convert or use loop)
Live	NO No	YES Yes
Type of contents	Nodes (can include text nodes)	Elements only

2.3.6 Best Practices

- Use `querySelectorAll()` and NodeLists for **predictable, static** results.
- Convert collections to arrays when you want to use **array methods**.
- Be cautious with live collections — they update automatically, which can be surprising.
- When iterating over HTMLCollections, prefer `for` loops or convert first.

2.3.7 Try It Yourself

HTML:

```
<ul class="fruits">
  <li>Apple</li>
  <li>Banana</li>
  <li>Cherry</li>
</ul>
```

JavaScript:

```
const nodeList = document.querySelectorAll(".fruits li");
const htmlCollection = document.getElementsByClassName("fruits");

console.log(Array.isArray(nodeList));            // false
console.log(Array.from(nodeList).map(el => el.textContent)); // ["Apple", "Banana", "Cherry"]
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>NodeList vs HTMLCollection Example</title>
</head>
<body>
  <ul class="fruits">
    <li>Apple</li>
    <li>Banana</li>
    <li>Cherry</li>
  </ul>

  <script>
    const nodeList = document.querySelectorAll(".fruits li");
    const htmlCollection = document.getElementsByClassName("fruits");

    console.log("Is nodeList an Array?", Array.isArray(nodeList)); // false
    console.log("nodeList items:", Array.from(nodeList).map(el => el.textContent)); // ["Apple", "Banan

    console.log("HTMLCollection length:", htmlCollection.length);  // 1
  </script>
</body>
</html>
```

Understanding these collection types is essential before diving into real-world DOM manipu-

lation and dynamic UI updates. Next, we'll learn how to access **element properties and attributes** in these collections.

2.4 Accessing Element Properties and Attributes

Once you've selected elements from the DOM, the next step is often to inspect or update their properties and attributes. While these two concepts are related, they are not the same — and understanding the difference is crucial to manipulating elements effectively.

2.4.1 DOM Properties vs. HTML Attributes

Attribute	The value defined in the **HTML source**.
Property	The current value in the **DOM object**.

For example:

```
<input id="nameInput" type="text" value="Alice" />
```

```
const input = document.getElementById("nameInput");

console.log(input.getAttribute("value")); // "Alice" (original attribute)
console.log(input.value);                 // "Alice" (current DOM value)

input.value = "Bob";
console.log(input.getAttribute("value")); // still "Alice"
```

- `getAttribute()` reads from the **HTML attribute**.
- `.value` reflects the **current DOM state** (which may have changed).

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Input Attribute vs Property</title>
</head>
<body>
  <input id="nameInput" type="text" value="Alice" />

  <script>
    const input = document.getElementById("nameInput");

    console.log("getAttribute('value'):", input.getAttribute("value")); // "Alice" (original attribute)
    console.log("input.value:", input.value);                           // "Alice" (current DOM value)
```

```
    input.value = "Bob";

    console.log("After change:");
    console.log("getAttribute('value'):", input.getAttribute("value")); // still "Alice"
    console.log("input.value:", input.value);                           // "Bob"
  </script>
</body>
</html>
```

2.4.2 Accessing and Modifying Common Properties

You can read and write many properties directly using dot notation:

`id` and `className`

```
const element = document.getElementById("myDiv");

console.log(element.id);          // "myDiv"
console.log(element.className);   // "highlight"

element.className = "updated";    // Updates class on the element
```

`value` (for form elements)

```
const input = document.querySelector("input");

console.log(input.value);         // Current value of the input
input.value = "New text";         // Updates input field on the page
```

`textContent` and `innerHTML`

```
const para = document.querySelector("p");

console.log(para.textContent);    // Text inside paragraph
para.textContent = "Updated text!";

para.innerHTML = "<strong>Bold!</strong>";  // Replaces content with bold text
```

> YES `textContent` is safer and faster when you're only working with plain text.
> WARNING Use `innerHTML` carefully — it can expose your page to XSS vulnerabilities if used with untrusted input.

2.4.3 Working with Attributes

Sometimes you need to read or change **HTML attributes**, especially non-standard ones or `data-*` attributes.

Reading and Writing Attributes

```
const link = document.querySelector("a");

// Read attributes
console.log(link.getAttribute("href"));     // "/home"
console.log(link.getAttribute("target"));   // "_blank"

// Write attributes
link.setAttribute("href", "/about");
link.setAttribute("title", "Go to About Page");
```

Removing Attributes

```
link.removeAttribute("target");
```

2.4.4 Accessing `data-*` Attributes

Custom data attributes (like `data-user-id`) are accessed using `.dataset`:

```
<div id="user" data-user-id="42" data-role="admin"></div>
```

```
const userDiv = document.getElementById("user");

console.log(userDiv.dataset.userId); // "42"
console.log(userDiv.dataset.role);   // "admin"

userDiv.dataset.role = "editor";     // Updates the data-role attribute
```

> YES `dataset` provides a clean way to work with custom data embedded in HTML.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Custom Data Attributes Example</title>
</head>
<body>
  <div id="user" data-user-id="42" data-role="admin"></div>

  <script>
```

```
    const userDiv = document.getElementById("user");

    console.log(userDiv.dataset.userId); // "42"
    console.log(userDiv.dataset.role);   // "admin"

    userDiv.dataset.role = "editor";     // Updates the data-role attribute

    // Verify the update
    console.log(userDiv.getAttribute("data-role")); // "editor"
  </script>
</body>
</html>
```

2.4.5 Common Pitfalls and Gotchas

Mistake	Why It Happens
Using `.getAttribute("value")` expecting the updated input value	It returns the original HTML attribute, not the live input state. Use `.value` instead.
Assuming `className` works like `classList`	`className` replaces the whole string. Use `.classList.add()`/`.remove()` for precise control.
Mixing up attribute vs. property behavior	HTML attributes set initial property values but don't stay in sync afterward.
Modifying `innerHTML` carelessly	Can introduce security issues or disrupt existing event listeners. Prefer DOM methods when possible.

2.4.6 Best Practices

- Use **property access** (e.g., `.value`, `.textContent`) for live, dynamic interactions.
- Use **attribute methods** (`getAttribute`, `setAttribute`) for static or non-standard attributes.
- Prefer `.dataset` for working with custom `data-*` values.
- Avoid overusing `innerHTML`; favor `textContent`, `appendChild()`, or `createElement()` for safety.

2.4.7 Try It Yourself

HTML:

```
<input id="email" type="email" value="user@example.com" data-label="User Email" />
```

JavaScript:

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

console.log(emailInput.value);                 // "user@example.com"
console.log(emailInput.getAttribute("value")); // "user@example.com"
console.log(emailInput.dataset.label);          // "User Email"

emailInput.value = "new@example.com";
console.log(emailInput.getAttribute("value")); // still "user@example.com"
```

Understanding how to read and modify properties and attributes gives you powerful control over web page behavior. Up next, we'll explore **real-world examples** of selecting and accessing elements in different ways.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Input Attributes Example</title>
</head>
<body>
  <input id="email" type="email" value="user@example.com" data-label="User Email" />

  <script>
    const emailInput = document.getElementById("email");

    console.log("Initial value (DOM):", emailInput.value);                    // "user@example.com"
    console.log("Initial value (attribute):", emailInput.getAttribute("value")); // "user@example.com"
    console.log("Data label:", emailInput.dataset.label);                      // "User Email"

    emailInput.value = "new@example.com";

    console.log("After change:");
    console.log("Value (DOM):", emailInput.value);                             // "new@example.com"
    console.log("Value (attribute):", emailInput.getAttribute("value"));       // still "user@example.
  </script>
</body>
</html>
```

2.5 Practical Examples: Selecting Elements in Various Ways

Now that you've learned how to select and access elements in the DOM using both basic and modern methods, let's apply that knowledge with some hands-on examples. These scenarios reflect common web development tasks like handling forms, menus, or highlighted content. You'll see how to **mix selectors**, **loop through elements**, and **choose the right method for clarity and performance**.

2.5.1 Example 1: Selecting a Form Input by ID

```
<input id="username" type="text" value="guest" />
```

```
const usernameInput = document.getElementById("username");
console.log(usernameInput.value); // "guest"
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Input Value Example</title>
</head>
<body>
  <input id="username" type="text" value="guest" />

  <script>
    const usernameInput = document.getElementById("username");
    console.log(usernameInput.value); // "guest"
  </script>
</body>
</html>
```

Why `getElementById()`? It's fast and direct — perfect for a unique element like a form field with a known `id`.

2.5.2 Example 2: Selecting All Items with a Class

```
<ul>
  <li class="menu-item">Home</li>
  <li class="menu-item">About</li>
  <li class="menu-item">Contact</li>
</ul>
```

```
const items = document.getElementsByClassName("menu-item");

for (let item of items) {
  console.log(item.textContent); // Logs each menu item's text
}
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Menu Items Logger</title>
</head>
<body>
  <ul>
```

```
    <li class="menu-item">Home</li>
    <li class="menu-item">About</li>
    <li class="menu-item">Contact</li>
  </ul>

  <script>
    const items = document.getElementsByClassName("menu-item");

    for (let item of items) {
      console.log(item.textContent); // Logs each menu item's text
    }
  </script>
</body>
</html>
```

Note: This returns an **HTMLCollection**, which is *live* and **not a true array**.

2.5.3 Example 3: Using `querySelectorAll()` with CSS Selectors

```
<div>
  <p class="highlight">This is important.</p>
  <p>This is normal.</p>
  <p class="highlight">So is this.</p>
</div>
```

```
const highlighted = document.querySelectorAll("p.highlight");

highlighted.forEach(p => {
  p.style.backgroundColor = "yellow";
});
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Highlight Example</title>
</head>
<body>
  <div>
    <p class="highlight">This is important.</p>
    <p>This is normal.</p>
    <p class="highlight">So is this.</p>
  </div>

  <script>
    const highlighted = document.querySelectorAll("p.highlight");

    highlighted.forEach(p => {
      p.style.backgroundColor = "yellow";
    });
  </script>
```

```
</body>
</html>
```

Why `querySelectorAll()`? It allows complex CSS selectors like `p.highlight`, `div > p`, or `[data-*]`.

2.5.4 Example 4: Selecting Form Inputs by Tag

```
<form>
  <input type="text" name="firstName" />
  <input type="text" name="lastName" />
</form>
```

```
const inputs = document.getElementsByTagName("input");

Array.from(inputs).forEach(input => {
  console.log(input.name); // "firstName", then "lastName"
});
```

Tip: Convert the `HTMLCollection` to an array with `Array.from()` for full array method support.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Form Input Names</title>
</head>
<body>
  <form>
    <input type="text" name="firstName" />
    <input type="text" name="lastName" />
  </form>

  <script>
    const inputs = document.getElementsByTagName("input");

    Array.from(inputs).forEach(input => {
      console.log(input.name); // "firstName", then "lastName"
    });
  </script>
</body>
</html>
```

2.5.5 Example 5: Mixing Selectors to Narrow Down

```
<div class="card featured">...</div>
<div class="card">...</div>
```

```
const featuredCard = document.querySelector(".card.featured");
console.log(JSON.stringify(featuredCard,null,2)); // The div with both 'card' and 'featured' classes
```

Benefit: Clear, concise selector that expresses both class conditions.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>QuerySelector Multi-Class Example</title>
  <style>
    .card {
      padding: 10px;
      border: 1px solid #ccc;
      margin-bottom: 10px;
    }
    .featured {
      background-color: #f9f871;
    }
  </style>
</head>
<body>
  <div class="card featured">This is the featured card.</div>
  <div class="card">This is a regular card.</div>

  <script>
    const featuredCard = document.querySelector(".card.featured");
    console.log(featuredCard); // The div with both 'card' and 'featured' classes
  </script>
</body>
</html>
```

2.5.6 Example 6: Selecting Custom Attributes

```
<button data-action="save">Save</button>
<button data-action="delete">Delete</button>
```

```
const buttons = document.querySelectorAll("button[data-action]");

buttons.forEach(btn => {
  console.log(btn.dataset.action); // "save", then "delete"
});
```

Why it's useful: You can filter based on attribute presence or value ([data-

action="save"]).

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Data Action Buttons</title>
</head>
<body>
  <button data-action="save">Save</button>
  <button data-action="delete">Delete</button>

  <script>
    const buttons = document.querySelectorAll("button[data-action]");

    buttons.forEach(btn => {
      console.log(btn.dataset.action); // "save", then "delete"
    });
  </script>
</body>
</html>
```

2.5.7 Example 7: Efficient Targeting with Descendant Selectors

```
<div class="form-section">
  <label>Email: <input type="email" /></label>
</div>
```

```
const emailInput = document.querySelector(".form-section input[type='email']");
```

Why it matters: Combining selectors avoids extra traversal and clarifies intent.

2.5.8 Choosing the Right Selector

Selector Method	When to Use
`getElementById()`	Fastest, for unique elements with known IDs
`getElementsByClassName()`	For multiple elements sharing a class (live collection)
`getElementsByTagName()`	For gathering elements by tag (live collection)
`querySelector()`	For selecting a single element using a CSS selector
`querySelectorAll()`	For selecting multiple elements using a CSS selector (static list)

2.5.9 Summary & Best Practices

- **Use `querySelector`/`querySelectorAll`** for most modern needs thanks to their flexibility.
- **Loop over collections** using `forEach()` (after converting if needed).
- **Keep performance in mind**: `getElementById()` is fastest for single-item lookups.
- **Combine selectors** to be precise, clear, and expressive.

2.5.10 Try It Yourself

Given this HTML:

```
<ul id="fruits">
  <li class="favorite">Apple</li>
  <li>Banana</li>
  <li class="favorite">Cherry</li>
</ul>
```

Can you select only the list items with class `favorite`?

```
const favs = document.querySelectorAll("#fruits li.favorite");
favs.forEach(item => console.log(item.textContent)); // "Apple", "Cherry"
```

These examples give you the power to select, inspect, and modify any part of your document dynamically and efficiently — a crucial first step in DOM programming mastery.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Select Favorite Fruits</title>
</head>
<body>
  <ul id="fruits">
    <li class="favorite">Apple</li>
    <li>Banana</li>
    <li class="favorite">Cherry</li>
  </ul>

  <script>
    const favs = document.querySelectorAll("#fruits li.favorite");
    favs.forEach(item => console.log(item.textContent)); // "Apple", "Cherry"
  </script>
</body>
</html>
```

Chapter 3.

Manipulating DOM Elements

1. Changing Text and HTML Content (`textContent`, `innerHTML`)
2. Modifying Attributes and Styles
3. Working with Classes (`classList` API)
4. Creating, Cloning, and Removing Elements
5. Practical Example: Building a Dynamic List

3 Manipulating DOM Elements

3.1 Changing Text and HTML Content (`textContent`, `innerHTML`)

When working with the DOM, two of the most common ways to modify what's displayed inside an element are through the properties `textContent` and `innerHTML`. Although both let you update the content, they behave differently and serve distinct purposes. Understanding these differences is crucial for writing secure, efficient, and maintainable code.

What is `textContent`?

- `textContent` represents the **text inside an element**, including its descendants, but **ignores any HTML tags**.
- When you set `textContent`, it **replaces all existing content** inside the element with plain text.
- It **escapes any HTML tags**, displaying them as literal text rather than rendering HTML.

```
<div id="message">Hello, <strong>world!</strong></div>

<script>
  const messageDiv = document.getElementById("message");

  // Change the text inside the div
  messageDiv.textContent = "Welcome to JavaScript DOM Programming!";

  // Result: The <strong> tag is removed, and plain text replaces all content
</script>
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>textContent Example</title>
</head>
<body>
  <div id="message">Hello, <strong>world!</strong></div>

  <script>
    const messageDiv = document.getElementById("message");

    // Change the text inside the div
    messageDiv.textContent = "Welcome to JavaScript DOM Programming!";

    // After this, the <strong> element is removed,
    // and the div contains plain text only
  </script>
</body>
</html>
```

Example: Updating text with `textContent` Result on page:

Welcome to JavaScript DOM Programming!

What is `innerHTML`?

- `innerHTML` contains the **HTML markup inside an element** as a string.
- Setting `innerHTML` parses the string as HTML, **rendering any tags or elements included**.
- It allows injecting **complex HTML structures dynamically**.

```
<div id="container"></div>

<script>
  const container = document.getElementById("container");

  // Inject a new HTML structure
  container.innerHTML = "<p>This is a <em>dynamic</em> paragraph.</p>";

  // The paragraph and emphasis will be rendered properly
</script>
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>innerHTML Injection Example</title>
</head>
<body>
  <div id="container"></div>

  <script>
    const container = document.getElementById("container");

    // Inject a new HTML structure
    container.innerHTML = "<p>This is a <em>dynamic</em> paragraph.</p>";

    // The paragraph and emphasis will be rendered properly
  </script>
</body>
</html>
```

Example: Injecting HTML with `innerHTML` Result on page:

This is a *dynamic* paragraph.

Security Considerations: Beware of `innerHTML` with Untrusted Input

While `innerHTML` is powerful, **it poses a security risk when used with untrusted or user-supplied data**. Because it parses and renders HTML, malicious scripts could be injected, leading to **Cross-Site Scripting (XSS)** attacks.

Example of a security risk:

```
const userInput = '<img src=x onerror="alert(\'XSS Attack\')">';
container.innerHTML = userInput; // Dangerous!
```

How to stay safe:

- Prefer `textContent` when displaying user input as plain text.
- Sanitize input thoroughly before inserting it via `innerHTML`.
- Avoid mixing user input directly into HTML strings.

When to Use `textContent` vs. `innerHTML`

Use Case	Recommended Property	Reason
Display or update **plain text**	`textContent`	Safer and faster; no HTML parsing involved
Insert **HTML markup**	`innerHTML`	Allows injecting tags and rich content
Display **user input or untrusted data**	`textContent`	Prevents XSS by escaping HTML characters

Summary Examples

```
<div id="demo"></div>

<script>
  const demo = document.getElementById("demo");

  // Set plain text safely
  demo.textContent = "<strong>This will not be bold</strong>";

  // Set HTML content dynamically
  demo.innerHTML = "<strong>This will be bold</strong>";
</script>
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>textContent vs innerHTML</title>
</head>
<body>
  <div id="demo"></div>

  <script>
    const demo = document.getElementById("demo");

    // Set plain text safely (HTML tags show as text)
    demo.textContent = "<strong>This will not be bold</strong>";
```

```
    // After 3 seconds, set HTML content dynamically (tags parsed)
    setTimeout(() => {
      demo.innerHTML = "<strong>This will be bold</strong>";
    }, 3000);
  </script>
</body>
</html>
```

3.1.1 Key Takeaways

- Use `textContent` for safe, plain text updates.
- Use `innerHTML` for rich content but sanitize input to avoid security risks.
- Both properties overwrite existing content, so choose the one fitting your needs.
- Understanding their differences improves your code's **security**, **performance**, and **maintainability**.

Try experimenting by changing text and HTML in your own page! Notice how `textContent` treats tags as text, while `innerHTML` parses and renders them. This fundamental knowledge is essential for effective DOM manipulation.

3.2 Modifying Attributes and Styles

Manipulating HTML attributes and styles is a fundamental part of making web pages dynamic and interactive. In this section, we'll explore how to read, add, modify, and remove attributes using DOM methods and properties, as well as how to dynamically change styles both inline and via CSS classes.

Working with Attributes

HTML attributes define additional information about elements, such as `id`, `href`, `src`, `alt`, and custom `data-*` attributes.

Reading Attributes You can read attributes in two main ways:

- Using the **property** of the DOM element, e.g., `element.id`, `element.href`.
- Using the **`getAttribute()`** method, which returns the exact value of the attribute in the HTML.

```
<a id="link" href="https://example.com" target="_blank">Visit Example</a>

<script>
  const link = document.getElementById("link");
```

```
  console.log(link.href); // "https://example.com/"
  console.log(link.getAttribute("href")); // "https://example.com"
</script>
```

Note: Properties like `href` may return fully resolved URLs, while `getAttribute()` returns exactly what is in the HTML.

Adding or Modifying Attributes You can set attributes using either:

- Direct property assignment: `element.href = "new-url.html";`
- The `setAttribute()` method: `element.setAttribute("href", "new-url.html");`

Example:

```
link.href = "https://openai.com";
link.setAttribute("target", "_self"); // Changes target attribute
```

Removing Attributes Use the `removeAttribute()` method to remove an attribute:

```
link.removeAttribute("target");
```

This removes the `target` attribute, so the link will open in the current tab by default.

Working with Custom Data Attributes Custom data attributes are prefixed with `data-`, such as `data-user-id="1234"`. You can access them via the `dataset` property:

```
<div id="user" data-user-id="1234" data-role="admin"></div>

<script>
  const userDiv = document.getElementById("user");

  // Read custom data attributes
  console.log(userDiv.dataset.userId); // "1234"
  console.log(userDiv.dataset.role);   // "admin"

  // Modify a data attribute
  userDiv.dataset.role = "editor";
</script>
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Custom Data Attributes Demo</title>
</head>
<body>
  <div id="user" data-user-id="1234" data-role="admin"></div>

  <script>
    const userDiv = document.getElementById("user");
```

```
    // Read custom data attributes
    console.log(userDiv.dataset.userId); // "1234"
    console.log(userDiv.dataset.role);   // "admin"

    // Modify a data attribute
    userDiv.dataset.role = "editor";

    // Confirm the change
    console.log(userDiv.getAttribute("data-role")); // "editor"
  </script>
</body>
</html>
```

Modifying Styles

There are two common ways to change element styles dynamically:

1. **Using the `style` property for inline styles**
2. **Using CSS classes**

Changing Inline Styles via `style` Property The `style` property allows you to change individual CSS properties directly on the element:

```
<div id="box" style="width: 100px; height: 100px; background-color: red;"></div>

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

  // Change background color
  box.style.backgroundColor = "blue";

  // Toggle visibility
  box.style.display = "none"; // Hide
  box.style.display = "block"; // Show
</script>
```

Note: Inline styles override styles defined in external CSS, so use them carefully to avoid specificity issues.

Toggling Styles with CSS Classes A better practice for maintaining separation of concerns is to use CSS classes and toggle them via JavaScript.

Example CSS:

```
.hidden {
  display: none;
}

.highlight {
  background-color: yellow;
}
```

JavaScript to toggle classes:

```
box.classList.add("highlight");    // Add a class
box.classList.remove("highlight"); // Remove a class
box.classList.toggle("hidden");    // Toggle a class on/off
```

Using CSS classes keeps style definitions centralized and makes it easier to maintain and update styles.

Practical Example: Toggling Visibility and Color

```
<button id="toggleBtn">Toggle Box</button>
<div id="colorBox" style="width: 100px; height: 100px; background-color: green;"></div>

<script>
  const toggleBtn = document.getElementById("toggleBtn");
  const colorBox = document.getElementById("colorBox");

  toggleBtn.addEventListener("click", () => {
    // Toggle visibility using inline style
    if (colorBox.style.display === "none") {
      colorBox.style.display = "block";
    } else {
      colorBox.style.display = "none";
    }

    // Toggle highlight class to change background color
    colorBox.classList.toggle("highlight");
  });
</script>
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Toggle Box Example</title>
  <style>
    #colorBox {
      width: 100px;
      height: 100px;
      background-color: green;
      transition: background-color 0.3s ease;
    }
    .highlight {
      background-color: orange !important;
    }
  </style>
</head>
<body>
  <button id="toggleBtn">Toggle Box</button>
  <div id="colorBox"></div>

  <script>
    const toggleBtn = document.getElementById("toggleBtn");
    const colorBox = document.getElementById("colorBox");

    toggleBtn.addEventListener("click", () => {
```

```
      // Toggle visibility using inline style
      if (colorBox.style.display === "none") {
        colorBox.style.display = "block";
      } else {
        colorBox.style.display = "none";
      }

      // Toggle highlight class to change background color
      colorBox.classList.toggle("highlight");
    });
  </script>
</body>
</html>
```

3.2.1 Best Practices

- **Use properties for standard attributes** when possible (e.g., `element.href`), but `getAttribute()` and `setAttribute()` are useful for custom or non-standard attributes.
- **Prefer CSS classes over inline styles** to keep your styles organized and avoid inline clutter.
- **Avoid mixing content and presentation**; keep HTML for structure/content and CSS for styling.
- Use the `classList` API to add, remove, or toggle classes efficiently.
- Be mindful of specificity and inheritance when dynamically applying styles.

By mastering attribute and style manipulation, you gain powerful tools to dynamically change the appearance and behavior of your web pages, while keeping your code clean and maintainable.

3.3 Working with Classes (`classList` API)

Managing CSS classes on DOM elements is a common task when creating interactive web pages. The modern and recommended way to handle this is with the **`classList`** API, which provides a clean, efficient, and readable interface to add, remove, toggle, and check classes on elements.

Why Use `classList`?

Before `classList`, developers manipulated the `className` string directly:

```
element.className = "foo bar";
```

But modifying this string manually is error-prone and cumbersome, especially when adding or removing a single class without affecting others.

The `classList` API solves this by providing methods specifically designed for class manipulation without you needing to parse or rebuild the class string.

Key Methods of `classList`

Method	Description
`add(className)`	Adds a class if it doesn't already exist.
`remove(className)`	Removes a class if it exists.
`toggle(className)`	Adds the class if missing, removes if present.
`contains(className)`	Checks if the element has the specified class.

Basic Usage Examples

Suppose we have the following HTML element:

```
<div id="box" class="box"></div>
```

JavaScript manipulation:

```
const box = document.getElementById("box");

// Add a class
box.classList.add("highlighted");

// Remove a class
box.classList.remove("box");

// Toggle a class
box.classList.toggle("active");

// Check if class exists
if (box.classList.contains("highlighted")) {
  console.log("Box is highlighted");
}
```

Practical Use Cases

```
<button id="menuBtn">Toggle Menu</button>
<nav id="menu" class="menu hidden">...</nav>
```

```
.hidden {
  display: none;
}
```

```
const menuBtn = document.getElementById("menuBtn");
const menu = document.getElementById("menu");
```

```
menuBtn.addEventListener("click", () => {
  menu.classList.toggle("hidden");
});
```

Responsive Menu Toggle Clicking the button toggles the menu visibility by adding or removing the `.hidden` class.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Toggle Menu Example</title>
  <style>
    .hidden {
      display: none;
    }
    .menu {
      border: 1px solid #333;
      padding: 10px;
      width: 200px;
      background-color: #eee;
      margin-top: 10px;
    }
  </style>
</head>
<body>

<button id="menuBtn">Toggle Menu</button>
<nav id="menu" class="menu hidden">
  <ul>
    <li><a href="#">Home</a></li>
    <li><a href="#">About</a></li>
    <li><a href="#">Contact</a></li>
  </ul>
</nav>

<script>
  const menuBtn = document.getElementById("menuBtn");
  const menu = document.getElementById("menu");

  menuBtn.addEventListener("click", () => {
    menu.classList.toggle("hidden");
  });
</script>

</body>
</html>
```

```
<input id="email" type="email" />
```

```
.invalid {
  border: 2px solid red;
```

```
}
```

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

emailInput.addEventListener("input", () => {
  if (!emailInput.value.includes("@")) {
    emailInput.classList.add("invalid");
  } else {
    emailInput.classList.remove("invalid");
  }
});
```

Form Validation Highlight This dynamically adds or removes a validation highlight based on user input.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Email Validation Example</title>
  <style>
    .invalid {
      border: 2px solid red;
    }
  </style>
</head>
<body>
  <input id="email" type="email" placeholder="Enter your email" />

  <script>
    const emailInput = document.getElementById("email");

    emailInput.addEventListener("input", () => {
      if (!emailInput.value.includes("@")) {
        emailInput.classList.add("invalid");
      } else {
        emailInput.classList.remove("invalid");
      }
    });
  </script>
</body>
</html>
```

3.3.1 Summary

- The `classList` API offers a simple and safe way to manipulate element classes.
- It prevents common bugs associated with manually editing the `className` string.
- Methods like `add`, `remove`, `toggle`, and `contains` enable flexible and readable code.
- Practical for UI features such as toggling visibility, applying styles, and validation feedback.

By embracing `classList`, your DOM manipulation code becomes more maintainable and easier to understand.

3.4 Creating, Cloning, and Removing Elements

Manipulating the DOM often requires creating new elements, duplicating existing ones, or removing elements from the page. In this section, we'll explore how to do these tasks efficiently and cleanly using standard DOM methods.

Creating New Elements with `document.createElement()`

To add new elements dynamically, you first create them using:

```
const newElement = document.createElement("tagName");
```

For example, to create a new list item:

```
const li = document.createElement("li");
li.textContent = "New item";
document.querySelector("ul").appendChild(li);
```

This creates a new `<li>`, sets its text content, and appends it to an existing `<ul>`.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Create Element Example</title>
</head>
<body>
  <ul>
    <li>Existing item</li>
  </ul>

  <script>
    const li = document.createElement("li");
    li.textContent = "New item";
    document.querySelector("ul").appendChild(li);
  </script>
</body>
</html>
```

Cloning Nodes with `cloneNode()`

Sometimes you want to duplicate an existing element, including or excluding its child elements.

- **Shallow clone (`cloneNode(false)`):** Copies the element only, without children.
- **Deep clone (`cloneNode(true)`):** Copies the element and all descendants recursively.

Example:

```
const original = document.getElementById("template");
const shallowClone = original.cloneNode(false); // no children
const deepClone = original.cloneNode(true); // with children

document.body.appendChild(deepClone);
```

Cloning is useful when you have a template element you want to reuse multiple times with the same structure.

Removing Elements from the DOM

The modern and straightforward way to remove an element is:

```
element.remove();
```

This removes the element from its parent directly.

Older method (widely supported):

```
element.parentNode.removeChild(element);
```

Example:

```
const item = document.querySelector(".old-item");
item.remove(); // modern way
// or
item.parentNode.removeChild(item); // older way
```

Practical Example: Adding and Removing List Items

Here's a simple example that creates a dynamic list where you can add and remove items:

```
<ul id="todo-list"></ul>
<button id="add-btn">Add Item</button>
```

```
const list = document.getElementById("todo-list");
const addBtn = document.getElementById("add-btn");

addBtn.addEventListener("click", () => {
  const newItem = document.createElement("li");
  newItem.textContent = "Task " + (list.children.length + 1);

  // Add a remove button to each item
  const removeBtn = document.createElement("button");
  removeBtn.textContent = "Remove";
  removeBtn.addEventListener("click", () => {
    newItem.remove();
  });

  newItem.appendChild(removeBtn);
  list.appendChild(newItem);
});
```

Each click adds a new list item with a remove button that deletes the item when clicked.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Todo List</title>
  <style>
    li {
      margin-bottom: 8px;
    }
    button {
      margin-left: 10px;
    }
  </style>
</head>
<body>
  <ul id="todo-list"></ul>
  <button id="add-btn">Add Item</button>

  <script>
    const list = document.getElementById("todo-list");
    const addBtn = document.getElementById("add-btn");

    addBtn.addEventListener("click", () => {
      const newItem = document.createElement("li");
      newItem.textContent = "Task " + (list.children.length + 1);

      // Add a remove button to each item
      const removeBtn = document.createElement("button");
      removeBtn.textContent = "Remove";
      removeBtn.addEventListener("click", () => {
        newItem.remove();
      });

      newItem.appendChild(removeBtn);
      list.appendChild(newItem);
    });
  </script>
</body>
</html>
```

Best Practices for Clean DOM Manipulation

- **Batch DOM updates:** Minimize reflows by building elements in memory before appending.
- **Use deep cloning for templates:** Avoid rebuilding complex structures manually.
- **Remove unused elements promptly:** Helps with memory management and keeps UI responsive.
- **Avoid unnecessary mutations:** Keep your DOM manipulation predictable and maintainable.

3.4.1 Summary

- Use `document.createElement()` to create new elements.
- Clone nodes with `cloneNode()`, choosing between shallow and deep copies.
- Remove elements cleanly with `remove()` or the older `parentNode.removeChild()` method.
- Proper DOM manipulation improves performance, memory usage, and code clarity.

This knowledge lays the groundwork for creating dynamic, interactive web pages in a performant and maintainable way.

3.5 Practical Example: Building a Dynamic List

In this section, we'll create a complete, interactive example that dynamically builds and updates a list based on user input. This example will bring together many DOM manipulation techniques you've learned so far—element creation, class manipulation, event handling, and DOM insertion—to create a simple todo list.

The Goal

Build a dynamic todo list where users can:

- Add new items by typing text and clicking a button
- Mark items as completed by clicking on them (toggles a class)
- Remove items with a button click

HTML Setup

We start with a simple HTML structure:

```
<div id="todo-app">
  <input type="text" id="todo-input" placeholder="Enter a new task" />
  <button id="add-btn">Add</button>
  <ul id="todo-list"></ul>
</div>
```

JavaScript Code

```
// Select relevant DOM elements
const input = document.getElementById("todo-input");
const addBtn = document.getElementById("add-btn");
const list = document.getElementById("todo-list");

// Function to create a new todo list item
function createTodoItem(text) {
  // Create <li> element
  const li = document.createElement("li");
  li.textContent = text;
```

```
  li.classList.add("todo-item");

  // Toggle completed class on click
  li.addEventListener("click", () => {
    li.classList.toggle("completed");
  });

  // Create Remove button
  const removeBtn = document.createElement("button");
  removeBtn.textContent = "Remove";
  removeBtn.classList.add("remove-btn");

  // Remove the list item when remove button is clicked
  removeBtn.addEventListener("click", (event) => {
    event.stopPropagation(); // Prevent triggering the toggle
    li.remove();
  });

  // Append remove button to list item
  li.appendChild(removeBtn);

  return li;
}

// Event listener for Add button
addBtn.addEventListener("click", () => {
  const text = input.value.trim();

  if (text === "") {
    console.log("Please enter a task.");
    return;
  }

  const todoItem = createTodoItem(text);
  list.appendChild(todoItem);

  // Clear input and focus
  input.value = "";
  input.focus();
});

// Optional: Add todo with Enter key
input.addEventListener("keydown", (event) => {
  if (event.key === "Enter") {
    addBtn.click();
  }
});
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Todo App</title>
  <style>
    #todo-app {
      max-width: 400px;
      margin: 2em auto;
```

```
      font-family: Arial, sans-serif;
    }
    #todo-input {
      width: 70%;
      padding: 0.5em;
      font-size: 1em;
    }
    #add-btn {
      padding: 0.5em 1em;
      font-size: 1em;
      margin-left: 0.5em;
    }
    #todo-list {
      list-style: none;
      padding-left: 0;
      margin-top: 1em;
    }
    .todo-item {
      background: #f9f9f9;
      margin-bottom: 0.5em;
      padding: 0.5em;
      cursor: pointer;
      display: flex;
      justify-content: space-between;
      align-items: center;
      border-radius: 4px;
      transition: background-color 0.3s;
    }
    .todo-item.completed {
      text-decoration: line-through;
      color: #888;
      background: #d3ffd3;
    }
    .remove-btn {
      background: #ff4d4d;
      border: none;
      color: white;
      padding: 0.3em 0.6em;
      border-radius: 3px;
      cursor: pointer;
      font-size: 0.9em;
      transition: background-color 0.2s;
    }
    .remove-btn:hover {
      background: #ff1a1a;
    }
  </style>
</head>
<body>

<div id="todo-app">
  <input type="text" id="todo-input" placeholder="Enter a new task" />
  <button id="add-btn">Add</button>
  <ul id="todo-list"></ul>
</div>

<script>
  // Select relevant DOM elements
```

```
  const input = document.getElementById("todo-input");
  const addBtn = document.getElementById("add-btn");
  const list = document.getElementById("todo-list");

  // Function to create a new todo list item
  function createTodoItem(text) {
    // Create <li> element
    const li = document.createElement("li");
    li.textContent = text;
    li.classList.add("todo-item");

    // Toggle completed class on click
    li.addEventListener("click", () => {
      li.classList.toggle("completed");
    });

    // Create Remove button
    const removeBtn = document.createElement("button");
    removeBtn.textContent = "Remove";
    removeBtn.classList.add("remove-btn");

    // Remove the list item when remove button is clicked
    removeBtn.addEventListener("click", (event) => {
      event.stopPropagation(); // Prevent triggering the toggle
      li.remove();
    });

    // Append remove button to list item
    li.appendChild(removeBtn);

    return li;
  }

  // Event listener for Add button
  addBtn.addEventListener("click", () => {
    const text = input.value.trim();

    if (text === "") {
      console.log("Please enter a task.");
      return;
    }

    const todoItem = createTodoItem(text);
    list.appendChild(todoItem);

    // Clear input and focus
    input.value = "";
    input.focus();
  });

  // Optional: Add todo with Enter key
  input.addEventListener("keydown", (event) => {
    if (event.key === "Enter") {
      addBtn.click();
    }
  });
</script>
```

```
</body>
</html>
```

Explanation Step-by-Step

1. **Select DOM elements:** Grab references to the input field, add button, and the list container using `getElementById()`.
2. **Create todo items dynamically:** The `createTodoItem()` function builds an `<li>` element with the task text and a remove button. It adds classes for styling and event listeners for toggling completed status and removing the item.
3. **Add event handlers:** Clicking the "Add" button or pressing Enter in the input adds a new todo item to the list. We trim input to avoid empty entries and reset the input field afterward.
4. **Toggle and remove logic:** Clicking a list item toggles the `"completed"` class, which you can style with CSS to indicate completion. The remove button deletes the item without toggling the completed state (using `stopPropagation`).

Encouragement to Experiment

- Try adding editing functionality by making list items editable on double-click.
- Enhance UI by adding styles for `.completed` and `.remove-btn` classes.
- Implement persistence by saving and loading todos from `localStorage`.
- Add a clear-all button or filter tasks (all/active/completed).

How This Example Ties Everything Together

- **Element creation:** Using `createElement()` for list items and buttons.
- **Class manipulation:** Using `classList.add()` and `classList.toggle()` to change appearance.
- **Event handling:** Using `addEventListener()` for user interactions.
- **DOM insertion and removal:** Adding items with `appendChild()` and removing with `remove()`.

This example showcases practical, real-world usage of DOM manipulation techniques, helping you build interactive web apps in a clean, modular way.

3.5.1 Summary

You now have a functional, dynamic todo list app built purely with JavaScript and DOM APIs. This demonstrates how to combine multiple DOM techniques to create rich, responsive user interfaces, setting the foundation for more complex interactive web applications.

Chapter 4.

Traversing the DOM Tree

1. Parent, Child, and Sibling Relationships
2. Navigating with `parentNode`, `children`, `nextSibling`, and `previousSibling`
3. Recursive DOM Traversal Techniques
4. Practical Example: Highlighting Related Elements

4 Traversing the DOM Tree

4.1 Parent, Child, and Sibling Relationships

The Document Object Model (DOM) represents an HTML document as a hierarchical tree of nodes. Understanding the relationships between these nodes — specifically parent, child, and sibling relationships — is fundamental for navigating and manipulating the DOM effectively.

The Hierarchical Structure of the DOM

Imagine the DOM as a family tree:

- **Parent nodes** are like the parents who have one or more children.
- **Child nodes** are the direct descendants of a parent.
- **Sibling nodes** share the same parent, like brothers and sisters.

This hierarchical organization means every node (element, text, comment) except the root has exactly one parent and can have multiple children and siblings.

Visualizing the Relationships

Consider this simple HTML snippet:

```
<ul id="fruits">
  <li>Apple</li>
  <li>Banana</li>
  <li>Cherry</li>
</ul>
```

The DOM tree looks like this:

```
<ul id="fruits">       <-- Parent node
 +- <li>Apple</li>     <-- Child node, first child
 +- <li>Banana</li>    <-- Child node, sibling of Apple
 +- <li>Cherry</li>    <-- Child node, sibling of Banana
```

- The `<ul>` is the **parent** of the three `<li>` elements.
- The `<li>` elements are **children** of the `<ul>`.
- The `<li>` elements are **siblings** to each other.

Accessing Parent, Child, and Sibling Nodes in JavaScript

You can use the following properties to navigate the DOM relationships:

Relationship	Property	Description
Parent node	`parentNode`	Returns the parent node of the current node
Child nodes	`children`	Returns an HTMLCollection of child elements
First child	`firstElementChild`	Returns the first child element

Relationship	Property	Description
Last child	`lastElementChild`	Returns the last child element
Next sibling	`nextElementSibling`	Returns the next sibling element
Previous sibling	`previousElementSibling`	Returns the previous sibling element

Simple Examples

```
const fruitsList = document.getElementById('fruits');

// Accessing parent node
console.log(fruitsList.parentNode); // Logs the parent of <ul>, typically <body> or other container

// Accessing child nodes
console.log(fruitsList.children);   // HTMLCollection of all <li> elements inside <ul>
console.log(fruitsList.firstElementChild.textContent); // "Apple"
console.log(fruitsList.lastElementChild.textContent);  // "Cherry"

// Accessing siblings
const firstFruit = fruitsList.firstElementChild;
console.log(firstFruit.nextElementSibling.textContent);      // "Banana"
console.log(firstFruit.nextElementSibling.nextElementSibling.textContent); // "Cherry"
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>DOM Traversal Example</title>
</head>
<body>
  <ul id="fruits">
    <li>Apple</li>
    <li>Banana</li>
    <li>Cherry</li>
  </ul>

  <script>
    const fruitsList = document.getElementById('fruits');

    // Accessing parent node
    console.log("Parent node:", fruitsList.parentNode); // Usually <body>

    // Accessing child nodes
    console.log("Children:", fruitsList.children); // HTMLCollection of <li> elements
    console.log("First child text:", fruitsList.firstElementChild.textContent); // "Apple"
    console.log("Last child text:", fruitsList.lastElementChild.textContent);   // "Cherry"

    // Accessing siblings
    const firstFruit = fruitsList.firstElementChild;
    console.log("Next sibling of first fruit:", firstFruit.nextElementSibling.textContent);      // "Ba
    console.log("Next sibling of next sibling:", firstFruit.nextElementSibling.nextElementSibling.textC
  </script>
</body>
</html>
```

Why Understanding These Relationships Matters

- **Navigation:** Efficient traversal to target or update specific nodes.
- **Manipulation:** Knowing where to insert, remove, or modify elements relative to others.
- **Event Handling:** Delegating events through parent or sibling elements.

Mastering these relationships empowers you to work confidently with the DOM tree, setting the stage for advanced manipulations and interactions.

4.1.1 Summary

- **Parent nodes** contain child nodes.
- **Child nodes** are descendants of a parent.
- **Sibling nodes** share the same parent and are next to each other in the DOM tree.
- Use properties like `parentNode`, `children`, and `nextElementSibling` to navigate these relationships in JavaScript.

This foundational knowledge enables you to traverse and manipulate the DOM structure effectively in your web projects.

4.2 Navigating with `parentNode`, `children`, `nextSibling`, and `previousSibling`

Navigating the DOM programmatically is key to dynamic web development. JavaScript provides several properties to move through the DOM tree by accessing parents, children, and siblings of nodes.

Core Navigation Properties

Property	Description
`parentNode`	Accesses the parent node of the current node
`children`	Returns an **HTMLCollection** of **element** child nodes
`nextSibling`	Returns the **next sibling node** (could be element, text, comment, etc.)
`previousSibling`	Returns the **previous sibling node** (could be element, text, comment, etc.)

Understanding Node Types and Their Impact

The DOM tree includes various node types, primarily:

- **Element nodes** (e.g., `<div>`, `<p>`, `<span>`) — visible HTML elements.

- **Text nodes** — whitespace, line breaks, or text content between elements.
- **Comment nodes** — HTML comments.

When navigating siblings, `nextSibling` and `previousSibling` can return *any* node type, including text nodes (like whitespace). This can lead to unexpected results if you assume they always point to elements.

Difference Between `children` and `childNodes`

- `children` returns only **element nodes**, ignoring text and comments.
- `childNodes` returns **all child nodes**, including text and comment nodes.

Practical Examples

Consider this HTML snippet:

```
<ul id="list">
  <li>Item 1</li>
  <li>Item 2</li>
  <!-- This is a comment -->
  <li>Item 3</li>
</ul>
```

```
const list = document.getElementById('list');

// Parent node of the <ul>
console.log(list.parentNode); // Usually the <body> element

// Children of the <ul> (only element nodes)
console.log(list.children); // HTMLCollection of 3 <li> elements

// Access first child element
console.log(list.firstElementChild.textContent); // "Item 1"

// Access all child nodes (includes text nodes and comment nodes)
console.log(list.childNodes); // NodeList with text nodes and comment node included
```

Accessing Parent and Children

```
const firstItem = list.firstChild;   // Might be a text node (whitespace before <li>)
const firstElement = list.firstElementChild; // Guaranteed to be an element node <li>

console.log(firstItem.nodeType); // 3 (Text node)
console.log(firstElement.nodeType); // 1 (Element node)

// Using nextSibling (may get text nodes)
console.log(firstItem.nextSibling); // Could be the first <li> or another text node

// Using nextElementSibling (skips text nodes, moves to next element)
console.log(firstElement.nextElementSibling.textContent); // "Item 2"
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>DOM Nodes and Siblings Example</title>
</head>
<body>

<ul id="list">
  <li>Item 1</li>
  <li>Item 2</li>
  <!-- This is a comment -->
  <li>Item 3</li>
</ul>

<script>
  const list = document.getElementById('list');

  // Parent node of the <ul>
  console.log("Parent node:", list.parentNode); // Usually <body>

  // Children of the <ul> (element nodes only)
  console.log("Children (element nodes):", list.children); // HTMLCollection of 3 <li>

  // Access first child element
  console.log("First element child text:", list.firstElementChild.textContent); // "Item 1"

  // All child nodes (includes text and comment nodes)
  console.log("All child nodes (including text & comments):", list.childNodes);

  // Access first child node (likely a text node due to whitespace)
  const firstItem = list.firstChild;
  const firstElement = list.firstElementChild;

  console.log("firstChild nodeType:", firstItem.nodeType);    // 3 = Text node
  console.log("firstElementChild nodeType:", firstElement.nodeType); // 1 = Element node

  // nextSibling might be a text node
  console.log("nextSibling of firstChild:", firstItem.nextSibling);

  // nextElementSibling skips text nodes and goes directly to next element
  console.log("nextElementSibling of firstElement:", firstElement.nextElementSibling.textContent); // "
</script>

</body>
</html>
```

Navigating Siblings with `nextSibling` and `previousSibling`

Filtering to Element Nodes When Navigating Siblings To safely navigate siblings while skipping text nodes, use the `nextElementSibling` and `previousElementSibling` properties:

```
let current = list.firstElementChild; // <li>Item 1</li>
```

```
while(current) {
  console.log(current.textContent); // Logs each list item text
  current = current.nextElementSibling;
}
```

Common Pitfalls and Tips

- **Whitespace and text nodes:** Browsers treat whitespace between elements as text nodes. Using `nextSibling` or `previousSibling` may return these invisible text nodes unexpectedly.
- **Use `nextElementSibling` and `previousElementSibling` when only elements matter:** These properties skip text and comment nodes and are more predictable when manipulating visible elements.
- **`children` vs `childNodes`:** Use `children` when you want only element children; use `childNodes` if you need all node types.
- **Node types:** You can check `node.nodeType` to distinguish elements (`1`), text (`3`), and comments (`8`).

4.2.1 Summary

- `parentNode` navigates up to the parent node.
- `children` returns an element-only collection of child nodes.
- `nextSibling` and `previousSibling` include all node types, including text and comments.
- Use `nextElementSibling` and `previousElementSibling` to skip non-element siblings.
- Always be cautious of whitespace text nodes when traversing siblings.
- Combine these properties to traverse the DOM efficiently and safely.

Understanding these navigation methods and node types helps you write robust code that traverses and manipulates the DOM reliably, avoiding common pitfalls caused by hidden text nodes or unexpected node types.

4.3 Recursive DOM Traversal Techniques

When working with the DOM, you often need to explore not just immediate children or siblings but deeply nested elements. Recursion—a function calling itself—is a powerful way to traverse the DOM tree systematically and thoroughly.

Why Use Recursion for DOM Traversal?

The DOM is a hierarchical tree structure, where each node can have zero or more child nodes, each of which can have their own children, and so forth. Recursion naturally fits this tree

structure by:

- Visiting a node.
- Processing it as needed.
- Recursively visiting each of its child nodes.

This approach simplifies tasks that require inspecting or modifying all nodes in a subtree, no matter how deeply nested.

Structure of a Recursive DOM Traversal Function

A typical recursive traversal function follows this pattern:

```
function traverse(node) {
  // 1. Process the current node (e.g., read or modify it)
  processNode(node);

  // 2. Recursively call traverse on each child element
  node = node.firstElementChild; // or use children property
  while (node) {
    traverse(node);
    node = node.nextElementSibling;
  }
}
```

- The function processes the node.
- Then it iterates over its children and recursively processes them.

Example: Recursively Collect All Elements of a Certain Tag

Let's say we want to collect all <p> elements under a container:

```
<div id="content">
  <p>Paragraph 1</p>
  <div>
    <p>Paragraph 2</p>
    <span>
      <p>Paragraph 3</p>
    </span>
  </div>
</div>
```

```
function collectParagraphs(node, result = []) {
  // Check if current node is a <p> element
  if (node.nodeType === 1 && node.tagName.toLowerCase() === 'p') {
    result.push(node);
  }

  // Traverse children recursively
  let child = node.firstElementChild;
  while (child) {
    collectParagraphs(child, result);
    child = child.nextElementSibling;
  }
```

```
  return result;
}

const content = document.getElementById('content');
const paragraphs = collectParagraphs(content);

console.log(paragraphs); // Array of <p> elements inside #content
```

Example: Recursively Modify Elements (Highlight All `li` Items)

Suppose you want to add a CSS class to all `<li>` elements in a list, regardless of nesting level:

```
function highlightListItems(node) {
  if (node.nodeType === 1 && node.tagName.toLowerCase() === 'li') {
    node.classList.add('highlight');
  }

  let child = node.firstElementChild;
  while (child) {
    highlightListItems(child);
    child = child.nextElementSibling;
  }
}

// Assuming a nested list somewhere in the DOM
const listContainer = document.querySelector('.nested-list');
highlightListItems(listContainer);
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Highlight Nested List Items</title>
  <style>
    .highlight {
      background-color: yellow;
      font-weight: bold;
    }
  </style>
</head>
<body>

<div class="nested-list">
  <ul>
    <li>Item 1</li>
    <li>Item 2
      <ul>
        <li>Subitem 2.1</li>
        <li>Subitem 2.2
          <ul>
            <li>Subsubitem 2.2.1</li>
          </ul>
        </li>
      </ul>
    </li>
    <li>Item 3</li>
```

```
  </ul>
</div>

<script>
  function highlightListItems(node) {
    if (node.nodeType === 1 && node.tagName.toLowerCase() === 'li') {
      node.classList.add('highlight');
    }

    let child = node.firstElementChild;
    while (child) {
      highlightListItems(child);
      child = child.nextElementSibling;
    }
  }

  const listContainer = document.querySelector('.nested-list');
  highlightListItems(listContainer);
</script>

</body>
</html>
```

Use Cases Where Recursion Shines

- **Deeply nested structures:** When elements can be nested at arbitrary depths, such as nested menus or comment threads.
- **Complex tree processing:** Tasks like searching, filtering, transforming, or collecting nodes matching specific criteria.
- **Generic traversal:** When you need to apply the same operation to all nodes or selectively based on node types or attributes.

Tips for Recursive DOM Traversal

- Use `firstElementChild` and `nextElementSibling` to navigate only element nodes, skipping text nodes.
- Always check `nodeType === 1` to confirm you are dealing with element nodes.
- Be careful of very deep or cyclic structures (rare in DOM) to avoid stack overflows.
- Recursion helps keep your traversal logic clean and expressive, especially for complex or nested DOM operations.

4.3.1 Summary

Recursion is a natural and elegant approach for deep DOM traversal. By writing recursive functions that process a node and then visit each child node recursively, you can easily handle complex document structures. This technique greatly simplifies tasks that would be cumbersome with iterative loops alone, making your DOM manipulation code more readable and maintainable.

4.4 Practical Example: Highlighting Related Elements

In this example, we'll build a small interactive feature that highlights a group of related elements in the DOM — specifically, all siblings of a clicked element. This kind of functionality is common in user interfaces, such as dynamically highlighting menu items, grouping form fields, or emphasizing related content.

Goal

When a user clicks on an element inside a container, all its sibling elements will be visually highlighted by adding a CSS class.

HTML Setup

```
<div id="menu">
  <div class="menu-item">Home</div>
  <div class="menu-item">About</div>
  <div class="menu-item">Services</div>
  <div class="menu-item">Contact</div>
</div>

<style>
  .highlight {
    background-color: #f9d342;
    font-weight: bold;
  }
</style>
```

JavaScript Code

```
const menu = document.getElementById('menu');

menu.addEventListener('click', function(event) {
  const clicked = event.target;

  // Only act if a menu-item was clicked
  if (!clicked.classList.contains('menu-item')) return;

  // Clear previous highlights
  clearHighlights(menu);

  // Highlight siblings of clicked element, including itself
  highlightSiblings(clicked);
});

function clearHighlights(container) {
  const highlighted = container.querySelectorAll('.highlight');
  highlighted.forEach(el => el.classList.remove('highlight'));
}

function highlightSiblings(element) {
  // Get the parent node to find siblings
  const parent = element.parentNode;
```

```
  // Traverse all children of the parent
  let child = parent.firstElementChild;
  while (child) {
    if (child.classList.contains('menu-item')) {
      child.classList.add('highlight');
    }
    child = child.nextElementSibling;
  }
}
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Highlight Menu Siblings</title>
  <style>
    #menu {
      max-width: 300px;
      margin: 2em auto;
      font-family: Arial, sans-serif;
    }
    .menu-item {
      padding: 10px;
      cursor: pointer;
      border: 1px solid #ccc;
      margin-bottom: 5px;
      user-select: none;
      transition: background-color 0.3s, font-weight 0.3s;
    }
    .menu-item:hover {
      background-color: #eee;
    }
    .highlight {
      background-color: #f9d342;
      font-weight: bold;
    }
  </style>
</head>
<body>

<div id="menu">
  <div class="menu-item">Home</div>
  <div class="menu-item">About</div>
  <div class="menu-item">Services</div>
  <div class="menu-item">Contact</div>
</div>

<script>
  const menu = document.getElementById('menu');

  menu.addEventListener('click', function(event) {
    const clicked = event.target;

    // Only act if a menu-item was clicked
    if (!clicked.classList.contains('menu-item')) return;

    // Clear previous highlights
```

```
    clearHighlights(menu);

    // Highlight siblings of clicked element, including itself
    highlightSiblings(clicked);
  });

  function clearHighlights(container) {
    const highlighted = container.querySelectorAll('.highlight');
    highlighted.forEach(el => el.classList.remove('highlight'));
  }

  function highlightSiblings(element) {
    // Get the parent node to find siblings
    const parent = element.parentNode;

    // Traverse all children of the parent
    let child = parent.firstElementChild;
    while (child) {
      if (child.classList.contains('menu-item')) {
        child.classList.add('highlight');
      }
      child = child.nextElementSibling;
    }
  }
</script>

</body>
</html>
```

How This Works

1. **Event Delegation:** We attach one click listener to the `#menu` container, which captures clicks on any `.menu-item` inside it.
2. **Identify the Clicked Element:** We check if the clicked target has the `.menu-item` class to ensure we highlight the correct group.
3. **Clear Previous Highlights:** Before highlighting the new group, we remove any existing `.highlight` classes.
4. **Highlight Siblings:** Using DOM traversal, we access all sibling `.menu-item` elements of the clicked item by iterating over the parent's children and add the `.highlight` class.

Real-World Connections

- **Dynamic Menus:** This pattern is useful for highlighting a section or group of menu items based on user interaction.
- **Forms:** You could highlight related form fields dynamically when a user focuses on one field.
- **Content Grouping:** Highlighting related cards or sections within a page for better user focus.

Experiment Ideas

- Modify `highlightSiblings()` to only highlight *true siblings* (excluding the clicked element).
- Change the highlight style to animate color changes or add borders.
- Extend the logic to highlight nested children recursively.
- Use recursion to highlight all descendants of a clicked element, not just siblings.

4.4.1 Summary

This example combines event handling with DOM traversal to manipulate groups of related elements dynamically. Using simple traversal methods like `parentNode` and iterating children, you can build interactive UI features that respond intuitively to user actions. Experimenting with these techniques helps solidify understanding and prepares you to tackle more complex DOM manipulations.

Chapter 5.

Handling DOM Events

1. Introduction to Events and Event Listeners
2. Event Types: Mouse, Keyboard, Form, and Custom Events
3. Adding and Removing Event Listeners
4. Event Propagation: Capturing and Bubbling
5. Practical Example: Interactive Button with Multiple Events

5 Handling DOM Events

5.1 Introduction to Events and Event Listeners

The web is fundamentally **event-driven**. This means that web pages don't just sit there passively; they react dynamically when users interact with them or when the browser triggers certain actions. This reactive behavior is made possible through **DOM events**.

What Are DOM Events?

A **DOM event** is a signal that something has happened on a web page. Events can be triggered by user interactions like clicks, key presses, or mouse movements — or by browser actions such as page loading or resizing.

Examples of common events include:

- `click` — when a user clicks a button or link
- `keydown` — when a user presses a key
- `submit` — when a form is submitted
- `load` — when a page or image finishes loading

Responding to Events with Event Listeners

JavaScript responds to these events using **event listeners** — functions that wait for and react to specific events on particular DOM elements.

When an event occurs, the browser calls the event listener function, allowing your code to run in response. This is how websites become interactive and dynamic.

Event Types vs. Event Handlers

- **Event Types:** These are predefined event names (like `"click"` or `"mouseover"`) that describe the kind of user or browser action to listen for.
- **Event Handlers (Listeners):** These are the functions you write that get called when the event happens.

Basic Example: Adding an Event Listener

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

<script>
  const button = document.getElementById('myButton');

  // Add an event listener for the 'click' event
  button.addEventListener('click', function() {
    console.log('Button was clicked!');
  });
</script>
```

Here's what happens:

1. We select the button element with `getElementById`.
2. We attach an event listener that listens for the `click` event.
3. When the button is clicked, the anonymous function runs and displays an alert.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Button Click Example</title>
</head>
<body>

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

<script>
  const button = document.getElementById('myButton');

  // Add an event listener for the 'click' event
  button.addEventListener('click', function() {
    console.log('Button was clicked!');
  });
</script>

</body>
</html>
```

Summary

- DOM events are signals that something happened on the page.
- JavaScript reacts to these signals with event listeners — functions that run when events occur.
- Understanding how to attach event listeners to DOM elements is foundational for building interactive web pages.

In the next sections, we'll dive deeper into the variety of event types and how to manage multiple event listeners effectively.

5.2 Event Types: Mouse, Keyboard, Form, and Custom Events

5.2.1 Event Types: Mouse, Keyboard, Form, and Custom Events

DOM events come in many flavors, categorized by the kind of user or programmatic interaction they represent. Understanding these event types helps you choose the right event to listen for and handle user input effectively.

Mouse Events

Mouse events respond to user actions involving the mouse (or similar pointing devices). Common mouse events include:

- **`click`**: Fires when the user clicks an element (presses and releases the mouse button).
- **`dblclick`**: Fires on a double-click.
- **`mouseover`**: Fires when the mouse pointer moves onto an element.
- **`mouseout`**: Fires when the mouse pointer leaves an element.
- **`mousemove`**: Fires when the mouse pointer moves within an element.

Example: Handling a click event

```
const button = document.getElementById('btn');

button.addEventListener('click', event => {
  console.log('Button clicked at coordinates:', event.clientX, event.clientY);
});
```

The `event` object provides useful properties like `clientX` and `clientY` (mouse position).

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Button Click Coordinates</title>
</head>
<body>

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

<script>
  const button = document.getElementById('btn');

  button.addEventListener('click', event => {
    console.log('Button clicked at coordinates:', event.clientX, event.clientY);
  });
</script>

</body>
</html>
```

Keyboard Events

Keyboard events detect user keyboard interactions. The main keyboard events are:

- **`keydown`**: Fires when a key is pressed down.
- **`keyup`**: Fires when a key is released.
- **`keypress`** (deprecated): Previously used for character input.

Example: Detecting the Enter key press

```
document.addEventListener('keydown', event => {
  if (event.key === 'Enter') {
    console.log('Enter key was pressed!');
  }
});
```

The event object has useful properties such as `key` (the character) and `code` (physical key

location).

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Enter Key Listener</title>
</head>
<body>

<p>Press the Enter key anywhere on this page.</p>

<script>
  document.addEventListener('keydown', event => {
    if (event.key === 'Enter') {
      console.log('Enter key was pressed!');
    }
  });
</script>

</body>
</html>
```

Form Events

Form events occur when users interact with form elements:

- **`submit`**: Fires when a form is submitted.
- **`change`**: Fires when an input, select, or textarea value changes and loses focus.
- **`input`**: Fires every time the value of an input changes (more immediate than `change`).

Example: Handling form submission

```
<form id="myForm">
  <input type="text" name="username" />
  <button type="submit">Submit</button>
</form>

<script>
  const form = document.getElementById('myForm');

  form.addEventListener('submit', event => {
    event.preventDefault(); // Prevents page reload
    console.log('Form submitted:', form.username.value);
  });
</script>
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Form Submit Example</title>
</head>
<body>
```

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

<script>
  const form = document.getElementById('myForm');

  form.addEventListener('submit', event => {
    event.preventDefault(); // Prevent page reload
    console.log('Form submitted:', form.username.value);
  });
</script>

</body>
</html>
```

Custom Events

Sometimes, built-in events don't cover your app's specific needs. You can create and dispatch **custom events** to communicate between parts of your application.

Creating and dispatching a custom event:

```
// Create a custom event with optional detail data
const myEvent = new CustomEvent('myCustomEvent', { detail: { message: 'Hello!' } });

// Listen for the custom event
document.addEventListener('myCustomEvent', event => {
  console.log('Custom event received:', event.detail.message);
});

// Dispatch the custom event
document.dispatchEvent(myEvent);
```

Custom events are powerful for decoupling components and enabling event-driven architecture beyond DOM user interactions.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Custom Event Example</title>
</head>
<body>

<script>
  // Create a custom event with optional detail data
  const myEvent = new CustomEvent('myCustomEvent', { detail: { message: 'Hello!' } });

  // Listen for the custom event
  document.addEventListener('myCustomEvent', event => {
    console.log('Custom event received:', event.detail.message);
  });
```

```
  // Dispatch the custom event
  document.dispatchEvent(myEvent);
</script>

</body>
</html>
```

5.2.2 Summary

Event Category	Common Events	When to Use	Key Event Properties
Mouse	`click`, `dblclick`, `mouseover`	User clicks, hovering, dragging	`clientX`, `clientY`, `button`
Keyboard	`keydown`, `keyup`	Detect key presses and releases	`key`, `code`, `altKey`, `ctrlKey`
Form	`submit`, `change`, `input`	Form interactions and input changes	`target.value`, `preventDefault()`
Custom	User-defined	Application-specific events	`detail` (custom data)

By understanding these event types and their specific uses, you can build responsive, interactive web applications that react fluidly to user input and programmatic triggers.

In the next section, we'll explore how to **add and remove** these event listeners efficiently and cleanly.

5.3 Adding and Removing Event Listeners

Handling events efficiently is key to creating interactive web applications. In JavaScript, you attach event listeners to DOM elements to respond when users interact with them. Knowing how to properly add and remove these listeners helps manage performance and avoid memory leaks.

Adding Event Listeners with `addEventListener()`

The primary method to listen for events is `addEventListener()`. It attaches a function (the event handler) to an element for a specific event type.

Syntax:

```
element.addEventListener(eventType, handlerFunction, options);
```

- `eventType` — a string like `"click"`, `"keydown"`, `"submit"`, etc.
- `handlerFunction` — the function to call when the event occurs.
- `options` (optional) — an object or boolean specifying characteristics like `capture`, `once`, and `passive`.

Example:

```
const btn = document.getElementById('myButton');

function handleClick(event) {
  console.log('Button clicked!');
}

btn.addEventListener('click', handleClick);
```

You can add multiple listeners to the same element and event type; all will be invoked in the order they were added.

Removing Event Listeners with `removeEventListener()`

To remove an event listener, you use `removeEventListener()` with the same event type and function reference you passed to `addEventListener()`.

Important: The function must be a named or stored function reference — anonymous functions can't be removed.

Example:

```
btn.removeEventListener('click', handleClick);
```

This stops `handleClick` from firing on button clicks.

Why Remove Event Listeners?

Removing event listeners is important for:

- **Performance:** Prevent unnecessary work when elements are no longer relevant (e.g., after removing elements from the DOM).
- **Memory Management:** Avoid memory leaks caused by lingering references to elements and functions that should be garbage collected.

Especially in single-page applications where elements appear and disappear dynamically, cleaning up listeners is crucial.

Adding Multiple Listeners and Removing Them Dynamically

You can add multiple listeners for different events or even multiple handlers for the same event:

```
function onHover() {
  console.log('Mouse over button!');
}

btn.addEventListener('mouseover', onHover);
btn.addEventListener('click', handleClick);
```

Later, you can selectively remove listeners:

```
btn.removeEventListener('mouseover', onHover);
```

Listener Options: `once` and `capture`

The third parameter of `addEventListener()` can be:

- **Boolean**: If `true`, the listener is set in the capturing phase (more on event propagation later).
- **Object**: More options for fine control:
  - `capture`: `true` or `false` — whether the event is captured during the capture phase.
  - `once`: `true` — listener automatically removed after being invoked once.
  - `passive`: `true` — indicates the listener will never call `preventDefault()`, allowing browser optimizations.

Example: Using `once`

```
btn.addEventListener('click', () => {
  console.log('This will log only once.');
}, { once: true });
```

This listener self-removes after the first click.

5.3.1 Summary

Method	Purpose	Important Notes
`addEventListener()`	Attach a listener to an element	Can add multiple listeners; supports options like `once` and `capture`
`removeEventListener()`	Remove a previously attached listener	Requires the same function reference; essential for cleanup

5.3.2 Complete Example: Adding and Removing Listeners

```
<button id="toggleBtn">Click Me</button>
<button id="removeListenerBtn">Remove Click Listener</button>

<script>
  const toggleBtn = document.getElementById('toggleBtn');
  const removeBtn = document.getElementById('removeListenerBtn');

  function onClick() {
    console.log('Button clicked!');
  }

  // Add listener
  toggleBtn.addEventListener('click', onClick);

  // Remove listener when the other button is clicked
  removeBtn.addEventListener('click', () => {
    toggleBtn.removeEventListener('click', onClick);
    console.log('Click listener removed!');
  });
</script>
```

Try clicking the first button — it will show an alert. Click the second button to remove the listener, then try clicking the first button again.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Remove Event Listener Example</title>
</head>
<body>

<button id="toggleBtn">Click Me</button>
<button id="removeListenerBtn">Remove Click Listener</button>

<script>
  const toggleBtn = document.getElementById('toggleBtn');
  const removeBtn = document.getElementById('removeListenerBtn');

  function onClick() {
    console.log('Button clicked!');
  }

  // Add listener
  toggleBtn.addEventListener('click', onClick);

  // Remove listener when the other button is clicked
  removeBtn.addEventListener('click', () => {
    toggleBtn.removeEventListener('click', onClick);
    console.log('Click listener removed!');
  });
</script>

</body>
```

```
</html>
```

5.3.3 Tips for Effective Event Listener Management

- Always store your event handler functions in variables or named functions if you plan to remove them later.
- Use the `once` option when you want a listener to trigger just one time.
- Clean up listeners when removing or replacing DOM elements.
- Prefer `addEventListener()` over inline event handlers (e.g., `onclick`) for better flexibility and maintainability.

With these basics, you can confidently attach, manage, and remove event listeners to build interactive, performant web pages. Next, we'll dive deeper into how events propagate through the DOM!

5.4 Event Propagation: Capturing and Bubbling

When a user interacts with a web page, events don't just fire on a single element — they follow a path through the DOM tree. Understanding **event propagation** is crucial for managing how and when event handlers execute.

5.4.1 The Two Phases of Event Propagation

Event propagation happens in **two main phases**:

1. **Capturing phase** (also called "trickling down"): The event starts from the top of the DOM tree (the root, usually `window` or `document`) and travels down **toward** the target element.
2. **Bubbling phase**: After reaching the target element, the event bubbles **back up** from the target to the root.

These phases allow event listeners at various levels of the DOM tree to respond to the event either **before** or **after** it reaches the target.

5.4.2 Visualizing Event Flow

Consider this HTML snippet:

```
<div id="outer">
  <button id="inner">Click me</button>
</div>
```

When you click the button:

- The event first **captures** down from the root → `<html>` → `<body>` → `#outer` → `#inner`.
- The event then fires on the **target** (`#inner` button).
- Finally, the event **bubbles** back up: `#inner` → `#outer` → `<body>` → `<html>` → `window`.

5.4.3 Controlling Propagation with `addEventListener()`

By default, event listeners listen during the **bubbling phase**. You can control this behavior using the third argument of `addEventListener()`.

```
element.addEventListener(type, listener, useCapture);
```

- If `useCapture` is `true`, the listener runs during the **capturing** phase.
- If `useCapture` is `false` or omitted, the listener runs during **bubbling**.

Example:

```
const outer = document.getElementById('outer');
const inner = document.getElementById('inner');

outer.addEventListener('click', () => {
  console.log('Outer DIV clicked (bubbling)');
});

outer.addEventListener('click', () => {
  console.log('Outer DIV clicked (capturing)');
}, true);

inner.addEventListener('click', () => {
  console.log('Inner BUTTON clicked');
});
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Event Bubbling and Capturing</title>
  <style>
```

```
    #outer {
      border: 2px solid #555;
      padding: 20px;
      margin: 30px;
    }

    #inner {
      padding: 10px 20px;
    }
  </style>
</head>
<body>

<div id="outer">
  <button id="inner">Click me</button>
</div>

<script>
  const outer = document.getElementById('outer');
  const inner = document.getElementById('inner');

  // Capturing phase listener
  outer.addEventListener('click', () => {
    console.log('Outer DIV clicked (capturing)');
  }, true);

  // Bubbling phase listener
  outer.addEventListener('click', () => {
    console.log('Outer DIV clicked (bubbling)');
  });

  // Target listener
  inner.addEventListener('click', () => {
    console.log('Inner BUTTON clicked');
  });
</script>

</body>
</html>
```

Clicking the button logs:

```
Outer DIV clicked (capturing)
Inner BUTTON clicked
Outer DIV clicked (bubbling)
```

5.4.4 Stopping Event Propagation

Sometimes you want to **stop** the event from continuing to propagate after a certain handler runs. You can do this with:

- `event.stopPropagation()`: Stops further propagation in both capturing and bubbling.

- `event.stopImmediatePropagation()`: Also prevents other listeners on the same element from running.

Example:

```
inner.addEventListener('click', (event) => {
  console.log('Inner BUTTON clicked - stopping propagation');
  event.stopPropagation(); // prevents bubbling up to outer
});
```

Now clicking the button will **not** trigger the outer DIV's bubbling listener.

5.4.5 Summary

Phase	Description	Listener Behavior
Capturing	Event travels *down* the DOM tree to the target	Listener runs if `useCapture` is `true`
Target	Event reaches the element that triggered it	All listeners run here
Bubbling	Event travels *up* the DOM tree from the target	Default phase; listener runs if `useCapture` is `false` or omitted

5.4.6 Practical Takeaways

- Use capturing listeners (`useCapture: true`) when you want to intercept events **before** they reach the target.
- Use bubbling listeners (default) for most common cases.
- Use `event.stopPropagation()` to prevent events from bubbling or capturing further.
- Be mindful of event propagation when nesting interactive elements, like buttons inside clickable divs.

5.4.7 Complete Example

```
<div id="outer" style="padding: 20px; background: lightblue;">
  Outer DIV
  <button id="inner">Inner BUTTON</button>
</div>

<script>
```

```
  const outer = document.getElementById('outer');
  const inner = document.getElementById('inner');

  outer.addEventListener('click', () => {
    console.log('Outer DIV clicked (bubbling)');
  });

  outer.addEventListener('click', () => {
    console.log('Outer DIV clicked (capturing)');
  }, true);

  inner.addEventListener('click', (event) => {
    console.log('Inner BUTTON clicked');
    // Uncomment to stop bubbling
    // event.stopPropagation();
  });
</script>
```

Try clicking the button and observe the console logs. Experiment with `stopPropagation()` to see how it affects event flow.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Event Capturing and Bubbling</title>
</head>
<body>
  <div id="outer" style="padding: 20px; background: lightblue;">
    Outer DIV
    <button id="inner">Inner BUTTON</button>
  </div>

  <script>
    const outer = document.getElementById('outer');
    const inner = document.getElementById('inner');

    // Capturing phase handler
    outer.addEventListener('click', () => {
      console.log('Outer DIV clicked (capturing)');
    }, true);

    // Bubbling phase handler
    outer.addEventListener('click', () => {
      console.log('Outer DIV clicked (bubbling)');
    });

    inner.addEventListener('click', (event) => {
      console.log('Inner BUTTON clicked');
      // Uncomment the next line to stop bubbling
      // event.stopPropagation();
    });
  </script>
</body>
</html>
```

Understanding event propagation helps you build complex interactions without unintended

side effects and enables efficient event delegation. Next, we'll explore how to manage event listeners dynamically for robust web applications.

5.5 Practical Example: Interactive Button with Multiple Events

In this section, we'll create a small interactive example where a button responds to multiple events: `click`, `mouseenter`, and `focus`. This will help you see how you can attach and manage multiple event listeners on a single element and explore useful event properties like `event.target` and `event.currentTarget`.

5.5.1 The HTML Setup

Let's start with a simple button in the HTML:

```
<button id="myButton">Hover, Focus, or Click Me!</button>
<div id="log"></div>
```

We also add a `<div>` with id `log` to display event messages.

5.5.2 JavaScript: Adding Multiple Event Listeners

Here's how we attach multiple listeners to the button for different events:

```
const button = document.getElementById('myButton');
const log = document.getElementById('log');

// Helper function to log messages
function logMessage(message) {
  const p = document.createElement('p');
  p.textContent = message;
  log.appendChild(p);
}

// Click event listener
button.addEventListener('click', (event) => {
  logMessage(`Button clicked! event.target: ${event.target.tagName}, event.currentTarget: ${event.curre
});

// Mouseenter event listener
button.addEventListener('mouseenter', (event) => {
  logMessage('Mouse entered the button area.');
});
```

```
// Focus event listener
button.addEventListener('focus', (event) => {
  logMessage('Button is focused (via keyboard or tab).');
});
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Button Events Logger</title>
  <style>
    #myButton {
      padding: 10px 20px;
      font-size: 16px;
      cursor: pointer;
    }
    #log {
      margin-top: 20px;
      max-height: 200px;
      overflow-y: auto;
      background: #f0f0f0;
      padding: 10px;
      font-family: monospace;
      border: 1px solid #ccc;
    }
    #log p {
      margin: 4px 0;
    }
  </style>
</head>
<body>
  <button id="myButton">Hover, Focus, or Click Me!</button>
  <div id="log"></div>

  <script>
    const button = document.getElementById('myButton');
    const log = document.getElementById('log');

    // Helper function to log messages
    function logMessage(message) {
      const p = document.createElement('p');
      p.textContent = message;
      log.appendChild(p);
      // Scroll to bottom to show the latest message
      log.scrollTop = log.scrollHeight;
    }

    // Click event listener
    button.addEventListener('click', (event) => {
      logMessage(`Button clicked! event.target: ${event.target.tagName}, event.currentTarget: ${event.cu
    });

    // Mouseenter event listener
    button.addEventListener('mouseenter', () => {
      logMessage('Mouse entered the button area.');
    });
```

```
    // Focus event listener
    button.addEventListener('focus', () => {
      logMessage('Button is focused (via keyboard or tab).');
    });
  </script>
</body>
</html>
```

5.5.3 Whats Happening?

- **Multiple event listeners:** We add three different listeners for `click`, `mouseenter`, and `focus`. They all coexist and respond independently to their respective events on the same button.
- **Event properties:**
  - `event.target` refers to the element where the event **originated** (in this example, always the button itself since it's a simple element).
  - `event.currentTarget` refers to the element the event listener is **attached to** — here, also the button.

These two are often the same, but when dealing with event delegation (listeners on parent elements), they can differ.

5.5.4 Try It Out

Open this HTML and JavaScript in your browser. When you:

- Hover the mouse over the button, you'll see "Mouse entered the button area."
- Click the button, you'll see "Button clicked!" with information about event targets.
- Tab to focus the button, you'll see "Button is focused…"

5.5.5 Extending the Example

You can try adding more events like:

- `mouseleave` to detect when the mouse leaves the button.
- `keydown` or `keyup` to handle keyboard events.
- `dblclick` for double-click actions.

You can also apply the same techniques to other interactive elements such as links, input fields, or custom widgets.

This example illustrates how DOM events can be layered on a single element, offering rich interactivity. Experiment by adding/removing listeners or applying this to other elements to deepen your understanding of event handling!

Chapter 6.

Advanced Event Handling

1. Event Delegation Techniques
2. Preventing Default Actions and Stopping Propagation
3. Throttling and Debouncing Event Handlers
4. Working with Event Objects and Targets
5. Practical Example: Efficient List Item Click Handling

6 Advanced Event Handling

6.1 Event Delegation Techniques

When building interactive web applications, you often need to handle events on many similar elements—like items in a list or cells in a table. Attaching individual event listeners to each child element can become inefficient and cumbersome, especially if the list changes dynamically. This is where **event delegation** shines.

6.1.1 What Is Event Delegation?

Event delegation is a technique where a single event listener is attached to a **common parent** element instead of each individual child element. Because of **event bubbling**—where events propagate up from the target element through its ancestors—the parent can catch events triggered by any of its children.

Using `event.target`, the parent listener can identify exactly which child was interacted with and respond accordingly.

6.1.2 Why Use Event Delegation?

- **Performance:** Only one event listener is needed, regardless of how many child elements exist. This reduces memory usage and speeds up event handling.
- **Dynamic elements:** If child elements are added or removed dynamically, the parent listener automatically manages events for new children without needing to attach or remove listeners.
- **Cleaner code:** Managing events in one place keeps your code simpler and easier to maintain.

6.1.3 Example: Delegating Clicks in a List

Let's look at a practical example of event delegation in action.

6.1.4 HTML

```
<ul id="todoList">
  <li data-id="1">Learn JavaScript</li>
  <li data-id="2">Build a project</li>
  <li data-id="3">Read documentation</li>
</ul>
```

6.1.5 JavaScript with Event Delegation

```
const list = document.getElementById('todoList');

list.addEventListener('click', (event) => {
  // event.target is the clicked element inside the list
  const clickedItem = event.target;

  // Check if the clicked element is an <li> (to avoid clicks on the <ul> itself)
  if (clickedItem.tagName === 'LI') {
    const itemId = clickedItem.getAttribute('data-id');
    console.log(`Clicked on item with ID: ${itemId} and text: "${clickedItem.textContent}"`);
  }
});
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Todo List Event Delegation</title>
  <style>
    #todoList {
      list-style-type: none;
      padding: 0;
      max-width: 300px;
      font-family: Arial, sans-serif;
    }
    #todoList li {
      padding: 10px;
      margin-bottom: 5px;
      background: #e0f7fa;
      cursor: pointer;
      border-radius: 4px;
      transition: background-color 0.2s ease;
    }
    #todoList li:hover {
      background-color: #b2ebf2;
    }
  </style>
</head>
<body>
  <ul id="todoList">
```

```
    <li data-id="1">Learn JavaScript</li>
    <li data-id="2">Build a project</li>
    <li data-id="3">Read documentation</li>
  </ul>

  <script>
    const list = document.getElementById('todoList');

    list.addEventListener('click', (event) => {
      const clickedItem = event.target;

      // Make sure we clicked on an LI element
      if (clickedItem.tagName === 'LI') {
        const itemId = clickedItem.getAttribute('data-id');
        console.log(`Clicked on item with ID: ${itemId} and text: "${clickedItem.textContent}"`);
      }
    });
  </script>
</body>
</html>
```

6.1.6 How This Works:

- We add a **single** click listener to the `<ul>` element.
- When any `<li>` is clicked, the event bubbles up to the `<ul>`.
- Inside the listener, `event.target` identifies the exact `<li>` clicked.
- We check that the target is an `<li>` to prevent handling clicks on empty parts of the `<ul>`.
- Then we retrieve the item's data and perform an action (here, showing an alert).

6.1.7 Handling Dynamic Children

If you later add new `<li>` items dynamically, this listener will **automatically** handle clicks on them without needing to attach new listeners:

```
const newItem = document.createElement('li');
newItem.textContent = "Practice event delegation";
newItem.setAttribute('data-id', '4');
list.appendChild(newItem);
// No extra event listener needed for newItem!
```

6.1.8 Event Delegation in Tables

The same concept applies to more complex structures like tables:

```
<table id="userTable">
  <tr><td data-user="1">Alice</td></tr>
  <tr><td data-user="2">Bob</td></tr>
</table>
```

```
const table = document.getElementById('userTable');

table.addEventListener('click', (event) => {
  if (event.target.tagName === 'TD') {
    const userId = event.target.getAttribute('data-user');
    console.log(`User ID clicked: ${userId}`);
  }
});
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>User Table Click</title>
  <style>
    #userTable {
      border-collapse: collapse;
      width: 200px;
      font-family: Arial, sans-serif;
    }
    #userTable td {
      border: 1px solid #ccc;
      padding: 10px;
      cursor: pointer;
      text-align: center;
      transition: background-color 0.2s;
    }
    #userTable td:hover {
      background-color: #f0f8ff;
    }
  </style>
</head>
<body>

  <table id="userTable">
    <tr><td data-user="1">Alice</td></tr>
    <tr><td data-user="2">Bob</td></tr>
  </table>

  <script>
    const table = document.getElementById('userTable');

    table.addEventListener('click', (event) => {
      if (event.target.tagName === 'TD') {
        const userId = event.target.getAttribute('data-user');
        console.log(`User ID clicked: ${userId}`);
      }
```

```
    });
  </script>

</body>
</html>
```

6.1.9 Summary of Advantages

Advantage	Description
Reduced memory usage	Fewer listeners, less overhead
Automatic for new elements	No need to attach/detach listeners dynamically
Simplified codebase	Centralized event handling logic
Better performance	Especially beneficial for large, dynamic lists

6.1.10 Key Points to Remember

- Always verify the `event.target` inside the delegated listener to ensure you respond to the correct element.
- Be mindful of event bubbling and how it affects which elements receive the event.
- Delegation works best when the parent element wraps the child elements that generate events.
- For very complex UI trees, sometimes combining delegation with direct listeners is appropriate.

Event delegation is a fundamental technique that leverages the DOM's event bubbling mechanism to efficiently handle events on many child elements. Mastering it will make your event handling code more performant, scalable, and easier to maintain!

6.2 Preventing Default Actions and Stopping Propagation

6.2.1 Preventing Default Actions and Stopping Propagation

When working with DOM events, understanding how to control the behavior of events is crucial. Sometimes, you need to prevent the browser's default behavior or stop the event from bubbling further through the DOM. JavaScript provides methods to handle these scenarios: `event.preventDefault()`, `event.stopPropagation()`, and `event.stopImmediatePropagation()`.

6.2.2 `event.preventDefault()`

By default, many HTML elements trigger browser actions when events occur. For example:

- Clicking a link navigates to another page.
- Submitting a form reloads the page.
- Pressing a key might trigger browser shortcuts.

If you need to **prevent** these default behaviors (for example, to handle navigation or form submission with JavaScript), you use:

```
event.preventDefault();
```

6.2.3 Example: Preventing a Link from Navigating

```
<a href="https://example.com" id="myLink">Go to Example.com</a>
```

```
const link = document.getElementById('myLink');
link.addEventListener('click', (event) => {
  event.preventDefault(); // Stop the browser from navigating
  console.log('Link clicked, but navigation prevented!');
});
```

In this example, clicking the link triggers the alert, but the browser **does not** follow the URL.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Prevent Link Navigation</title>
  <style>
    #myLink {
      font-size: 18px;
      color: blue;
      text-decoration: underline;
      cursor: pointer;
    }
  </style>
</head>
<body>

  <a href="https://example.com" id="myLink">Go to Example.com</a>

  <script>
    const link = document.getElementById('myLink');
    link.addEventListener('click', (event) => {
      event.preventDefault(); // Stop the browser from navigating
      console.log('Link clicked, but navigation prevented!');
```

```
    });
  </script>

</body>
</html>
```

6.2.4 `event.stopPropagation()`

Events bubble up the DOM tree by default, meaning that when an event is triggered on a child element, it also fires on all ancestor elements unless stopped.

Sometimes, you want to **stop the event from propagating further up**:

```
event.stopPropagation();
```

This prevents parent elements from receiving the same event.

6.2.5 Example: Stopping Propagation

```
<div id="parent" style="padding:20px; border: 1px solid black;">
  Parent Div
  <button id="childBtn">Click Me</button>
</div>
```

```
const parent = document.getElementById('parent');
const childBtn = document.getElementById('childBtn');

parent.addEventListener('click', () => {
  console.log('Parent clicked');
});

childBtn.addEventListener('click', (event) => {
  event.stopPropagation(); // Prevent parent listener from firing
  console.log('Button clicked');
});
```

Clicking the button shows only **"Button clicked"** alert because the event does **not** bubble to the parent.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Event Propagation Demo</title>
  <style>
```

```
    #parent {
      padding: 20px;
      border: 1px solid black;
      max-width: 200px;
      font-family: Arial, sans-serif;
    }
    #childBtn {
      margin-top: 10px;
      padding: 8px 12px;
      cursor: pointer;
    }
  </style>
</head>
<body>

  <div id="parent">
    Parent Div
    <button id="childBtn">Click Me</button>
  </div>

  <script>
    const parent = document.getElementById('parent');
    const childBtn = document.getElementById('childBtn');

    parent.addEventListener('click', () => {
      console.log('Parent clicked');
    });

    childBtn.addEventListener('click', (event) => {
      event.stopPropagation(); // Prevent parent listener from firing
      console.log('Button clicked');
    });
  </script>

</body>
</html>
```

6.2.6 `event.stopImmediatePropagation()`

This method does everything `stopPropagation()` does **plus** it prevents **other listeners on the same element** from running.

Use it when you want to ensure **no other event listeners**, even on the same element, are called.

6.2.7 Example: Stopping Immediate Propagation

```
button.addEventListener('click', (event) => {
  console.log('First handler');
  event.stopImmediatePropagation();
});

button.addEventListener('click', () => {
  console.log('Second handler'); // This will NOT run
});
```

Here, only the "First handler" alert will show because the propagation stops immediately.

6.2.8 Common Mistakes and Consequences

- **Not calling `preventDefault()` when needed:** You may get unexpected page reloads or navigations.
- **Overusing `stopPropagation()` or `stopImmediatePropagation()`:** It can break event delegation or interfere with other event handlers unexpectedly.
- **Confusing `stopPropagation()` with `preventDefault()`:** They control different things—one controls event flow, the other the browser's default action.
- **Not checking event targets:** Sometimes you only want to prevent default or stop propagation for specific elements/events.

6.2.9 Summary Table

Method	Purpose	When to Use
`event.preventDefault()`	Prevent browser's default action for the event	Prevent link navigation, form submission, etc.
`event.stopPropagation()`	Stop event bubbling to parent elements	Prevent ancestor event handlers from firing
`event.stopImmediatePropagation()`	Stop bubbling and prevent other listeners on the same element	When you want exclusive control on an element

By mastering these methods, you can precisely control how events behave, creating rich, responsive interfaces without unintended side effects.

6.3 Throttling and Debouncing Event Handlers

When working with events that can fire very frequently—such as `mousemove`, `scroll`, or `resize`—executing an event handler on **every** event can cause serious performance issues. To improve efficiency and responsiveness, two common techniques are used to **control how often** event handlers run: **throttling** and **debouncing**.

6.3.1 What Are Throttling and Debouncing?

Both throttling and debouncing limit the rate at which a function is called, but they do so in different ways:

Technique	Description	When to Use
Throttling	Ensures a function runs at most once every specified interval (e.g., once per 200ms).	Use when you want to handle periodic updates regularly but limit frequency (e.g., scroll or resize).
Debouncing	Delays function execution until after a specified period of inactivity (e.g., wait 300ms after the last event).	Use when you want to trigger an action only after the user stops performing the event (e.g., search input).

6.3.2 Throttling Explained

Throttling **limits** the number of times a function can execute over time. Even if an event fires continuously, throttling guarantees the handler runs at a controlled, steady pace.

6.3.3 Throttle Example

```
function throttle(fn, delay) {
  let lastCall = 0;
  return function (...args) {
    const now = Date.now();
    if (now - lastCall >= delay) {
      lastCall = now;
      fn.apply(this, args);
    }
  };
}
```

```
// Usage: Log mouse position at most once every 200ms
window.addEventListener('mousemove', throttle((event) => {
  console.log(`Mouse at (${event.clientX}, ${event.clientY})`);
}, 200));
```

Here, even though `mousemove` fires very rapidly, the logging happens at most once every 200 milliseconds.

6.3.4 Debouncing Explained

Debouncing **delays** execution until the event stops firing for a certain period. If the event keeps happening, the timer resets.

6.3.5 Debounce Example

```
function debounce(fn, delay) {
  let timerId;
  return function (...args) {
    clearTimeout(timerId);
    timerId = setTimeout(() => fn.apply(this, args), delay);
  };
}

// Usage: Log window resize event after user finishes resizing (no resize for 300ms)
window.addEventListener('resize', debounce(() => {
  console.log('Resize event finished');
}, 300));
```

This means the handler only runs **once** the user has stopped resizing the window for 300 milliseconds.

6.3.6 Choosing Between Throttle and Debounce

- **Throttle** is ideal for events where you want to perform **regular updates** but not overload your application, such as:
  - Updating scroll position indicators.
  - Animating elements on scroll.
  - Resizing layouts dynamically.
- **Debounce** works best when you want to react **after the user stops** an action, such as:

– Waiting for text input to finish before sending a search request.
– Reacting to window resizing once resizing stops.
– Form validation after typing is complete.

6.3.7 Summary

Aspect	Throttling	Debouncing
Execution pattern	Executes function at most once per interval	Executes function only after inactivity
Use cases	Scroll, resize, mousemove events	Search input, form validation, window resize
Effect	Limits function calls over time	Delays function call until quiet period

By applying throttling and debouncing, you can create smoother user experiences and avoid performance pitfalls associated with rapid event firing. Experiment with these patterns to find the right balance for your application's needs.

6.4 Working with Event Objects and Targets

When you add an event listener in JavaScript, the browser passes an **event object** to your event handler function. This object contains valuable information about the event and methods to control its behavior. Understanding the event object is key to writing effective and responsive event handlers.

6.4.1 Key Properties of the Event Object

Here are some of the most commonly used properties and methods available on event objects:

- `type` The type of event that occurred, such as `"click"`, `"keydown"`, `"submit"`, etc.
- `target` The actual DOM element on which the event originated — the element that was **clicked**, **hovered**, or interacted with.
- `currentTarget` The DOM element on which the event listener is currently invoked. This can differ from `target` in event delegation scenarios.
- `preventDefault()` A method that stops the browser's default action related to the event, for example, preventing a link from navigating or a form from submitting.

- **`stopPropagation()`** Stops the event from bubbling up (or capturing down) to other listeners.

6.4.2 Understanding `event.target` vs. `event.currentTarget`

These two properties are often confused but serve distinct purposes:

- **`event.target`** is the **element that triggered the event** — the deepest nested element that was actually interacted with.
- **`event.currentTarget`** is the **element the event listener is attached to** — the element handling the event at that moment during propagation.

6.4.3 Example: Event Delegation on a List

```
<ul id="menu">
  <li><button>Home</button></li>
  <li><button>About</button></li>
  <li><button>Contact</button></li>
</ul>
```

```
const menu = document.getElementById('menu');

menu.addEventListener('click', (event) => {
  console.log('target:', event.target.tagName);        // Element clicked, e.g., BUTTON
  console.log('currentTarget:', event.currentTarget.id); // The UL element 'menu'
});
```

If a user clicks the "About" button, `event.target` will be the `<button>` element, while `event.currentTarget` will always be the `<ul id="menu">`, because the listener is attached to the `<ul>`.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Menu Event Delegation</title>
  <style>
    #menu {
      list-style: none;
      padding: 0;
      font-family: Arial, sans-serif;
      max-width: 200px;
    }
    #menu li {
      margin: 8px 0;
```

```
    }
    #menu button {
      width: 100%;
      padding: 8px 12px;
      cursor: pointer;
      font-size: 16px;
    }
  </style>
</head>
<body>

  <ul id="menu">
    <li><button>Home</button></li>
    <li><button>About</button></li>
    <li><button>Contact</button></li>
  </ul>

  <script>
    const menu = document.getElementById('menu');

    menu.addEventListener('click', (event) => {
      console.log('target:', event.target.tagName);          // Element clicked, e.g., BUTTON
      console.log('currentTarget:', event.currentTarget.id); // The UL element 'menu'
    });
  </script>

</body>
</html>
```

6.4.4 Using Event Objects for Dynamic Behavior

Event objects allow you to react dynamically based on what element triggered the event or other event details:

6.4.5 Example: Highlighting Clicked Items

```
menu.addEventListener('click', (event) => {
  if (event.target.tagName === 'BUTTON') {
    alert(`You clicked: ${event.target.textContent}`);
    // Add a class to highlight the clicked button
    event.target.classList.add('highlight');
  }
});
```

This code checks the `event.target` to ensure only button clicks inside the menu trigger the action.

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Menu Button Highlight</title>
  <style>
    #menu {
      list-style: none;
      padding: 0;
      font-family: Arial, sans-serif;
      max-width: 200px;
    }
    #menu li {
      margin: 8px 0;
    }
    #menu button {
      width: 100%;
      padding: 8px 12px;
      cursor: pointer;
      font-size: 16px;
      transition: background-color 0.3s ease;
    }
    /* Highlight class */
    .highlight {
      background-color: yellow;
      font-weight: bold;
    }
  </style>
</head>
<body>

  <ul id="menu">
    <li><button>Home</button></li>
    <li><button>About</button></li>
    <li><button>Contact</button></li>
  </ul>

  <script>
    const menu = document.getElementById('menu');

    menu.addEventListener('click', (event) => {
      if (event.target.tagName === 'BUTTON') {
        console.log(`You clicked: ${event.target.textContent}`);
        // Add a class to highlight the clicked button
        event.target.classList.add('highlight');
      }
    });
  </script>

</body>
</html>
```

6.4.6 Other Useful Event Object Properties

- **`event.key`**: The key pressed during keyboard events.
- **`event.clientX`, `event.clientY`**: Mouse cursor coordinates relative to the viewport.
- **`event.shiftKey`, `event.ctrlKey`**: Boolean flags for modifier keys.

6.4.7 Summary

- Event handlers receive an **event object** describing what happened.
- `event.target` is the element that initiated the event.
- `event.currentTarget` is the element handling the event.
- You can prevent default browser behavior with `preventDefault()`.
- Use these properties and methods to write flexible, context-aware event handlers.

Mastering the event object unlocks powerful event-driven programming patterns in the DOM!

6.5 Practical Example: Efficient List Item Click Handling

Handling events on many dynamic elements can be tricky — especially if the elements are added or removed after the initial page load. Adding event listeners to each individual item is inefficient and can cause memory leaks.

Event delegation solves this by attaching a single listener to a common ancestor element that exists from the start. This listener can detect and respond to events triggered by any of its child elements, even if those children are added later dynamically.

6.5.1 Example: Delegated Click Handling on a Dynamic List

HTML:

```
<style>
  .selected {
    background-color: #d0f0c0;
  }
</style>

<h2>Click on a list item to select it</h2>
<ul id="itemList">
  <li data-id="1">Item 1</li>
  <li data-id="2">Item 2</li>
</ul>
```

```
  <button id="addItemBtn">Add New Item</button>
```

Javascript:

```
  <script>
    const list = document.getElementById('itemList');
    const addBtn = document.getElementById('addItemBtn');
    let nextId = 3;

    // Event delegation: single listener on the parent UL
    list.addEventListener('click', (event) => {
      // Only respond if a <li> was clicked (ignore clicks on the UL itself)
      if (event.target && event.target.nodeName === 'LI') {
        // Remove "selected" class from all items
        [...list.children].forEach(li => li.classList.remove('selected'));

        // Add "selected" class to the clicked item
        event.target.classList.add('selected');

        // Log the data-id of the clicked item
        console.log('Clicked item ID:', event.target.dataset.id);
      }
    });

    // Dynamically add new items to the list
    addBtn.addEventListener('click', () => {
      const newItem = document.createElement('li');
      newItem.textContent = `Item ${nextId}`;
      newItem.dataset.id = nextId;
      list.appendChild(newItem);
      nextId++;
    });
  </script>
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Delegated List Click Handling</title>
  <style>
    .selected {
      background-color: #d0f0c0;
    }
  </style>
</head>
<body>

  <h2>Click on a list item to select it</h2>
  <ul id="itemList">
    <li data-id="1">Item 1</li>
    <li data-id="2">Item 2</li>
  </ul>

  <button id="addItemBtn">Add New Item</button>

  <script>
```

```
    const list = document.getElementById('itemList');
    const addBtn = document.getElementById('addItemBtn');
    let nextId = 3;

    // Event delegation: single listener on the parent UL
    list.addEventListener('click', (event) => {
      // Only respond if a <li> was clicked (ignore clicks on the UL itself)
      if (event.target && event.target.nodeName === 'LI') {
        // Remove "selected" class from all items
        [...list.children].forEach(li => li.classList.remove('selected'));

        // Add "selected" class to the clicked item
        event.target.classList.add('selected');

        // Log the data-id of the clicked item
        console.log('Clicked item ID:', event.target.dataset.id);
      }
    });

    // Dynamically add new items to the list
    addBtn.addEventListener('click', () => {
      const newItem = document.createElement('li');
      newItem.textContent = `Item ${nextId}`;
      newItem.dataset.id = nextId;
      list.appendChild(newItem);
      nextId++;
    });
  </script>

</body>
</html>
```

6.5.2 How This Works

1. **Single Event Listener on the Parent (`ul`)** Instead of adding listeners to each `<li>`, we add one listener to the `<ul id="itemList">`. This listener catches **all click events bubbling up** from child `<li>` elements.
2. **Identifying the Clicked Item Using `event.target`** Inside the event handler, we check `event.target.nodeName === 'LI'` to make sure the click was on a list item, not the parent or any other element.
3. **Updating UI Efficiently** When an `<li>` is clicked, we first remove the `selected` class from all list items, then add it only to the clicked one. This provides clear visual feedback.
4. **Handling Dynamic Items** New items can be added anytime with the "Add New Item" button. Since the listener is on the parent `<ul>`, clicks on these new `<li>` elements are handled automatically without extra setup.

6.5.3 Benefits of This Pattern

- **Performance:** Only one event listener instead of many, reducing memory usage.
- **Maintainability:** No need to re-bind listeners when new elements are added dynamically.
- **Simplicity:** Centralized event logic instead of scattered handlers.

6.5.4 Extending This Pattern

- Delegate events for forms, tables, menus, or any dynamic list.
- Use more complex selectors or attribute checks inside the handler to filter events.
- Combine with `event.stopPropagation()` if you need to control event bubbling behavior more precisely.

Mastering event delegation helps you build responsive, efficient, and scalable web applications with less code and better performance!

Chapter 7.

Forms and User Input

1. Accessing Form Elements and Input Values
2. Validating User Input Programmatically
3. Handling Form Submission and Reset Events
4. Practical Example: Live Form Validation with Feedback

7 Forms and User Input

7.1 Accessing Form Elements and Input Values

Working with forms is a fundamental task in web development, and JavaScript provides several ways to access form elements and read user input. This section explains how to interact with form controls efficiently using the DOM.

7.1.1 Accessing Form Elements

Every `<form>` element in the DOM has a built-in `elements` property — a collection of all the form controls inside it. You can use this collection to access inputs, selects, checkboxes, radio buttons, and more by name or index.

7.1.2 Example HTML Form

```
<form id="myForm">
  <input type="text" name="username" value="JohnDoe" />
  <input type="checkbox" name="subscribe" checked />
  <input type="radio" name="gender" value="male" checked />
  <input type="radio" name="gender" value="female" />
  <select name="country">
    <option value="us" selected>United States</option>
    <option value="ca">Canada</option>
  </select>
</form>
```

7.1.3 Access via Form Element and `elements` Collection

```
const form = document.getElementById('myForm');

// Access inputs by name
const usernameInput = form.elements.username;
const subscribeCheckbox = form.elements.subscribe;
const genderRadios = form.elements.gender;  // Radio buttons with the same name grouped

const countrySelect = form.elements.country;
```

7.1.4 Reading Values from Different Form Controls

Text Inputs (`input type"text"`)

Use the `.value` property to read or write the input's current text.

```
console.log(usernameInput.value);  // Output: JohnDoe
```

Checkboxes (`input type"checkbox"`)

Check the `.checked` boolean property to see if the checkbox is selected.

```
console.log(subscribeCheckbox.checked);  // Output: true
```

Radio Buttons (`input type"radio"`)

Since radio buttons with the same `name` form a group, `form.elements.gender` returns a `RadioNodeList`. To find the selected radio value:

```
const selectedGender = genderRadios.value;  // "male"
console.log(selectedGender);
```

Alternatively, loop through the group:

```
for (const radio of genderRadios) {
  if (radio.checked) {
    console.log('Selected gender:', radio.value);
    break;
  }
}
```

Select Elements (`select`)

Use `.value` to get the selected option's value.

```
console.log(countrySelect.value);  // Output: us
```

To get the text of the selected option:

```
const selectedOptionText = countrySelect.options[countrySelect.selectedIndex].text;
console.log(selectedOptionText);  // Output: United States
```

7.1.5 Handling Nested and Complex Forms

If your form contains fieldsets or nested elements, you can still reliably access inputs through the `elements` collection by their `name` attribute.

For example:

```
<form id="complexForm">
  <fieldset>
    <input type="text" name="email" />
  </fieldset>
</form>
```

```
const complexForm = document.getElementById('complexForm');
console.log(complexForm.elements.email.value);
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Form Elements Access</title>
</head>
<body>

  <form id="complexForm">
    <fieldset>
      <input type="text" name="email" placeholder="Enter your email" />
    </fieldset>
  </form>

  <script>
    const complexForm = document.getElementById('complexForm');
    // Log the value of the input named "email" (initially empty)
    console.log('Email input value:', complexForm.elements.email.value);

    // Optional: log value whenever it changes
    complexForm.elements.email.addEventListener('input', (event) => {
      console.log('Email input changed to:', event.target.value);
    });
  </script>

</body>
</html>
```

7.1.6 Best Practices

- **Use `name` Attributes:** Always assign meaningful and unique `name` attributes to inputs to simplify access.
- **Avoid IDs for Form Controls:** While IDs can be used, `elements[name]` is often simpler and groups related controls better.
- **Handle Missing or Default Values:** Inputs may be empty or unchecked. Always check for undefined or empty values before processing.
- **Sanitize User Input:** Never trust user input blindly — validate and sanitize as needed before using or sending it to servers.

7.1.7 Summary

- Use the form's `.elements` collection to access inputs by name.
- Different form controls expose their values differently: `.value` for text, select, and radio groups; `.checked` for checkboxes and radios.
- For radio buttons, use the group's `.value` or iterate to find the checked input.
- Always handle edge cases where inputs may be missing or unselected.

Mastering how to access and read form inputs programmatically lays the foundation for building interactive, dynamic forms in JavaScript. Next, we will explore validating this input to ensure data correctness!

7.2 Validating User Input Programmatically

Validating user input on the client side is essential for providing a smooth user experience and preventing invalid data from being submitted. JavaScript offers powerful techniques to enforce rules before the form is sent to the server.

7.2.1 Why Validate on the Client Side?

- **Immediate feedback:** Users can fix errors instantly without waiting for server responses.
- **Reduced server load:** Avoid unnecessary requests caused by invalid data.
- **Improved data quality:** Enforce business rules consistently.

7.2.2 Common Validation Checks

- **Required fields:** Ensure essential inputs are not empty.
- **Length constraints:** Check minimum and maximum length for strings.
- **Format patterns:** Validate formats like email addresses, phone numbers, or zip codes.
- **Custom rules:** Enforce specific conditions like password complexity or matching fields.

7.2.3 Using the Constraint Validation API

HTML5 provides a built-in Constraint Validation API that works seamlessly with JavaScript. It allows you to check validity and display browser default error messages or customize your

own.

7.2.4 Example: Basic Constraint Validation

```
<form id="signupForm">
  <input type="email" id="email" name="email" required placeholder="Email" />
  <input type="password" id="password" name="password" required minlength="6" placeholder="Password" />
  <button type="submit">Sign Up</button>
</form>

<div id="errorMessages" style="color: red;"></div>
```

```
const form = document.getElementById('signupForm');
const errorDiv = document.getElementById('errorMessages');

form.addEventListener('submit', function(event) {
  errorDiv.textContent = ''; // Clear previous errors

  if (!form.checkValidity()) {
    event.preventDefault();  // Stop form submission

    // Display custom messages
    if (!form.email.validity.valid) {
      errorDiv.textContent += 'Please enter a valid email address.\n';
    }

    if (!form.password.validity.valid) {
      if (form.password.validity.valueMissing) {
        errorDiv.textContent += 'Password is required.\n';
      }
      if (form.password.validity.tooShort) {
        errorDiv.textContent += 'Password must be at least 6 characters.\n';
      }
    }
  }
});
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Signup Form Validation</title>
  <style>
    #signupForm {
      max-width: 300px;
      font-family: Arial, sans-serif;
      display: flex;
      flex-direction: column;
      gap: 10px;
    }
    #errorMessages {
```

```
      color: red;
      white-space: pre-line; /* preserve newlines */
      margin-top: 10px;
      font-weight: bold;
    }
    input, button {
      padding: 8px;
      font-size: 16px;
    }
  </style>
</head>
<body>

  <form id="signupForm" novalidate>
    <input type="email" id="email" name="email" required placeholder="Email" />
    <input type="password" id="password" name="password" required minlength="6" placeholder="Password" />
    <button type="submit">Sign Up</button>
  </form>

  <div id="errorMessages"></div>

  <script>
    const form = document.getElementById('signupForm');
    const errorDiv = document.getElementById('errorMessages');

    form.addEventListener('submit', function(event) {
      errorDiv.textContent = ''; // Clear previous errors

      if (!form.checkValidity()) {
        event.preventDefault();  // Stop form submission

        // Display custom messages
        if (!form.email.validity.valid) {
          errorDiv.textContent += 'Please enter a valid email address.\n';
        }

        if (!form.password.validity.valid) {
          if (form.password.validity.valueMissing) {
            errorDiv.textContent += 'Password is required.\n';
          }
          if (form.password.validity.tooShort) {
            errorDiv.textContent += 'Password must be at least 6 characters.\n';
          }
        }
      }
    });
  </script>

</body>
</html>
```

7.2.5 Custom Validation Logic

Sometimes built-in validation isn't enough. You can create custom functions to enforce rules and show error messages dynamically.

7.2.6 Example: Validate Phone Number Format

```
<input type="tel" id="phone" name="phone" placeholder="Phone (e.g., 123-456-7890)" />
<span id="phoneError" style="color: red;"></span>
```

```
const phoneInput = document.getElementById('phone');
const phoneError = document.getElementById('phoneError');

phoneInput.addEventListener('input', function() {
  const phonePattern = /^\d{3}-\d{3}-\d{4}$/;
  if (phoneInput.value && !phonePattern.test(phoneInput.value)) {
    phoneError.textContent = 'Phone number must be in the format 123-456-7890.';
  } else {
    phoneError.textContent = '';
  }
});
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Phone Input Validation</title>
  <style>
    #phone {
      padding: 8px;
      font-size: 16px;
      width: 200px;
      margin-right: 10px;
    }
    #phoneError {
      color: red;
      font-family: Arial, sans-serif;
      font-size: 14px;
    }
  </style>
</head>
<body>

  <label for="phone">Phone:</label>
  <input type="tel" id="phone" name="phone" placeholder="Phone (e.g., 123-456-7890)" />
  <span id="phoneError"></span>

  <script>
    const phoneInput = document.getElementById('phone');
    const phoneError = document.getElementById('phoneError');
```

```
    phoneInput.addEventListener('input', function() {
      const phonePattern = /^\d{3}-\d{3}-\d{4}$/;
      if (phoneInput.value && !phonePattern.test(phoneInput.value)) {
        phoneError.textContent = 'Phone number must be in the format 123-456-7890.';
      } else {
        phoneError.textContent = '';
      }
    });
  </script>

</body>
</html>
```

7.2.7 Live Validation on Input

Validating fields as users type or change focus improves usability by giving immediate feedback.

```
const passwordInput = document.getElementById('password');

passwordInput.addEventListener('input', function() {
  if (passwordInput.value.length < 6) {
    passwordInput.setCustomValidity('Password must be at least 6 characters long.');
  } else {
    passwordInput.setCustomValidity(''); // Clear error
  }
});
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Password Validation</title>
  <style>
    form {
      max-width: 300px;
      font-family: Arial, sans-serif;
      margin-top: 20px;
    }
    input {
      width: 100%;
      padding: 8px;
      font-size: 16px;
      margin-bottom: 10px;
      box-sizing: border-box;
    }
  </style>
</head>
<body>

  <form>
    <label for="password">Password (min 6 chars):</label><br />
```

```
    <input type="password" id="password" name="password" required />
    <button type="submit">Submit</button>
  </form>

  <script>
    const passwordInput = document.getElementById('password');

    passwordInput.addEventListener('input', function() {
      if (passwordInput.value.length < 6) {
        passwordInput.setCustomValidity('Password must be at least 6 characters long.');
      } else {
        passwordInput.setCustomValidity(''); // Clear error
      }
    });
  </script>

</body>
</html>
```

7.2.8 Summary

- Use the **Constraint Validation API** to leverage built-in HTML validation rules.
- Add **custom validation logic** when necessary to enforce complex rules.
- Validate **on form submission** and **on-the-fly** to enhance UX.
- Display clear, user-friendly **error messages** dynamically.

Mastering these techniques ensures your forms guide users to enter valid, high-quality data, leading to better applications and happier users! Next, we'll explore handling form submission and reset events effectively.

7.3 Handling Form Submission and Reset Events

Forms are a primary way users interact with web pages. Understanding how to control form submission and reset events allows you to create responsive, user-friendly interfaces that avoid unnecessary page reloads and provide immediate feedback.

7.3.1 Intercepting Form Submission

By default, submitting a form causes the browser to reload or navigate away from the page to the form's `action` URL. To create dynamic experiences—such as sending data asynchronously or validating before sending—you need to intercept the submission event.

You can do this by attaching an event listener to the form's `submit` event and calling `event.preventDefault()` to stop the default page reload.

7.3.2 Example: Prevent Default Submission and Log Data

```
<form id="contactForm">
  <input type="text" name="name" placeholder="Your Name" required />
  <input type="email" name="email" placeholder="Your Email" required />
  <button type="submit">Send</button>
</form>
<div id="status"></div>
```

```
const form = document.getElementById('contactForm');
const statusDiv = document.getElementById('status');

form.addEventListener('submit', function(event) {
  event.preventDefault();  // Prevent the default form submission

  const formData = new FormData(form);  // Collect form data
  const data = Object.fromEntries(formData.entries());

  // Log data or send via AJAX
  console.log('Form data submitted:', data);

  statusDiv.textContent = 'Thank you for submitting the form!';

  // Optionally, reset the form after submission
  form.reset();
});
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Contact Form Submission</title>
  <style>
    #contactForm {
      max-width: 300px;
      font-family: Arial, sans-serif;
      display: flex;
      flex-direction: column;
      gap: 10px;
    }
    input, button {
      padding: 8px;
      font-size: 16px;
      box-sizing: border-box;
    }
    #status {
      margin-top: 15px;
      font-weight: bold;
```

```
      color: green;
      font-family: Arial, sans-serif;
    }
  </style>
</head>
<body>

  <form id="contactForm" novalidate>
    <input type="text" name="name" placeholder="Your Name" required />
    <input type="email" name="email" placeholder="Your Email" required />
    <button type="submit">Send</button>
  </form>

  <div id="status"></div>

  <script>
    const form = document.getElementById('contactForm');
    const statusDiv = document.getElementById('status');

    form.addEventListener('submit', function(event) {
      event.preventDefault();  // Prevent the default form submission

      if (!form.checkValidity()) {
        // If form is invalid, show built-in validation UI
        form.reportValidity();
        return;
      }

      const formData = new FormData(form);  // Collect form data
      const data = Object.fromEntries(formData.entries());

      // Log data or send via AJAX
      console.log('Form data submitted:', data);

      statusDiv.textContent = 'Thank you for submitting the form!';

      // Optionally, reset the form after submission
      form.reset();
    });
  </script>

</body>
</html>
```

7.3.3 Processing Form Data Without Reloads (AJAX-style Submission)

Using JavaScript's Fetch API or XMLHttpRequest, you can send form data to the server asynchronously, allowing the page to remain visible and interactive.

7.3.4 Example: Sending Form Data via Fetch

```
form.addEventListener('submit', async (event) => {
  event.preventDefault();

  const formData = new FormData(form);

  try {
    const response = await fetch('/submit-form', {
      method: 'POST',
      body: formData
    });

    if (response.ok) {
      statusDiv.textContent = 'Form submitted successfully!';
      form.reset();
    } else {
      statusDiv.textContent = 'Submission failed. Please try again.';
    }
  } catch (error) {
    statusDiv.textContent = 'Network error. Please check your connection.';
  }
});
```

Note: This example assumes a server endpoint at `/submit-form` that handles the form data.

7.3.5 Handling Form Reset Events

The `reset` event fires when a form is reset, either by user action on a reset button or programmatically via `form.reset()`. You can listen for this event to customize what happens when the form clears.

7.3.6 Example: Custom Reset Behavior

```
<form id="profileForm">
  <input type="text" name="username" value="JohnDoe" />
  <input type="email" name="email" value="john@example.com" />
  <button type="reset">Clear</button>
</form>
```

```
const profileForm = document.getElementById('profileForm');

profileForm.addEventListener('reset', (event) => {
  // Show a message or perform additional cleanup
```

```
  console.log('Form has been reset to default values.');
});
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Form Reset Event</title>
  <style>
    form {
      max-width: 300px;
      font-family: Arial, sans-serif;
      display: flex;
      flex-direction: column;
      gap: 10px;
      margin-top: 20px;
    }
    input, button {
      padding: 8px;
      font-size: 16px;
      box-sizing: border-box;
    }
  </style>
</head>
<body>

  <form id="profileForm">
    <input type="text" name="username" value="JohnDoe" />
    <input type="email" name="email" value="john@example.com" />
    <button type="reset">Clear</button>
  </form>

  <script>
    const profileForm = document.getElementById('profileForm');

    profileForm.addEventListener('reset', (event) => {
      console.log('Form has been reset to default values.');
    });
  </script>

</body>
</html>
```

You can also programmatically reset form fields to custom values if needed:

```
function resetToCustomDefaults(form) {
  form.username.value = 'DefaultUser';
  form.email.value = 'default@example.com';
}

profileForm.addEventListener('reset', (event) => {
  event.preventDefault(); // Prevent default reset
  resetToCustomDefaults(profileForm); // Apply custom reset logic
});
```

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Custom Form Reset</title>
  <style>
    form {
      max-width: 300px;
      font-family: Arial, sans-serif;
      display: flex;
      flex-direction: column;
      gap: 10px;
      margin-top: 20px;
    }
    input, button {
      padding: 8px;
      font-size: 16px;
      box-sizing: border-box;
    }
  </style>
</head>
<body>

  <form id="profileForm">
    <input type="text" name="username" value="JohnDoe" />
    <input type="email" name="email" value="john@example.com" />
    <button type="reset">Clear</button>
  </form>

  <script>
    const profileForm = document.getElementById('profileForm');

    function resetToCustomDefaults(form) {
      form.username.value = 'DefaultUser';
      form.email.value = 'default@example.com';
    }

    profileForm.addEventListener('reset', (event) => {
      event.preventDefault(); // Prevent default reset
      resetToCustomDefaults(profileForm); // Apply custom reset logic
      console.log('Form reset to custom default values.');
    });
  </script>

</body>
</html>
```

7.3.7 Summary

- Use `submit` event listeners with `event.preventDefault()` to stop page reload and handle data dynamically.
- Process form data with the `FormData` API and send asynchronously using `fetch()` or other methods.

- Listen for `reset` events to customize behavior when the form clears.
- Combine these techniques for responsive, modern form handling that improves user experience.

In the next section, you'll see how to put these concepts together for a live form validation example with user feedback!

7.4 Practical Example: Live Form Validation with Feedback

Building a form that validates user input **live** — as the user types or changes fields — significantly improves user experience by catching errors early and guiding users in real time. This section walks you through creating a simple, interactive form with live validation and feedback.

7.4.1 Complete Live Validation Example

Here's a form that validates a user's **name** and **email** as they type, showing helpful messages below each input field.

```
<form id="signupForm" novalidate>
  <label>
    Name:
    <input type="text" id="name" name="name" required />
    <small class="error-message"></small>
  </label>

  <label>
    Email:
    <input type="email" id="email" name="email" required />
    <small class="error-message"></small>
  </label>

  <button type="submit">Submit</button>
  <div id="formStatus"></div>
</form>

<style>
  input {
    display: block;
    margin-bottom: 5px;
  }
  .error-message {
    color: red;
    font-size: 0.9em;
    height: 1.2em;
  }
  input.invalid {
```

```
    border-color: red;
  }
  input.valid {
    border-color: green;
  }
  #formStatus {
    margin-top: 10px;
    font-weight: bold;
  }
</style>
```

```
const form = document.getElementById('signupForm');
const nameInput = document.getElementById('name');
const emailInput = document.getElementById('email');
const formStatus = document.getElementById('formStatus');

// Utility function to show or clear error messages
function setError(input, message) {
  const errorMsg = input.nextElementSibling;
  if (message) {
    errorMsg.textContent = message;
    input.classList.add('invalid');
    input.classList.remove('valid');
  } else {
    errorMsg.textContent = '';
    input.classList.remove('invalid');
    input.classList.add('valid');
  }
}

// Validation functions
function validateName(name) {
  if (!name.trim()) return 'Name is required.';
  if (name.length < 3) return 'Name must be at least 3 characters.';
  return '';
}

function validateEmail(email) {
  if (!email.trim()) return 'Email is required.';
  // Simple email regex pattern for demo purposes
  const emailPattern = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
  if (!emailPattern.test(email)) return 'Please enter a valid email address.';
  return '';
}

// Live validation event handlers
nameInput.addEventListener('input', () => {
  const error = validateName(nameInput.value);
  setError(nameInput, error);
});

emailInput.addEventListener('input', () => {
  const error = validateEmail(emailInput.value);
  setError(emailInput, error);
});

// Handle form submission
```

```
form.addEventListener('submit', (event) => {
  event.preventDefault();

  const nameError = validateName(nameInput.value);
  const emailError = validateEmail(emailInput.value);

  setError(nameInput, nameError);
  setError(emailInput, emailError);

  if (!nameError && !emailError) {
    formStatus.textContent = 'Form submitted successfully!';
    formStatus.style.color = 'green';
    form.reset();
    nameInput.classList.remove('valid');
    emailInput.classList.remove('valid');
  } else {
    formStatus.textContent = 'Please fix the errors above.';
    formStatus.style.color = 'red';
  }
});
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Signup Form Validation</title>
  <style>
    input {
      display: block;
      margin-bottom: 5px;
      padding: 6px;
      font-size: 16px;
      width: 250px;
      box-sizing: border-box;
      border: 2px solid #ccc;
      border-radius: 4px;
      transition: border-color 0.3s;
    }
    .error-message {
      color: red;
      font-size: 0.9em;
      height: 1.2em;
      margin-bottom: 10px;
      font-family: Arial, sans-serif;
    }
    input.invalid {
      border-color: red;
    }
    input.valid {
      border-color: green;
    }
    #formStatus {
      margin-top: 10px;
      font-weight: bold;
      font-family: Arial, sans-serif;
    }
    label {
```

```
      font-family: Arial, sans-serif;
      margin-bottom: 10px;
      display: block;
    }
    button {
      padding: 8px 16px;
      font-size: 16px;
      cursor: pointer;
      border-radius: 4px;
      border: none;
      background-color: #007BFF;
      color: white;
      transition: background-color 0.3s ease;
    }
    button:hover {
      background-color: #0056b3;
    }
  </style>
</head>
<body>

  <form id="signupForm" novalidate>
    <label>
      Name:
      <input type="text" id="name" name="name" required />
      <small class="error-message"></small>
    </label>

    <label>
      Email:
      <input type="email" id="email" name="email" required />
      <small class="error-message"></small>
    </label>

    <button type="submit">Submit</button>
    <div id="formStatus"></div>
  </form>

  <script>
    const form = document.getElementById('signupForm');
    const nameInput = document.getElementById('name');
    const emailInput = document.getElementById('email');
    const formStatus = document.getElementById('formStatus');

    // Utility function to show or clear error messages
    function setError(input, message) {
      const errorMsg = input.nextElementSibling;
      if (message) {
        errorMsg.textContent = message;
        input.classList.add('invalid');
        input.classList.remove('valid');
      } else {
        errorMsg.textContent = '';
        input.classList.remove('invalid');
        input.classList.add('valid');
      }
    }
```

```
    // Validation functions
    function validateName(name) {
      if (!name.trim()) return 'Name is required.';
      if (name.length < 3) return 'Name must be at least 3 characters.';
      return '';
    }

    function validateEmail(email) {
      if (!email.trim()) return 'Email is required.';
      // Simple email regex pattern for demo purposes
      const emailPattern = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
      if (!emailPattern.test(email)) return 'Please enter a valid email address.';
      return '';
    }

    // Live validation event handlers
    nameInput.addEventListener('input', () => {
      const error = validateName(nameInput.value);
      setError(nameInput, error);
    });

    emailInput.addEventListener('input', () => {
      const error = validateEmail(emailInput.value);
      setError(emailInput, error);
    });

    // Handle form submission
    form.addEventListener('submit', (event) => {
      event.preventDefault();

      const nameError = validateName(nameInput.value);
      const emailError = validateEmail(emailInput.value);

      setError(nameInput, nameError);
      setError(emailInput, emailError);

      if (!nameError && !emailError) {
        formStatus.textContent = 'Form submitted successfully!';
        formStatus.style.color = 'green';
        form.reset();
        nameInput.classList.remove('valid');
        emailInput.classList.remove('valid');
      } else {
        formStatus.textContent = 'Please fix the errors above.';
        formStatus.style.color = 'red';
      }
    });
  </script>

</body>
</html>
```

7.4.2 How It Works

- **Live input validation:** The `input` event triggers on each keystroke, running validation and updating error messages immediately.
- **Clear error/success feedback:** The `setError` helper updates the small text under inputs and toggles red/green borders.
- **Submission validation:** On submit, it validates both fields again to ensure no errors before proceeding.
- **User-friendly:** Error messages guide users on what to fix without page reloads.
- **Visual feedback:** Input borders change color dynamically to show valid or invalid states.

7.4.3 Managing Error States Cleanly

- Each input's error message is shown in a dedicated `<small>` tag.
- The border color communicates validity without relying solely on text.
- Feedback messages update promptly on user input or form submission.
- The submit status message informs users if the entire form is valid or requires correction.

7.4.4 Encouragement to Extend

You can enhance this example by:

- Adding **more fields** (password, phone, etc.) with their own validation.
- Using **custom validation rules**, like checking password strength or matching confirm fields.
- Improving **accessibility** by linking error messages with `aria-describedby` and managing focus.
- Implementing **debouncing** to avoid validating on every keystroke for performance.
- Styling feedback more elaborately or animating error messages for better UX.

7.4.5 Summary

This live validation form ties together event listening, input value access, validation logic, and DOM updates — all crucial skills for working effectively with forms. Experiment with adding new rules and controls to deepen your mastery of DOM and JavaScript interaction!

Ready to build even more interactive web forms? Let's move on!

Chapter 8.

Working with CSS and Computed Styles

1. Reading and Modifying Inline Styles
2. Accessing Computed Styles with `getComputedStyle()`
3. Dynamic Style Changes and Transitions
4. Practical Example: Theme Switcher with Smooth Transitions

8 Working with CSS and Computed Styles

8.1 Reading and Modifying Inline Styles

When working with CSS in JavaScript, it's important to understand the differences between **inline styles**, **stylesheets**, and **computed styles** — and how to interact with each.

8.1.1 Understanding Styles: Inline vs Stylesheets vs Computed Styles

- **Inline styles** are CSS rules applied directly on an element via its `style` attribute, e.g., `<div style="color: red;">`.
- **Stylesheets** define CSS rules in `<style>` tags or external `.css` files and apply styles based on selectors.
- **Computed styles** represent the final, resolved styles after combining inline styles, stylesheets, and browser defaults.

Why this matters: Inline styles override stylesheets for the same properties, but modifying stylesheets affects many elements at once. Computed styles help us see the final actual style applied.

8.1.2 Reading Inline Styles with the `style` Property

You can access an element's inline styles through its `.style` property in JavaScript. This property gives you a **CSSStyleDeclaration** object representing inline styles only.

```
const box = document.getElementById('box');
console.log(box.style.color);      // e.g., "red"
console.log(box.style.width);      // e.g., "100px"
```

If the style is not set inline, accessing `.style` for that property returns an empty string.

8.1.3 Modifying Inline Styles Dynamically

You can assign CSS property values via `.style`:

```
// Select the element
const box = document.getElementById('box');

// Change the background color
box.style.backgroundColor = 'lightblue';
```

```
// Set the width and height
box.style.width = '200px';
box.style.height = '150px';

// Position the element
box.style.position = 'absolute';
box.style.left = '50px';
box.style.top = '100px';
```

Note that when setting styles in JavaScript, use **camelCase** property names instead of CSS hyphenated names (`background-color` → `backgroundColor`).

8.1.4 Example: Changing Color and Size on Button Click

```
<button id="changeBtn">Change Box Style</button>
<div id="box" style="width: 100px; height: 100px; background-color: coral;"></div>

<script>
  const btn = document.getElementById('changeBtn');
  const box = document.getElementById('box');

  btn.addEventListener('click', () => {
    box.style.backgroundColor = 'mediumseagreen';
    box.style.width = '150px';
    box.style.height = '150px';
  });
</script>
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Change Box Style</title>
  <style>
    #box {
      width: 100px;
      height: 100px;
      background-color: coral;
      transition: all 0.3s ease;
      margin-top: 10px;
    }
    #changeBtn {
      padding: 8px 16px;
      font-size: 16px;
      cursor: pointer;
    }
  </style>
</head>
<body>

  <button id="changeBtn">Change Box Style</button>
```

```
  <div id="box"></div>

  <script>
    const btn = document.getElementById('changeBtn');
    const box = document.getElementById('box');

    btn.addEventListener('click', () => {
      box.style.backgroundColor = 'mediumseagreen';
      box.style.width = '150px';
      box.style.height = '150px';
    });
  </script>

</body>
</html>
```

8.1.5 Limitations of Inline Styles

- Inline styles only affect **one element** at a time.
- They can quickly become **hard to maintain** if you apply many styles directly.
- Inline styles **override** stylesheets, which can make debugging complex.
- Some CSS features (like pseudo-classes `:hover`) **cannot be controlled via inline styles**.

8.1.6 When to Use CSS Classes Instead

For better maintainability and performance:

- Use JavaScript to **add, remove, or toggle CSS classes** with `.classList` to apply multiple styles defined in stylesheets.
- Use inline styles for **dynamic, one-off style changes** or computed values.
- Avoid cluttering HTML with many inline styles.

Example:

```
box.classList.add('highlighted');  // Apply styles defined in CSS
```

8.1.7 Summary

- The `.style` property reads and modifies **inline styles** only.
- Inline styles have the **highest specificity** but limited flexibility.
- For dynamic styling, inline styles are useful for single or computed style changes.

- For larger or reusable styling, prefer CSS classes and stylesheet rules.

Next, we'll learn how to read **computed styles**—the actual styles an element has after applying all CSS rules from multiple sources.

8.2 Accessing Computed Styles with `getComputedStyle()`

When working with styles in JavaScript, it's often necessary to know the **final styles** that an element has on the page—not just the inline styles but also those applied via CSS rulesheets, inherited styles, and browser defaults. This is where the **`getComputedStyle()`** function becomes invaluable.

8.2.1 What is `getComputedStyle()`?

`getComputedStyle()` is a browser API method that returns a **live, read-only CSSStyleDeclaration object** representing the **computed styles** for a given element. Computed styles are the actual values applied after resolving all CSS rules, including stylesheets, inline styles, and inherited styles.

Syntax:

```
const styles = window.getComputedStyle(element);
```

- `element`: The DOM element whose computed styles you want to retrieve.
- `styles`: An object containing all CSS properties with their computed values.

8.2.2 Understanding the Computed Styles Object

The returned object behaves like a dictionary of CSS properties, where you can query properties as camelCase or hyphenated strings:

```
const color = styles.color;           // e.g., "rgb(255, 0, 0)"
const marginTop = styles.marginTop;   // e.g., "16px"
const fontSize = styles.getPropertyValue('font-size'); // e.g., "14px"
```

The values are always **resolved** and often expressed in absolute units (e.g., pixels), which makes them reliable for measurement and logic.

8.2.3 Practical Use Cases for `getComputedStyle()`

1. **Determine element dimensions before manipulation**

If you need to animate or resize an element, reading its computed width and height is essential:

```
const box = document.getElementById('box');
const computedStyles = window.getComputedStyle(box);
const width = computedStyles.width;   // e.g., "200px"
const height = computedStyles.height; // e.g., "150px"
console.log(`Box dimensions: ${width} x ${height}`);
```

2. **Check the actual applied color, font, or visibility**

Even if an element doesn't have inline styles, you can check the effective styles:

```
console.log('Background color:', computedStyles.backgroundColor);
console.log('Display:', computedStyles.display);
```

3. **Read properties that cannot be accessed directly from `.style`**

For example, `element.style.color` might be empty if the color comes from a stylesheet. `getComputedStyle()` gives you the resolved value.

8.2.4 Example: Comparing Inline vs Computed Styles

```
<style>
  #myText {
    color: blue;
    font-size: 18px;
  }
</style>

<p id="myText" style="color: red;">Hello World!</p>

<script>
  const el = document.getElementById('myText');

  // Inline style
  console.log('Inline color:', el.style.color); // "red"

  // Computed style
  const computed = window.getComputedStyle(el);
  console.log('Computed color:', computed.color); // "red" (inline overrides stylesheet)
  console.log('Computed font-size:', computed.fontSize); // "18px" (from stylesheet)
</script>
```

In this example:

- The inline style sets `color` to red, which overrides the stylesheet's blue.

- The `font-size` is from the stylesheet, so it only appears in computed styles.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Inline vs Computed Style</title>
  <style>
    #myText {
      color: blue;
      font-size: 18px;
    }
  </style>
</head>
<body>

  <p id="myText" style="color: red;">Hello World!</p>

  <script>
    const el = document.getElementById('myText');

    // Inline style
    console.log('Inline color:', el.style.color); // "red"

    // Computed style
    const computed = window.getComputedStyle(el);
    console.log('Computed color:', computed.color); // "red" (inline overrides stylesheet)
    console.log('Computed font-size:', computed.fontSize); // "18px" (from stylesheet)
  </script>

</body>
</html>
```

8.2.5 Notes and Best Practices

- `getComputedStyle()` returns all CSS properties, even those not explicitly set.
- The returned object is **read-only** — you cannot change styles via `getComputedStyle()`.
- Values are often normalized, e.g., colors returned as `rgb()` strings and lengths in pixels.
- Use it to **read** current styles before applying dynamic changes or animations.

8.2.6 Summary

- Use `window.getComputedStyle(element)` to get the **final applied styles** of any element.
- It complements `.style`, which only accesses inline styles.
- This method is critical for **measuring**, **condition-based styling**, or understanding how CSS affects elements in real time.

In the next section, we'll learn how to apply **dynamic style changes and transitions** smoothly using JavaScript.

8.3 Dynamic Style Changes and Transitions

Adding smooth, visually appealing transitions and animations can greatly enhance the user experience of your web page. In JavaScript, you can trigger these effects by **changing inline styles or CSS classes dynamically**, causing the browser to animate property changes.

8.3.1 Triggering CSS Transitions Programmatically

CSS transitions let you animate changes to CSS properties over a specified duration. You set these transitions in CSS, then trigger them by changing the properties via JavaScript.

8.3.2 How it works:

1. Define the transition in CSS:

```
.box {
  width: 100px;
  height: 100px;
  background-color: blue;
  transition: background-color 0.5s ease, width 0.5s ease;
}
```

2. Change the property in JavaScript:

```
const box = document.querySelector('.box');
box.style.backgroundColor = 'red';
box.style.width = '200px';
```

The browser automatically animates the changes smoothly.

8.3.3 Using Classes to Control Transitions

Often, managing transitions by toggling CSS classes is cleaner and more maintainable. For example:

```css
.box {
  width: 100px;
  height: 100px;
  background-color: blue;
  transition: background-color 0.5s ease, width 0.5s ease;
}

.box.active {
  background-color: red;
  width: 200px;
}
```

Toggle the class in JavaScript:

```javascript
box.classList.toggle('active');
```

This triggers the transition on the affected properties.

8.3.4 Best Practices for Smooth Transitions

- **Choose properties wisely**: Properties like `transform` and `opacity` are GPU-accelerated and generally perform better than properties like `width` or `height`.
- **Use hardware acceleration hints**: Using CSS like `transform: translateZ(0)` can sometimes trigger GPU acceleration for smoother animations.
- **Keep transitions short and responsive**: Avoid very long or complex transitions that might frustrate users.
- **Define easing functions** (`ease`, `ease-in-out`, `linear`, `cubic-bezier`) for natural-looking animations.

8.3.5 Listening for Transition End Events

Sometimes you want to perform actions after a transition completes, like showing a message or triggering another animation.

Use the `transitionend` event:

```javascript
box.addEventListener('transitionend', (event) => {
  console.log(`Transition for ${event.propertyName} completed.`);
  // Additional actions here
});
```

Note: When multiple properties are transitioned, `transitionend` fires once for each.

8.3.6 Example: Toggling a Theme with Transitions

Here's a runnable example that toggles a light/dark theme on a container with smooth background color and text color transitions:

```
<style>
  .container {
    width: 300px;
    height: 150px;
    background-color: white;
    color: black;
    padding: 20px;
    transition: background-color 0.6s ease, color 0.6s ease;
    border: 1px solid #ccc;
  }

  .dark-theme {
    background-color: #222;
    color: #eee;
  }

  button {
    margin-top: 20px;
  }
</style>

<div class="container" id="themeBox">
  This box changes theme smoothly!
</div>

<button id="toggleBtn">Toggle Theme</button>

<script>
  const box = document.getElementById('themeBox');
  const button = document.getElementById('toggleBtn');

  button.addEventListener('click', () => {
    box.classList.toggle('dark-theme');
  });

  box.addEventListener('transitionend', (e) => {
    if (e.propertyName === 'background-color') {
      console.log('Background color transition ended.');
    }
  });
</script>
```

Try clicking the button to see the background and text colors transition smoothly.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Theme Toggle Example</title>
  <style>
    .container {
```

```
      width: 300px;
      height: 150px;
      background-color: white;
      color: black;
      padding: 20px;
      transition: background-color 0.6s ease, color 0.6s ease;
      border: 1px solid #ccc;
      font-family: Arial, sans-serif;
    }

    .dark-theme {
      background-color: #222;
      color: #eee;
    }

    button {
      margin-top: 20px;
      padding: 8px 16px;
      font-size: 16px;
      cursor: pointer;
    }
  </style>
</head>
<body>

  <div class="container" id="themeBox">
    This box changes theme smoothly!
  </div>

  <button id="toggleBtn">Toggle Theme</button>

  <script>
    const box = document.getElementById('themeBox');
    const button = document.getElementById('toggleBtn');

    button.addEventListener('click', () => {
      box.classList.toggle('dark-theme');
    });

    box.addEventListener('transitionend', (e) => {
      if (e.propertyName === 'background-color') {
        console.log('Background color transition ended.');
      }
    });
  </script>

</body>
</html>
```

8.3.7 Summary

- CSS transitions can be triggered by changing inline styles or toggling classes.
- Use hardware-accelerated properties like `transform` and `opacity` for better perfor-

mance.

- Listen for `transitionend` to coordinate UI changes after animations.
- Managing transitions via CSS classes keeps your code clean and maintainable.
- Practical animations can enhance user experience with minimal code.

Next, we'll build a complete **theme switcher** example with smooth transitions combining these concepts.

8.4 Practical Example: Theme Switcher with Smooth Transitions

Creating a **theme switcher** is a great way to apply what we've learned about dynamic style changes and CSS transitions. In this example, we'll build a button that toggles the page between **light** and **dark** themes with smooth animated transitions.

8.4.1 Step 1: HTML Structure

We'll create a simple page with some content and a toggle button:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1" />
  <title>Theme Switcher</title>
  <link rel="stylesheet" href="styles.css" />
</head>
<body>
  <div class="container">
    <h1>Welcome to the Theme Switcher!</h1>
    <p>This example toggles between light and dark themes smoothly.</p>
    <button id="themeToggle">Toggle Theme</button>
  </div>

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

8.4.2 Step 2: CSS with Transitions and Themes

Define styles for light and dark modes, and add transitions for smooth visual effects:

```css
/* styles.css */
body {
  margin: 0;
  font-family: Arial, sans-serif;
  transition: background-color 0.6s ease, color 0.6s ease;
  background-color: #fff;
  color: #222;
}

body.dark-theme {
  background-color: #121212;
  color: #eee;
}

.container {
  max-width: 600px;
  margin: 100px auto;
  padding: 20px;
  text-align: center;
}

button {
  padding: 10px 20px;
  font-size: 1rem;
  cursor: pointer;
  border: none;
  border-radius: 4px;
  background-color: #007acc;
  color: white;
  transition: background-color 0.3s ease;
}

button:hover {
  background-color: #005fa3;
}

/* Accessibility: respect user's reduced motion preference */
@media (prefers-reduced-motion: reduce) {
  * {
    transition: none !important;
  }
}
```

8.4.3 Step 3: JavaScript for Theme Toggling

Add interactivity to toggle the theme class on the `body` element:

```javascript
// script.js

const toggleButton = document.getElementById('themeToggle');
const body = document.body;

// Load saved user preference (optional, see notes below)
```

```
const savedTheme = localStorage.getItem('theme');
if (savedTheme) {
  body.classList.toggle('dark-theme', savedTheme === 'dark');
}

toggleButton.addEventListener('click', () => {
  const isDark = body.classList.toggle('dark-theme');
  // Save user preference to localStorage (link to storage chapter)
  localStorage.setItem('theme', isDark ? 'dark' : 'light');
});
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1" />
  <title>Theme Switcher</title>
  <style>
    body {
      margin: 0;
      font-family: Arial, sans-serif;
      transition: background-color 0.6s ease, color 0.6s ease;
      background-color: #fff;
      color: #222;
    }

    body.dark-theme {
      background-color: #121212;
      color: #eee;
    }

    .container {
      max-width: 600px;
      margin: 100px auto;
      padding: 20px;
      text-align: center;
    }

    button {
      padding: 10px 20px;
      font-size: 1rem;
      cursor: pointer;
      border: none;
      border-radius: 4px;
      background-color: #007acc;
      color: white;
      transition: background-color 0.3s ease;
    }

    button:hover {
      background-color: #005fa3;
    }

    /* Accessibility: respect user's reduced motion preference */
    @media (prefers-reduced-motion: reduce) {
      * {
        transition: none !important;
```

```
      }
    }
  </style>
</head>
<body>
  <div class="container">
    <h1>Welcome to the Theme Switcher!</h1>
    <p>This example toggles between light and dark themes smoothly.</p>
    <button id="themeToggle">Toggle Theme</button>
  </div>

  <script>
    const toggleButton = document.getElementById('themeToggle');
    const body = document.body;

    // Load saved user preference
    const savedTheme = localStorage.getItem('theme');
    if (savedTheme) {
      body.classList.toggle('dark-theme', savedTheme === 'dark');
    }

    toggleButton.addEventListener('click', () => {
      const isDark = body.classList.toggle('dark-theme');
      localStorage.setItem('theme', isDark ? 'dark' : 'light');
    });
  </script>
</body>
</html>
```

8.4.4 How This Works

- When the page loads, the script checks `localStorage` for a saved theme preference and applies it.
- Clicking the **Toggle Theme** button toggles the `dark-theme` class on the `<body>`.
- CSS transitions animate the background and text color changes smoothly.
- The button color also changes on hover for better user interaction.
- The CSS respects users' **reduced motion** preferences by disabling transitions if requested.

8.4.5 Handling Edge Cases and Enhancements

- **Reduced Motion**: The CSS media query `prefers-reduced-motion` disables transitions to respect users who prefer minimal animations.
- **User Preference Persistence**: Storing the preference in `localStorage` preserves the chosen theme across page reloads and sessions.
- **Extensibility**: You can easily add more themes (e.g., high contrast, sepia) by adding

new CSS classes and toggling accordingly.
- **Additional Effects**: Experiment with transition properties like `transform` or `box-shadow` for more dynamic UI polish.

8.4.6 Try It Yourself

- Add a third theme and a new toggle button.
- Animate other page elements (like headers or buttons) when switching themes.
- Integrate with system color scheme preferences using the `prefers-color-scheme` media query for automatic theme selection.

8.4.7 Summary

This example ties together key concepts:

- **CSS transitions** for smooth visual changes.
- **Class toggling** in JavaScript for dynamic style updates.
- **User preference persistence** using `localStorage`.
- **Accessibility considerations** for reduced motion users.

With these skills, you can create polished, user-friendly UI themes that enhance your web apps!

Chapter 9.

DOM Storage and State Persistence

1. Using `localStorage` and `sessionStorage`
2. Storing and Retrieving Complex Data Structures
3. Practical Example: Persistent User Preferences

9 DOM Storage and State Persistence

9.1 Using `localStorage` and `sessionStorage`

Modern web applications often need to **remember user preferences, session data, or state** between page reloads or visits. The **Web Storage API** provides two powerful storage mechanisms for this purpose:

- **`localStorage`**: Stores data persistently with no expiration date — data remains even after closing the browser.
- **`sessionStorage`**: Stores data only for the duration of the page session — data is cleared when the tab or window is closed.

9.1.1 What Are `localStorage` and `sessionStorage`?

Both are part of the **Web Storage API** and provide a simple key-value storage interface accessible via JavaScript in the browser.

Feature	`localStorage`	`sessionStorage`
Scope	Across browser sessions	Current tab or window only
Persistence	Permanent until cleared by code/user	Cleared when tab/window closes
Storage limit	Around 5–10 MB per origin	Similar to `localStorage`
Accessible from	Same origin (protocol + domain + port)	Same origin and same tab/window

9.1.2 Basic Operations

The API revolves around **string-based key-value pairs** stored in either `localStorage` or `sessionStorage`.

9.1.3 Storing Data

```
localStorage.setItem('username', 'Alice');
sessionStorage.setItem('sessionID', 'abc123');
```

9.1.4 Retrieving Data

```
const user = localStorage.getItem('username'); // "Alice"
const session = sessionStorage.getItem('sessionID'); // "abc123"
```

9.1.5 Removing Data

```
localStorage.removeItem('username');
sessionStorage.removeItem('sessionID');
```

9.1.6 Clearing All Stored Data

```
localStorage.clear();
sessionStorage.clear();
```

9.1.7 Typical Use Cases

- `localStorage`:
  - Saving user preferences (e.g., theme, language).
  - Remembering authentication tokens (with caution).
  - Caching data to avoid repeated server requests.
- `sessionStorage`:
  - Storing temporary session data like form inputs or multi-step wizard states.
  - Keeping ephemeral state tied to a particular tab or window.

9.1.8 Security Considerations

- Both storage mechanisms store data as **plain text strings** on the client side.
- Avoid storing sensitive data like passwords or personal information.
- Be cautious about **Cross-Site Scripting (XSS)** attacks, which can access Web Storage data if the page is vulnerable.
- Implement proper input sanitization and Content Security Policy (CSP) headers to

mitigate risks.

9.1.9 Runnable Example: Using `localStorage` and `sessionStorage`

HTML:

```
  <h2>Web Storage Example</h2>
  <input type="text" id="inputField" placeholder="Type something" />
  <button id="saveLocal">Save to localStorage</button>
  <button id="saveSession">Save to sessionStorage</button>
  <button id="loadLocal">Load from localStorage</button>
  <button id="loadSession">Load from sessionStorage</button>
  <button id="clearStorage">Clear Both</button>

  <p id="output"></p>
```

Javascript:

```
  <script>
    const input = document.getElementById('inputField');
    const output = document.getElementById('output');

    document.getElementById('saveLocal').addEventListener('click', () => {
      localStorage.setItem('myData', input.value);
      output.textContent = 'Saved to localStorage!';
    });

    document.getElementById('saveSession').addEventListener('click', () => {
      sessionStorage.setItem('myData', input.value);
      output.textContent = 'Saved to sessionStorage!';
    });

    document.getElementById('loadLocal').addEventListener('click', () => {
      const data = localStorage.getItem('myData');
      output.textContent = data ? `Loaded from localStorage: ${data}` : 'No data in localStorage.';
    });

    document.getElementById('loadSession').addEventListener('click', () => {
      const data = sessionStorage.getItem('myData');
      output.textContent = data ? `Loaded from sessionStorage: ${data}` : 'No data in sessionStorage.';
    });

    document.getElementById('clearStorage').addEventListener('click', () => {
      localStorage.clear();
      sessionStorage.clear();
      output.textContent = 'Cleared both storages.';
    });
  </script>
```

```
<!DOCTYPE html>
<html lang="en">
<head>
```

```
  <meta charset="UTF-8" />
  <title>Web Storage Demo</title>
</head>
<body>
  <h2>Web Storage Example</h2>
  <input type="text" id="inputField" placeholder="Type something" />
  <button id="saveLocal">Save to localStorage</button>
  <button id="saveSession">Save to sessionStorage</button>
  <button id="loadLocal">Load from localStorage</button>
  <button id="loadSession">Load from sessionStorage</button>
  <button id="clearStorage">Clear Both</button>

  <p id="output"></p>

  <script>
    const input = document.getElementById('inputField');
    const output = document.getElementById('output');

    document.getElementById('saveLocal').addEventListener('click', () => {
      localStorage.setItem('myData', input.value);
      output.textContent = 'Saved to localStorage!';
    });

    document.getElementById('saveSession').addEventListener('click', () => {
      sessionStorage.setItem('myData', input.value);
      output.textContent = 'Saved to sessionStorage!';
    });

    document.getElementById('loadLocal').addEventListener('click', () => {
      const data = localStorage.getItem('myData');
      output.textContent = data ? `Loaded from localStorage: ${data}` : 'No data in localStorage.';
    });

    document.getElementById('loadSession').addEventListener('click', () => {
      const data = sessionStorage.getItem('myData');
      output.textContent = data ? `Loaded from sessionStorage: ${data}` : 'No data in sessionStorage.';
    });

    document.getElementById('clearStorage').addEventListener('click', () => {
      localStorage.clear();
      sessionStorage.clear();
      output.textContent = 'Cleared both storages.';
    });
  </script>
</body>
</html>
```

Try typing text into the input field and saving it to either storage. Then reload or open a new tab and load from storage to see how persistence differs.

9.1.10 Summary

- **`localStorage`** and **`sessionStorage`** provide simple, fast key-value storage in the browser.
- Use `localStorage` for data that should persist beyond sessions; use `sessionStorage` for temporary session data.
- Both store strings, so serialization (e.g., JSON) is needed for complex data (covered in next section).
- Always consider security best practices and avoid sensitive data storage.

This foundational knowledge of the Web Storage API will help you build richer, stateful web applications that remember user context smoothly and efficiently.

9.2 Storing and Retrieving Complex Data Structures

The Web Storage API (`localStorage` and `sessionStorage`) **only supports storing strings** as values. This means that if you need to store complex data types like **objects, arrays, or nested data**, you need to **serialize** them into strings before storage, and **deserialize** them back into usable JavaScript structures when reading.

9.2.1 Serializing and Deserializing with JSON

The most common way to convert objects and arrays to strings is using:

- `JSON.stringify()` — Converts a JavaScript object or array into a JSON string.
- `JSON.parse()` — Parses a JSON string back into a JavaScript object or array.

9.2.2 Example: Storing and Retrieving an Object

```
const userPreferences = {
  theme: "dark",
  fontSize: 16,
  showNotifications: true,
  favorites: ["news", "sports", "music"]
};

// Serialize and store in localStorage
localStorage.setItem('prefs', JSON.stringify(userPreferences));

// Later, retrieve and deserialize
const storedPrefs = localStorage.getItem('prefs');
```

```
if (storedPrefs) {
  const prefs = JSON.parse(storedPrefs);
  console.log(prefs.theme); // "dark"
  console.log(prefs.favorites[1]); // "sports"
}
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>LocalStorage JSON Example</title>
  <style>
    body {
      font-family: Arial, sans-serif;
      padding: 20px;
    }
    pre {
      background-color: #f4f4f4;
      padding: 10px;
      border-radius: 4px;
    }
  </style>
</head>
<body>

  <h1>LocalStorage Demo</h1>
  <p>Open your browser's developer console to see the stored preferences.</p>
  <pre id="output"></pre>

  <script>
    const userPreferences = {
      theme: "dark",
      fontSize: 16,
      showNotifications: true,
      favorites: ["news", "sports", "music"]
    };

    // Serialize and store in localStorage
    localStorage.setItem('prefs', JSON.stringify(userPreferences));

    // Retrieve and deserialize
    const storedPrefs = localStorage.getItem('prefs');
    const output = document.getElementById('output');

    if (storedPrefs) {
      const prefs = JSON.parse(storedPrefs);
      console.log("Theme:", prefs.theme); // "dark"
      console.log("Second favorite:", prefs.favorites[1]); // "sports"
      output.textContent = `Theme: ${prefs.theme}\nFont Size: ${prefs.fontSize}\nShow Notifications: ${
    }
  </script>

</body>
</html>
```

9.2.3 Pitfalls to Watch Out For

Loss of Methods

- Only data is serialized; **methods (functions) attached to objects are not stored**.
- When you deserialize, you get plain objects without behavior.

Example:

```
const obj = {
  name: "Alice",
  greet() { console.log("Hello " + this.name); }
};

localStorage.setItem('obj', JSON.stringify(obj));

const data = JSON.parse(localStorage.getItem('obj'));
console.log(typeof data.greet); // undefined - method lost
```

Solution: If methods are important, consider reattaching them after parsing or use classes with custom serialization logic.

9.2.4 Non-Serializable Data

- Some data types cannot be serialized, including:
  - Functions
  - Symbols
  - DOM elements
  - `undefined` values (they are omitted)
  - Circular references (cause errors in `JSON.stringify`)

9.2.5 Managing Stateful Data Like User Preferences

Using JSON, you can store complex state, such as a user's UI settings or session data, persistently:

```
function saveUserSettings(settings) {
  localStorage.setItem('userSettings', JSON.stringify(settings));
}

function loadUserSettings() {
  const settings = localStorage.getItem('userSettings');
  return settings ? JSON.parse(settings) : null;
}
```

```
// Example usage
saveUserSettings({ language: 'en', notifications: false, layout: 'grid' });

const settings = loadUserSettings();
if (settings) {
  console.log(`Language: ${settings.language}`); // Output: en
}
```

9.2.6 Handling Versioning and Data Migration

Over time, the structure of your stored data might change (e.g., adding new fields). To handle this gracefully:

- **Store a version number** alongside your data.

```
const appState = {
  version: 2,
  preferences: { theme: "light" },
  // other properties
};

localStorage.setItem('appState', JSON.stringify(appState));
```

- **Check the version on load** and migrate data accordingly.

```
function loadAppState() {
  const raw = localStorage.getItem('appState');
  if (!raw) return null;

  const state = JSON.parse(raw);

  switch (state.version) {
    case 1:
      // Migration logic from version 1 to 2
      state.preferences = { theme: "dark" }; // default new field
      state.version = 2;
      // Save migrated state
      localStorage.setItem('appState', JSON.stringify(state));
      break;
    case 2:
      // Current version, no migration needed
      break;
    default:
      console.warn("Unknown app state version");
  }

  return state;
}
```

9.2.7 Summary

- Use `JSON.stringify()` and `JSON.parse()` to store and retrieve complex data structures in Web Storage.
- Be aware that methods and non-serializable data will be lost.
- Manage data structure changes over time by including versioning and migration strategies.
- This approach lets you safely persist user preferences, app state, and other structured data.

Mastering serialization techniques is essential for building **robust, stateful web applications** that store and restore rich data using the browser's storage APIs.

9.3 Practical Example: Persistent User Preferences

One of the most common uses of `localStorage` is saving user preferences — such as theme choice or font size — so that these settings **persist across page reloads** or browser sessions. In this example, we'll build a simple preferences panel that:

- Lets the user choose between **light** and **dark** themes.
- Allows the user to adjust the **font size**.
- Saves these preferences to `localStorage`.
- Restores preferences when the page loads.
- Provides a way to reset preferences to defaults.

9.3.1 Step 1: HTML Setup

Here's a minimal HTML structure for the preferences UI:

```
<div id="preferences">
  <label>
    Theme:
    <select id="theme-select">
      <option value="light">Light</option>
      <option value="dark">Dark</option>
    </select>
  </label>
  <label>
    Font Size:
    <input type="number" id="font-size-input" min="12" max="36" step="1" value="16" />
  </label>
  <button id="reset-btn">Reset to Default</button>
</div>

<div id="content">
```

```
  <p>This is some sample text to see your preferences in action.</p>
</div>
```

9.3.2 Step 2: JavaScript for Saving and Restoring Preferences

```
// Default preferences
const DEFAULT_PREFS = {
  theme: 'light',
  fontSize: 16,
};

// Load preferences from localStorage or use defaults
function loadPreferences() {
  const stored = localStorage.getItem('userPrefs');
  if (stored) {
    try {
      return JSON.parse(stored);
    } catch {
      console.warn('Failed to parse preferences, using defaults.');
    }
  }
  return { ...DEFAULT_PREFS };
}

// Save preferences to localStorage
function savePreferences(prefs) {
  localStorage.setItem('userPrefs', JSON.stringify(prefs));
}

// Apply preferences to the page UI
function applyPreferences(prefs) {
  // Apply theme class to <body>
  document.body.className = prefs.theme;

  // Apply font size style to content container
  const content = document.getElementById('content');
  content.style.fontSize = prefs.fontSize + 'px';

  // Update UI controls
  document.getElementById('theme-select').value = prefs.theme;
  document.getElementById('font-size-input').value = prefs.fontSize;
}

// Initialize the app
function initPreferences() {
  let prefs = loadPreferences();
  applyPreferences(prefs);

  // Event listeners to update preferences live
  document.getElementById('theme-select').addEventListener('change', e => {
    prefs.theme = e.target.value;
    applyPreferences(prefs);
```

```
    savePreferences(prefs);
  });

  document.getElementById('font-size-input').addEventListener('input', e => {
    const size = parseInt(e.target.value, 10);
    if (!isNaN(size) && size >= 12 && size <= 36) {
      prefs.fontSize = size;
      applyPreferences(prefs);
      savePreferences(prefs);
    }
  });

  // Reset preferences button
  document.getElementById('reset-btn').addEventListener('click', () => {
    prefs = { ...DEFAULT_PREFS };
    applyPreferences(prefs);
    savePreferences(prefs);
  });
}

// Run on page load
window.addEventListener('DOMContentLoaded', initPreferences);
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1" />
  <title>User Preferences Demo</title>
  <style>
    body {
      font-family: Arial, sans-serif;
      margin: 20px;
      transition: background-color 0.4s ease, color 0.4s ease;
    }

    /* Light and Dark Themes */
    body.light {
      background-color: #ffffff;
      color: #222222;
    }

    body.dark {
      background-color: #1e1e1e;
      color: #f0f0f0;
    }

    #preferences {
      margin-bottom: 20px;
      padding: 15px;
      border: 1px solid #ccc;
      border-radius: 6px;
      max-width: 400px;
    }

    #preferences label {
      display: block;
```

```
      margin-bottom: 10px;
    }

    #content {
      padding: 10px;
      border: 1px solid #ddd;
      border-radius: 6px;
      background: #f9f9f9;
    }

    body.dark #content {
      background: #2a2a2a;
      border-color: #444;
    }

    input, select, button {
      padding: 5px 10px;
      font-size: 14px;
      margin-top: 5px;
    }

    button {
      margin-top: 10px;
    }
  </style>
</head>
<body>

  <div id="preferences">
    <label>
      Theme:
      <select id="theme-select">
        <option value="light">Light</option>
        <option value="dark">Dark</option>
      </select>
    </label>
    <label>
      Font Size:
      <input type="number" id="font-size-input" min="12" max="36" step="1" value="16" />
    </label>
    <button id="reset-btn">Reset to Default</button>
  </div>

  <div id="content">
    <p>This is some sample text to see your preferences in action.</p>
  </div>

  <script>
    // Default preferences
    const DEFAULT_PREFS = {
      theme: 'light',
      fontSize: 16,
    };

    // Load preferences from localStorage or use defaults
    function loadPreferences() {
      const stored = localStorage.getItem('userPrefs');
      if (stored) {
```

```
        try {
          return JSON.parse(stored);
        } catch {
          console.warn('Failed to parse preferences, using defaults.');
        }
      }
      return { ...DEFAULT_PREFS };
    }

    // Save preferences to localStorage
    function savePreferences(prefs) {
      localStorage.setItem('userPrefs', JSON.stringify(prefs));
    }

    // Apply preferences to the page UI
    function applyPreferences(prefs) {
      document.body.className = prefs.theme;
      const content = document.getElementById('content');
      content.style.fontSize = prefs.fontSize + 'px';
      document.getElementById('theme-select').value = prefs.theme;
      document.getElementById('font-size-input').value = prefs.fontSize;
    }

    // Initialize the app
    function initPreferences() {
      let prefs = loadPreferences();
      applyPreferences(prefs);

      // Update on theme change
      document.getElementById('theme-select').addEventListener('change', e => {
        prefs.theme = e.target.value;
        applyPreferences(prefs);
        savePreferences(prefs);
      });

      // Update on font size input
      document.getElementById('font-size-input').addEventListener('input', e => {
        const size = parseInt(e.target.value, 10);
        if (!isNaN(size) && size >= 12 && size <= 36) {
          prefs.fontSize = size;
          applyPreferences(prefs);
          savePreferences(prefs);
        }
      });

      // Reset to default preferences
      document.getElementById('reset-btn').addEventListener('click', () => {
        prefs = { ...DEFAULT_PREFS };
        applyPreferences(prefs);
        savePreferences(prefs);
      });
    }

    window.addEventListener('DOMContentLoaded', initPreferences);
  </script>

</body>
</html>
```

9.3.3 How It Works

- On page load, preferences are loaded from `localStorage` or defaulted.
- The page's appearance updates immediately to reflect stored settings.
- When the user changes the theme or font size, event listeners update preferences both **in the UI and in storage**.
- The reset button clears preferences back to defaults.
- All updates happen **live without page reloads**, providing a smooth user experience.

9.3.4 Extending This Example

Here are some ideas to enhance this example:

- **Add more preference categories:** e.g., language selection, layout options, or accessibility settings.
- **Sync preferences across tabs:** using the `storage` event to detect changes in other windows and update UI accordingly.
- **Support `sessionStorage`** for preferences that only persist during a single browser session.
- **Add accessibility enhancements:** such as ARIA attributes or keyboard navigation support.
- **Persist preferences on the server** for logged-in users, syncing between devices.

9.3.5 Summary

This example demonstrates how to combine DOM event handling with `localStorage` operations to create a **persistent, interactive user preferences panel**. This pattern is a foundation for many real-world web applications where user customization and state persistence enhance the overall user experience.

Chapter 10.

Manipulating Document Fragments and Templates

1. What Are Document Fragments and Why Use Them?
2. Using `<template>` Elements for Cloning Content
3. Practical Example: Efficient List Rendering with Templates

10 Manipulating Document Fragments and Templates

10.1 What Are Document Fragments and Why Use Them?

When working with the DOM, inserting or modifying elements directly can cause the browser to perform costly **reflows** and **repaints**, especially when many nodes are involved. To optimize this, JavaScript provides **Document Fragments** — lightweight, in-memory containers that let you batch DOM updates efficiently before inserting them into the live document.

What Is a Document Fragment?

A **Document Fragment** is a special type of DOM node that acts as a **temporary container** for other DOM nodes. It behaves like a mini DOM subtree but is **not part of the main document tree**. Because it's disconnected, changes to it do **not trigger rendering** or layout recalculations by the browser.

Think of a Document Fragment like a **draft workspace** where you can assemble multiple DOM elements without affecting the visible page. Once ready, you can insert the entire fragment into the document **all at once**, causing only a single reflow and repaint.

Why Use Document Fragments?

1. **Performance Improvement:** Adding multiple elements individually causes the browser to update the layout repeatedly. Using a fragment defers this until all elements are ready to be inserted in one go, reducing overhead.
2. **Cleaner Code:** Fragments let you build complex element groups before adding them to the DOM, making your code easier to organize and maintain.
3. **Avoid Intermediate States:** Since fragments are detached, users won't see partially constructed UI elements during script execution.

Analogy: Building a Puzzle

Imagine building a puzzle piece by piece on the table (the live DOM). Every time you place a piece, you slightly disturb the whole picture (the page layout). Instead, imagine assembling the pieces first on a tray (the Document Fragment). When the puzzle is complete, you place the tray on the table all at once — the picture appears instantly and cleanly with only one disturbance.

Basic Example: Using a Document Fragment

```
// Create a new Document Fragment
const fragment = document.createDocumentFragment();

// Create and append elements to the fragment
for (let i = 1; i <= 5; i++) {
  const li = document.createElement('li');
```

```
  li.textContent = `Item ${i}`;
  fragment.appendChild(li);
}

// Append the entire fragment to an existing <ul> in the DOM
const list = document.querySelector('ul');
list.appendChild(fragment);
```

In this example:

- Five `<li>` items are created and added to the fragment.
- The fragment is appended once to the `<ul>`.
- The browser performs a **single reflow/repaint**, making it more efficient than appending each `<li>` individually.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Document Fragment Example</title>
  <style>
    body {
      font-family: Arial, sans-serif;
      padding: 20px;
    }
    ul {
      padding-left: 20px;
    }
    li {
      margin-bottom: 5px;
    }
  </style>
</head>
<body>

  <h2>List of Items</h2>
  <ul id="itemList"></ul>

  <script>
    // Create a new Document Fragment
    const fragment = document.createDocumentFragment();

    // Create and append elements to the fragment
    for (let i = 1; i <= 5; i++) {
      const li = document.createElement('li');
      li.textContent = `Item ${i}`;
      fragment.appendChild(li);
    }

    // Append the entire fragment to the existing <ul>
    const list = document.getElementById('itemList');
    list.appendChild(fragment);
  </script>

</body>
</html>
```

Summary

- **Document Fragments** are invisible, lightweight containers used to hold DOM nodes temporarily.
- They help improve performance by batching DOM insertions.
- They prevent intermediate rendering, ensuring cleaner updates.
- Use them whenever adding multiple nodes to the DOM to optimize rendering and provide smoother user experiences.

In the next section, we'll explore how `<template>` elements complement Document Fragments by providing reusable HTML snippets for cloning content efficiently.

10.2 Using `<template>` Elements for Cloning Content

When building dynamic web applications, you often need to create multiple copies of the same HTML structure—like list items, cards, or form groups. Writing this HTML repeatedly or constructing it entirely with JavaScript can be cumbersome and error-prone. The **`<template>` element** solves this problem by allowing you to define reusable chunks of HTML that are **inert and not rendered** until you decide to use them.

What Is the `template` Element?

The `<template>` tag is a special HTML element designed to hold **markup fragments** that are:

- **Not displayed** on the page when the HTML loads.
- **Not part of the active DOM** until you clone and insert their content.
- Used as **blueprints** for creating DOM nodes dynamically.

This means you can keep your HTML markup clean and separate from JavaScript logic, and instantiate copies of the template as needed.

How Does a Template Work?

Inside a `<template>`, the HTML markup exists but is inactive. When you want to use it, you:

1. Select the `<template>` element from the DOM.
2. Access its `.content` property, which is a Document Fragment containing the template's nodes.
3. Clone this content with `cloneNode(true)` to create a fresh, live copy.
4. Insert the cloned nodes into the visible DOM.

Since the template itself never renders, it acts like a **hidden mold** for repeated content.

Practical Use Cases for `template`

- Rendering lists or tables with repeated rows or items.

- Creating cards, modals, or UI components dynamically.
- Injecting forms or complex elements based on user interaction.

Basic Example: Rendering List Items with a Template

```
<!-- Define the template in HTML -->
<template id="item-template">
  <li class="item">
    <span class="item-name"></span>
    <button class="remove-btn">Remove</button>
  </li>
</template>

<ul id="item-list"></ul>

<script>
  // Sample data to render
  const items = ['Apple', 'Banana', 'Cherry'];

  // Select the template and list container
  const template = document.getElementById('item-template');
  const list = document.getElementById('item-list');

  items.forEach(name => {
    // Clone the template content deeply
    const clone = template.content.cloneNode(true);

    // Customize the cloned content
    clone.querySelector('.item-name').textContent = name;

    // Append the clone to the list
    list.appendChild(clone);
  });
</script>
```

In this example:

- The `<template>` defines the structure of each list item.
- The JavaScript clones the template content for each data item.
- The cloned nodes are updated with the item name and inserted into the DOM.
- This keeps the markup separate and the code clean and efficient.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Template Example</title>
  <style>
    body {
      font-family: Arial, sans-serif;
      padding: 20px;
    }

    ul {
      padding-left: 20px;
```

```
    }

    .item {
      display: flex;
      align-items: center;
      margin-bottom: 8px;
    }

    .item-name {
      flex-grow: 1;
    }

    .remove-btn {
      margin-left: 10px;
      padding: 4px 8px;
      cursor: pointer;
    }
  </style>
</head>
<body>

  <h2>Fruit List</h2>

  <!-- Template Definition -->
  <template id="item-template">
    <li class="item">
      <span class="item-name"></span>
      <button class="remove-btn">Remove</button>
    </li>
  </template>

  <!-- List Container -->
  <ul id="item-list"></ul>

  <script>
    // Sample data
    const items = ['Apple', 'Banana', 'Cherry'];

    const template = document.getElementById('item-template');
    const list = document.getElementById('item-list');

    items.forEach(name => {
      const clone = template.content.cloneNode(true);
      clone.querySelector('.item-name').textContent = name;

      // Optional: add functionality to remove item
      clone.querySelector('.remove-btn').addEventListener('click', (e) => {
        e.target.closest('li').remove();
      });

      list.appendChild(clone);
    });
  </script>

</body>
</html>
```

Key Points About Templates

- Templates **do not render their content** until cloned and inserted.
- `.content` is a Document Fragment, allowing efficient manipulation.
- `cloneNode(true)` makes a deep clone, copying all child nodes.
- Templates help maintain separation of concerns by isolating markup.

Summary

Using `<template>` elements allows you to define reusable HTML fragments that remain dormant until needed. They enable you to efficiently render repeated UI elements, keep your HTML organized, and improve performance by leveraging Document Fragments under the hood.

Next, we will explore a practical example combining Document Fragments and templates for efficient list rendering.

10.3 Practical Example: Efficient List Rendering with Templates

In this section, we'll combine the power of **`<template>` elements** and **Document Fragments** to efficiently render a dynamic list from an array of data. Using these together helps minimize costly DOM updates, leading to better performance—especially when rendering many items.

The Scenario

Suppose we want to display a list of tasks dynamically. We'll:

- Use a `<template>` to define the HTML structure for each task.
- Clone and customize the template for each task.
- Use a Document Fragment to batch all new nodes before inserting them into the DOM at once.

Complete Example Code

HTML:

```
<h2>Task List</h2>
<ul id="task-list"></ul>

<!-- Template defining structure for each task -->
<template id="task-template">
  <li class="task-item">
    <span class="task-name"></span>
    <button class="remove-btn">Remove</button>
  </li>
</template>
```

Javascript:

```
    // Sample array of tasks
    const tasks = [
      { id: 1, name: 'Buy groceries' },
      { id: 2, name: 'Clean the house' },
      { id: 3, name: 'Finish project' }
    ];

    // Select the template and the list container
    const template = document.getElementById('task-template');
    const taskList = document.getElementById('task-list');

    // Create a document fragment to batch DOM insertion
    const fragment = document.createDocumentFragment();

    tasks.forEach(task => {
      // Clone the template content
      const clone = template.content.cloneNode(true);
      const taskItem = clone.querySelector('.task-item'); // This is the <li>

      // Set the task name
      taskItem.querySelector('.task-name').textContent = task.name;

      // Attach event listener
      taskItem.querySelector('.remove-btn').addEventListener('click', () => {
        taskItem.remove(); // Remove this specific task item
      });

      // Append the taskItem to the fragment
      fragment.appendChild(taskItem);
    });

    // Insert all items into the DOM
    taskList.appendChild(fragment);
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Efficient List Rendering</title>
  <style>
    .task-item {
      margin: 5px 0;
      padding: 8px;
      background: #f0f0f0;
      border-radius: 4px;
      display: flex;
      justify-content: space-between;
      align-items: center;
    }
    .remove-btn {
      background: #e74c3c;
      border: none;
      color: white;
      padding: 4px 8px;
      cursor: pointer;
      border-radius: 3px;
    }
```

```
  </style>
</head>
<body>
  <h2>Task List</h2>
  <ul id="task-list"></ul>

  <!-- Template defining structure for each task -->
  <template id="task-template">
    <li class="task-item">
      <span class="task-name"></span>
      <button class="remove-btn">Remove</button>
    </li>
  </template>

  <script>
    // Sample array of tasks
    const tasks = [
      { id: 1, name: 'Buy groceries' },
      { id: 2, name: 'Clean the house' },
      { id: 3, name: 'Finish project' }
    ];

    // Select the template and the list container
    const template = document.getElementById('task-template');
    const taskList = document.getElementById('task-list');

    // Create a document fragment to batch DOM insertion
    const fragment = document.createDocumentFragment();

    tasks.forEach(task => {
      // Clone the template content
      const clone = template.content.cloneNode(true);
      const taskItem = clone.querySelector('.task-item'); // This is the <li>

      // Set the task name
      taskItem.querySelector('.task-name').textContent = task.name;

      // Attach event listener
      taskItem.querySelector('.remove-btn').addEventListener('click', () => {
        taskItem.remove(); // Remove this specific task item
      });

      // Append the taskItem to the fragment
      fragment.appendChild(taskItem);
    });

    // Insert all items into the DOM
    taskList.appendChild(fragment);
  </script>

</body>
</html>
```

Step-by-Step Explanation

1. **Template Definition**: The `<template>` contains the HTML for each task list item—a `<li>` with a task name span and a remove button.

2. **Selecting Elements**: We grab references to the template and the list container (`<ul>`).
3. **Creating a Document Fragment**: The fragment acts as a temporary container for all cloned and customized task nodes. This prevents multiple reflows caused by adding each element directly to the DOM.
4. **Cloning and Binding Data**: For each task object, we clone the template content deeply (`cloneNode(true)`), then update the `.task-name` span's text to the task's name.
5. **Adding Event Listeners**: We add a click listener to the "Remove" button inside each clone. When clicked, it removes the corresponding task item from the DOM.
6. **Batch Append**: After cloning and customizing all tasks, we append the entire fragment to the list container in one operation. This is much more efficient than inserting each item individually.

Encouragement for Experimentation

- **Edit Functionality**: Add an edit button that allows modifying task names inline.
- **Dynamic Data**: Update the `tasks` array dynamically and re-render the list efficiently.
- **Persistence**: Save tasks to `localStorage` so the list persists between page reloads.
- **Styling**: Add CSS classes dynamically using `classList` to highlight completed tasks.
- **Animations**: Use CSS transitions to smoothly fade out removed items.

Why This Approach?

- **Performance**: Minimizes browser reflows and repaints by inserting multiple elements at once.
- **Maintainability**: Keeps HTML markup declarative and separate from JS logic.
- **Scalability**: Easily handles large datasets without significant performance hits.
- **User Experience**: Enables smooth, dynamic UI updates.

This example demonstrates how combining templates with document fragments is a best practice for rendering dynamic lists efficiently, keeping your code clean and your app performant.

Chapter 11.

Working with Shadow DOM

1. Understanding Shadow DOM and Web Components
2. Creating and Attaching Shadow Roots
3. Styling and Encapsulation in Shadow DOM
4. Practical Example: Building a Custom Web Component

11 Working with Shadow DOM

11.1 Understanding Shadow DOM and Web Components

Modern web development emphasizes building reusable, modular components that behave consistently across projects. However, traditional DOM and CSS can lead to challenges such as style conflicts and accidental interference between components. This is where **Shadow DOM** and **Web Components** come into play.

What is Shadow DOM?

The **Shadow DOM** is a browser technology that allows developers to create **encapsulated DOM trees** inside regular DOM elements. Think of it as a "mini-DOM" hidden inside a component, completely isolated from the main document's DOM.

- **Encapsulation:** The Shadow DOM provides a boundary between the component's internals (markup, styles, scripts) and the rest of the page. Styles and scripts inside the shadow root do not leak out, and likewise, the outside page cannot accidentally affect the internals.
- **Scoped Styles:** CSS defined inside a shadow root applies only to that shadow DOM subtree. This solves common problems like CSS class name clashes and unexpected inheritance.
- **Improved Reusability:** Components built with Shadow DOM behave consistently regardless of the surrounding page's styles or scripts.

What Are Web Components?

Web Components are a set of standards designed to help developers create custom, reusable HTML elements with encapsulated functionality and style. They combine three key technologies:

1. **Custom Elements** – Define new HTML tags (e.g., `<my-button>`).
2. **Shadow DOM** – Encapsulate the internal DOM and styles.
3. **HTML Templates** – Define reusable markup structures.

Together, these allow you to build fully self-contained widgets that behave like native HTML elements.

Benefits of Using Shadow DOM and Web Components

Benefit	Description
Style Encapsulation	Prevent CSS from leaking in or out, avoiding conflicts
Scoped DOM	Internal DOM structure hidden and inaccessible from outside
Modularity	Components can be developed, tested, and maintained independently
Reusability	Use components across projects without fear of interference

Benefit	Description
Improved Maintainability	Easier to reason about component behavior in isolation

Visualizing Shadow DOM

Imagine a web page as a big tree. Each custom element with a shadow root has its **own little tree inside it**, disconnected from the main tree:

```
Main DOM Tree
+-- <body>
     +-- <my-widget>   <-- Custom element with Shadow DOM
           +-- Shadow Root (isolated DOM subtree)
               +-- <div> ... </div>
               +-- <style> ... </style>
     +-- <p>Some text</p>
```

In this example, the `<my-widget>`'s shadow root contains markup and styles invisible to the rest of the page's DOM tree.

Analogy: Shadow DOM as a "Black Box"

Think of a Shadow DOM like a black box or a component's "private room":

- Inside the box, the component can have anything it wants—furniture (DOM elements), decoration (CSS), behavior (JS).
- Outside the box, the rest of the page cannot see or affect what's inside.
- Likewise, what happens inside the box doesn't spill out to the rest of the page.

11.1.1 Summary

Shadow DOM is a powerful feature that enables real **encapsulation** of DOM and CSS. By isolating a component's internals from the global page, it avoids conflicts and improves reusability. When combined with custom elements and templates, Shadow DOM is the cornerstone of the Web Components standard, helping developers build robust, modular web applications.

In the next sections, we'll dive into how to create and attach shadow roots, style components, and build practical, reusable web components step by step.

11.2 Creating and Attaching Shadow Roots

Once you understand what the Shadow DOM is and why it's useful, the next step is **creating and attaching a shadow root** to an element. This section will guide you through how to do this using JavaScript and explain key concepts like shadow root modes.

Creating a Shadow Root with `attachShadow()`

To create a shadow root, you call the `attachShadow()` method on a DOM element. This method creates a new shadow root (the root of the shadow DOM subtree) and attaches it to the element.

Syntax:

```
const shadowRoot = element.attachShadow({ mode: 'open' });
```

- `element` is any DOM element (often a custom element).
- The `mode` option specifies how the shadow root is exposed to JavaScript outside the component (more on this below).
- The method returns the shadow root object, where you can add child nodes and styles.

Shadow Root Modes: `open` vs `closed`

The `mode` option determines the accessibility of the shadow root from outside the component:

Mode	Description	Access Example
`open`	The shadow root is accessible via the element's `shadowRoot` property	`element.shadowRoot` returns the shadow root
`closed`	The shadow root is **not accessible** via `shadowRoot`; it is hidden from external scripts	`element.shadowRoot` returns `null`

Implications:

- **Open Shadow Roots** allow external scripts and developers to inspect or manipulate the shadow DOM if needed.
- **Closed Shadow Roots** enforce stronger encapsulation, hiding the shadow DOM subtree completely from outside access.

  Most developers use `open` mode during development for debugging convenience, but `closed` can be chosen for stricter encapsulation.

Adding Content Inside the Shadow Root

Once you have the shadow root, you can populate it just like a normal DOM element. You can:

- Create elements programmatically and append them.
- Insert HTML content via `innerHTML`.

- Attach styles that will only apply inside the shadow DOM.

Example:

```
<div id="host"></div>

<script>
  const host = document.getElementById('host');

  // Attach an open shadow root to the host element
  const shadowRoot = host.attachShadow({ mode: 'open' });

  // Add some HTML content
  shadowRoot.innerHTML = `
    <style>
      p {
        color: teal;
        font-weight: bold;
      }
    </style>
    <p>This paragraph is inside the shadow DOM.</p>
  `;
</script>
```

In this example:

- The `<div id="host">` acts as the **shadow host**.
- We create a shadow root on it.
- We add styles and markup that are scoped **only** inside the shadow DOM.
- The styles do **not** affect elements outside the shadow root.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Shadow DOM Example</title>
  <style>
    body {
      font-family: Arial, sans-serif;
      padding: 20px;
    }
    #host {
      border: 1px solid #ccc;
      padding: 10px;
      margin-top: 20px;
    }
  </style>
</head>
<body>

  <h2>Shadow DOM Demo</h2>
  <p>This paragraph is in the light DOM.</p>

  <div id="host"></div>

  <script>
```

```
    const host = document.getElementById('host');

    // Attach an open shadow root to the host element
    const shadowRoot = host.attachShadow({ mode: 'open' });

    // Add some HTML content to the shadow root
    shadowRoot.innerHTML = `
      <style>
        p {
          color: teal;
          font-weight: bold;
          margin: 0;
        }
      </style>
      <p>This paragraph is inside the shadow DOM.</p>
    `;
  </script>

</body>
</html>
```

Relationship Between Shadow DOM and Light DOM

- The **light DOM** refers to the regular DOM tree you see on the page.
- The **shadow DOM** is an isolated subtree attached to a shadow host element.

Key points:

- The shadow DOM is **hidden** from the light DOM in terms of CSS and JavaScript accessibility (depending on mode).
- From the browser's rendering perspective, the shadow DOM **replaces** the host's original children visually.
- Scripts can interact with the shadow DOM by accessing the shadow root (if `open`) or through the custom element's API.
- Events triggered inside shadow DOM nodes will **bubble through** to the light DOM unless stopped, allowing interaction between the two DOMs.

11.2.1 Summary

Creating and attaching a shadow root is straightforward with `attachShadow()`. Choosing the right mode (`open` or `closed`) balances debugging access and encapsulation needs. Once attached, the shadow root acts like a mini-DOM where you can add isolated markup and styles, keeping your component's internals safe from outside interference.

11.3 Styling and Encapsulation in Shadow DOM

One of the key benefits of the Shadow DOM is **style encapsulation** — the ability to keep a component's CSS isolated from the rest of the page, preventing style conflicts and unintended side effects. This section explains how styles behave inside the shadow DOM, how to customize styles from outside, and best practices for working with Shadow DOM styling.

Scoped CSS: Styles Are Isolated Inside Shadow DOM

When you add CSS inside a shadow root (via `<style>` tags or linked stylesheets), these styles apply **only to the elements inside that shadow root**. They do **not** leak out to affect elements in the main document or other shadow roots.

Similarly, styles defined in the main document's stylesheet do **not** affect elements inside the shadow DOM. This isolation means:

- You can use generic class names or element selectors without worrying about collisions.
- Your component styles won't accidentally override page-wide styles, and vice versa.

Example:

```
<div id="host"></div>

<script>
  const host = document.getElementById('host');
  const shadowRoot = host.attachShadow({ mode: 'open' });

  shadowRoot.innerHTML = `
    <style>
      p {
        color: blue;
      }
    </style>
    <p>This text is blue and styled inside shadow DOM.</p>
  `;

  // In the main document:
  // p { color: red; } would not affect the paragraph inside the shadow DOM.
</script>
```

Style Inheritance and Limitations

Some styles, such as `color`, `font-family`, and `font-size`, **do inherit** into the shadow DOM from the host element and the outer document. However, many other CSS properties do **not** cross the shadow boundary, ensuring encapsulation.

- If you need consistent typography or colors, you might set CSS variables on the host and use them inside the shadow DOM (see next section).
- Certain global CSS resets or styles will not affect shadow DOM content.

Using CSS Custom Properties (Variables) for Flexible Styling

CSS variables (`--my-var`) **do cross the shadow boundary**, making them a powerful tool to provide customizable styling from outside a shadow DOM.

How it works:

- Define variables on the shadow host or an ancestor in the light DOM.
- Use those variables inside shadow DOM styles.

Example:

```
/* Light DOM / global styles */
#host {
  --primary-color: crimson;
}
```

```
shadowRoot.innerHTML = `
  <style>
    p {
      color: var(--primary-color, black); /* fallback to black */
    }
  </style>
  <p>Styled with CSS variable.</p>
`;
```

Changing `--primary-color` on `#host` dynamically updates the paragraph color inside the shadow DOM.

Exposing Parts of Shadow DOM for Styling: `::part` and `::slotted`

Sometimes you want to allow limited styling of shadow DOM internals from outside. The Shadow DOM spec provides two pseudo-elements for this:

1. **`::part()`** — Allows the shadow DOM to expose named parts that can be styled externally.

Inside the shadow DOM, elements can declare a `part` attribute:

```
<button part="button">Click me</button>
```

Outside, CSS can target that part:

```
#host::part(button) {
  background-color: orange;
}
```

2. **`::slotted()`** — Targets light DOM nodes that are passed into the shadow DOM's `<slot>` elements.

Example:

```
<!-- Light DOM -->
<div id="host">
  <span class="highlighted">Hello</span>
</div>
```

```
// Shadow DOM content
shadowRoot.innerHTML = `
  <style>
    ::slotted(.highlighted) {
      color: green;
      font-weight: bold;
    }
  </style>
  <slot></slot>
`;
```

This styles the `.highlighted` element that is slotted inside the shadow root.

Challenges and Best Practices

- **Balancing encapsulation and flexibility:** Too much encapsulation can make customization hard. Use `::part` and CSS variables wisely to expose customizable hooks.
- **Avoid global styles leaking:** Don't rely on global CSS inside shadow DOM; encapsulation means global styles won't apply.
- **Manage complex styling carefully:** Large, deeply nested shadow DOM trees can be tricky to style; consider maintaining consistent CSS variables.
- **Accessibility considerations:** Styles like focus outlines or animations inside shadow DOM still need proper attention to keep components accessible.

11.3.1 Summary

Styling inside the Shadow DOM provides **powerful encapsulation** that prevents CSS conflicts but also requires thoughtful use of CSS variables, `::part`, and `::slotted` selectors to expose and customize component styles from outside. By leveraging these techniques, you can build robust, reusable components with clean, maintainable styles.

11.4 Practical Example: Building a Custom Web Component

In this section, we'll create a simple custom web component that uses Shadow DOM encapsulation. This example will demonstrate defining a new element, attaching a shadow root, applying scoped styles, and using the component on a webpage while keeping its styles and markup isolated from the rest of the document.

Step 1: Define the Custom Element Class

```
class MyGreeting extends HTMLElement {
  constructor() {
    super();

    // Attach an open shadow root to this element
    this.attachShadow({ mode: 'open' });

    // Add HTML content and styles inside the shadow root
    this.shadowRoot.innerHTML = `
      <style>
        p {
          color: teal;
          font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
          padding: 10px;
          background-color: #f0f8ff;
          border-radius: 5px;
          text-align: center;
        }
      </style>
      <p>Hello, <span id="name">World</span>!</p>
    `;
  }

  // Optional: Observe attribute changes for dynamic updates
  static get observedAttributes() {
    return ['name'];
  }

  attributeChangedCallback(attrName, oldVal, newVal) {
    if (attrName === 'name' && this.shadowRoot) {
      const nameSpan = this.shadowRoot.getElementById('name');
      if (nameSpan) {
        nameSpan.textContent = newVal || 'World';
      }
    }
  }
}

// Register the custom element with the browser
customElements.define('my-greeting', MyGreeting);
```

Step 2: Use the Custom Element in HTML

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Custom Web Component Demo</title>
</head>
<body>

  <!-- Using the custom element -->
  <my-greeting></my-greeting>
```

```
  <my-greeting name="Alice"></my-greeting>
  <my-greeting name="Bob"></my-greeting>

  <script src="my-greeting.js"></script>
</body>
</html>
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Custom Web Component Demo</title>
  <style>
    body {
      font-family: Arial, sans-serif;
      padding: 20px;
    }
  </style>
</head>
<body>

  <h2>Custom Element: &lt;my-greeting&gt;</h2>

  <!-- Using the custom element -->
  <my-greeting></my-greeting>
  <my-greeting name="Alice"></my-greeting>
  <my-greeting name="Bob"></my-greeting>

  <script>
    class MyGreeting extends HTMLElement {
      constructor() {
        super();
        this.attachShadow({ mode: 'open' });
        this.shadowRoot.innerHTML = `
          <style>
            p {
              color: teal;
              font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
              padding: 10px;
              background-color: #f0f8ff;
              border-radius: 5px;
              text-align: center;
              margin: 10px 0;
            }
          </style>
          <p>Hello, <span id="name">World</span>!</p>
        `;
      }

      static get observedAttributes() {
        return ['name'];
      }

      attributeChangedCallback(attrName, oldVal, newVal) {
        if (attrName === 'name' && this.shadowRoot) {
          const nameSpan = this.shadowRoot.getElementById('name');
          if (nameSpan) {
```

```
            nameSpan.textContent = newVal || 'World';
          }
        }
      }
    }

    customElements.define('my-greeting', MyGreeting);
  </script>

</body>
</html>
```

11.4.1 How It Works

- **Custom Element Class**: We create a class `MyGreeting` extending `HTMLElement`. The constructor attaches a shadow root (`open` mode), so JavaScript outside can access it if needed.
- **Shadow DOM Content**: Inside the shadow root, we inject HTML with a styled paragraph. Styles are scoped — they won't leak out or be affected by global styles.
- **Attributes & Reactivity**: The component watches for changes to the `name` attribute. When it changes, the greeting text updates dynamically inside the shadow DOM.
- **Usage in HTML**: The custom element `<my-greeting>` can be used like any standard HTML tag. We pass a `name` attribute to customize the greeting.

11.4.2 Isolation and Encapsulation

- The styles defined inside the shadow root apply only to the content inside this component.
- Global CSS or styles on the main page won't affect this greeting component.
- Similarly, styles inside the component do not affect the rest of the page.

11.4.3 Extending This Example

- Add more attributes for customization (e.g., colors, fonts).
- Dispatch custom events on user interaction inside the component.
- Use slots for more flexible content insertion.
- Add methods or properties to the component class to expose functionality.

Chapter 12.

Animations and Transitions with the DOM

1. Using CSS Transitions via JavaScript
2. Manipulating Keyframe Animations
3. Using the Web Animations API
4. Practical Example: Animated Modal Dialog

12 Animations and Transitions with the DOM

12.1 Using CSS Transitions via JavaScript

CSS transitions allow developers to create smooth, animated changes to CSS properties over time, without relying on complex JavaScript animation logic. They are perfect for enhancing user experience with subtle effects like fading elements in or out, sliding menus, or smoothly changing colors.

In this section, you'll learn how to use JavaScript to control transitions by manipulating styles and classes dynamically.

12.1.1 Understanding CSS Transitions

A **CSS transition** defines how a change in a CSS property should animate over time. For example, instead of a background color changing instantly, it can fade gradually.

The basic transition properties include:

- `transition-property`: the CSS property to animate (e.g., `opacity`, `transform`, `background-color`)
- `transition-duration`: how long the transition lasts (`0.5s`, `200ms`, etc.)
- `transition-timing-function`: the speed curve (`ease`, `linear`, `ease-in`, etc.)
- `transition-delay`: how long to wait before starting the transition

Example CSS:

```
.box {
  background-color: skyblue;
  transition: background-color 0.3s ease;
}
.box.active {
  background-color: steelblue;
}
```

12.1.2 Triggering Transitions with JavaScript

The power of transitions comes from how easily they can be activated by **adding or removing classes** or modifying inline styles.

HTML + CSS + JS Example: Background Color Transition

Style:

```
  <style>
    .box {
      width: 200px;
      height: 100px;
      background-color: skyblue;
      transition: background-color 0.5s ease;
      margin: 20px auto;
      text-align: center;
      line-height: 100px;
      font-weight: bold;
    }
    .box.active {
      background-color: tomato;
    }
  </style>
```

HTML:

```
<div class="box" id="myBox">Hover Me</div>
```

Javascript:

```
    const box = document.getElementById('myBox');

    box.addEventListener('mouseenter', () => {
      box.classList.add('active');
    });

    box.addEventListener('mouseleave', () => {
      box.classList.remove('active');
    });
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <style>
    .box {
      width: 200px;
      height: 100px;
      background-color: skyblue;
      transition: background-color 0.5s ease;
      margin: 20px auto;
      text-align: center;
      line-height: 100px;
      font-weight: bold;
    }
    .box.active {
      background-color: tomato;
    }
  </style>
</head>
<body>

  <div class="box" id="myBox">Hover Me</div>
```

```
  <script>
    const box = document.getElementById('myBox');

    box.addEventListener('mouseenter', () => {
      box.classList.add('active');
    });

    box.addEventListener('mouseleave', () => {
      box.classList.remove('active');
    });
  </script>

</body>
</html>
```

What This Does:

- When the mouse enters the box, it adds the `active` class, triggering the `background-color` transition.
- When the mouse leaves, the class is removed, reverting the style with a smooth transition.

12.1.3 Transitioning Other Properties

You can animate multiple properties at once:

```
transition: opacity 0.3s ease, transform 0.3s ease;
```

Common Examples:

- **Fade In/Out:**

```
.fade {
  opacity: 0;
  transition: opacity 0.4s ease;
}
.fade.visible {
  opacity: 1;
}
```

- **Slide Down:**

```
.slide {
  transform: translateY(-20px);
  opacity: 0;
  transition: transform 0.4s ease, opacity 0.4s ease;
}
.slide.visible {
  transform: translateY(0);
  opacity: 1;
}
```

- **Button Highlight:**

```
.btn {
  background-color: white;
  color: black;
  transition: background-color 0.3s, color 0.3s;
}
.btn:hover {
  background-color: black;
  color: white;
}
```

12.1.4 Best Practices

- **Use classes for control**: Rather than setting styles directly in JavaScript, toggle a class that contains transition rules.
- **Avoid layout thrashing**: Don't change multiple styles back-to-back in a loop; batch DOM reads and writes.
- **Use `will-change` sparingly**: Hinting that an element is going to change (e.g., `will-change: transform`) can improve performance but may increase memory usage.
- **Keep transitions responsive**: Respect user preferences like `prefers-reduced-motion` when designing UI transitions.

12.1.5 Summary

CSS transitions provide an easy way to animate changes in appearance using minimal JavaScript. By toggling classes or modifying styles, you can create polished, responsive UI behaviors like:

- Fading elements in/out
- Sliding panels or menus
- Smooth hover effects

In the next section, we'll explore how to animate using **keyframes**, giving you even more control over complex animations.

12.2 Manipulating Keyframe Animations

CSS keyframe animations are a powerful tool for creating rich, multi-step animations directly in your stylesheets. Unlike transitions, which animate from one state to another, keyframes allow you to define multiple intermediate stages and greater control over the animation sequence.

In this section, you'll learn how to define and trigger keyframe animations using JavaScript, as well as how to control their playback dynamically.

12.2.1 What Are Keyframe Animations?

A keyframe animation uses the `@keyframes` rule to describe how an element should animate over time. You define the animation steps and then apply the animation using CSS properties like `animation-name`, `animation-duration`, `animation-timing-function`, etc.

Example: CSS Keyframe Animation

```
@keyframes bounce {
  0%, 100% {
    transform: translateY(0);
  }
  50% {
    transform: translateY(-20px);
  }
}

.box {
  animation-name: bounce;
  animation-duration: 0.6s;
  animation-timing-function: ease-in-out;
  animation-iteration-count: infinite;
}
```

12.2.2 Controlling Animations with JavaScript

You can dynamically trigger or control CSS animations through JavaScript by:

1. **Adding or removing classes**
2. **Modifying inline animation styles**
3. **Using animation-related properties**

Example: Trigger Animation with a Class

```
<style>
@keyframes shake {
  0% { transform: translateX(0); }
  25% { transform: translateX(-5px); }
  50% { transform: translateX(5px); }
  75% { transform: translateX(-5px); }
  100% { transform: translateX(0); }
}
```

```
.shake {
  animation: shake 0.4s ease;
}
</style>

<button id="btn">Click to Shake</button>

<script>
  const btn = document.getElementById('btn');

  btn.addEventListener('click', () => {
    btn.classList.remove('shake'); // reset if applied
    void btn.offsetWidth;          // force reflow to retrigger animation
    btn.classList.add('shake');
  });
</script>
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Shake Button Example</title>
  <style>
    body {
      font-family: Arial, sans-serif;
      padding: 30px;
    }

    @keyframes shake {
      0% { transform: translateX(0); }
      25% { transform: translateX(-5px); }
      50% { transform: translateX(5px); }
      75% { transform: translateX(-5px); }
      100% { transform: translateX(0); }
    }

    .shake {
      animation: shake 0.4s ease;
    }

    #btn {
      padding: 10px 20px;
      font-size: 16px;
      cursor: pointer;
    }
  </style>
</head>
<body>

  <h2>Click the Button to Shake</h2>
  <button id="btn">Click to Shake</button>

  <script>
    const btn = document.getElementById('btn');

    btn.addEventListener('click', () => {
      btn.classList.remove('shake');  // Reset animation if already applied
```

```
      void btn.offsetWidth;           // Force reflow to allow reapplying animation
      btn.classList.add('shake');
    });
  </script>

</body>
</html>
```

Why use `void btn.offsetWidth`? It forces a DOM reflow so that the same animation can be restarted, even if the class was already present.

12.2.3 Controlling Playback State

You can **pause** or **resume** an animation using the `animation-play-state` property:

```
.box.paused {
  animation-play-state: paused;
}
```

```
box.classList.add('paused'); // pauses the animation
box.classList.remove('paused'); // resumes the animation
```

Alternatively, you can modify the inline style:

```
element.style.animationPlayState = 'paused';
```

12.2.4 Listening to Animation Events

CSS animations fire events during their lifecycle, which you can use to coordinate other logic:

- `animationstart`: fired when animation starts
- `animationend`: fired when animation ends
- `animationiteration`: fired at each iteration

Example:

```
const element = document.querySelector('.box');

element.addEventListener('animationend', () => {
  console.log('Animation complete!');
});
```

12.2.5 Practical Example: Animate a Notification

CSS:

```
<style>
@keyframes slideIn {
  from { opacity: 0; transform: translateY(-20px); }
  to { opacity: 1; transform: translateY(0); }
}

.notification {
  opacity: 0;
  background: #4caf50;
  color: white;
  padding: 10px;
  margin: 20px;
  border-radius: 5px;
}

.notification.show {
  animation: slideIn 0.5s ease forwards;
}
</style>
```

HTML:

```
<div id="message" class="notification">Success! Your action was completed.</div>
<button id="showBtn">Show Notification</button>
```

Javascript:

```
<script>
  const msg = document.getElementById('message');
  const btn = document.getElementById('showBtn');

  btn.addEventListener('click', () => {
    msg.classList.remove('show');
    void msg.offsetWidth; // restart animation
    msg.classList.add('show');
  });
</script>
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Slide-In Notification</title>
  <style>
    body {
      font-family: Arial, sans-serif;
      padding: 30px;
    }

    @keyframes slideIn {
      from { opacity: 0; transform: translateY(-20px); }
```

```
      to { opacity: 1; transform: translateY(0); }
    }

    .notification {
      opacity: 0;
      background: #4caf50;
      color: white;
      padding: 10px 20px;
      margin: 20px 0;
      border-radius: 5px;
      max-width: 300px;
      transition: opacity 0.5s ease;
    }

    .notification.show {
      animation: slideIn 0.5s ease forwards;
    }

    #showBtn {
      padding: 10px 20px;
      font-size: 16px;
      cursor: pointer;
    }
  </style>
</head>
<body>

  <h2>Slide-In Notification Demo</h2>
  <div id="message" class="notification">YES Success! Your action was completed.</div>
  <button id="showBtn">Show Notification</button>

  <script>
    const msg = document.getElementById('message');
    const btn = document.getElementById('showBtn');

    btn.addEventListener('click', () => {
      msg.classList.remove('show');
      void msg.offsetWidth; // force reflow to restart animation
      msg.classList.add('show');
    });
  </script>

</body>
</html>
```

12.2.6 Summary

Keyframe animations allow for expressive, multi-step animations defined entirely in CSS. JavaScript enhances their usefulness by:

- Toggling classes to start animations
- Using `animation-play-state` to pause/resume
- Listening for animation lifecycle events

These techniques offer powerful control over animation flow and timing while keeping code performant and maintainable. Up next, we'll explore the **Web Animations API** for even more advanced, JavaScript-driven animation control.

12.3 Using the Web Animations API

The **Web Animations API (WAAPI)** is a modern, JavaScript-driven way to animate DOM elements with precision and flexibility. Unlike CSS-based animations, the WAAPI gives you full programmatic control over timing, direction, playback, and sequencing — all without needing to modify CSS files or class names.

12.3.1 Why Use the Web Animations API?

The Web Animations API provides several advantages:

- Animations are controlled entirely in JavaScript.
- You can pause, reverse, or cancel animations on the fly.
- It supports runtime-generated keyframes and dynamic properties.
- Fine-grained timing control (delays, easing, iterations, playback rate).

  It's especially useful in interactive or complex UI components where CSS animations are too limited or difficult to coordinate.

12.3.2 The `animate()` Method

The core of the Web Animations API is the `Element.animate()` method, which takes two arguments:

```
element.animate(keyframes, options);
```

- **`keyframes`**: An array of style states.
- **`options`**: An object defining timing, duration, iterations, etc.

Example: Fade In an Element

```
const box = document.querySelector('.box');

box.animate(
  [
    { opacity: 0, transform: 'translateY(-20px)' },
```

```
    { opacity: 1, transform: 'translateY(0)' }
  ],
  {
    duration: 500,
    easing: 'ease-out',
    fill: 'forwards'
  }
);
```

12.3.3 Timing Options Explained

Option	Description
`duration`	Total time (in ms) for one iteration
`easing`	Controls speed curve (e.g., `linear`, `ease-in`)
`fill`	Determines if end styles are retained (`forwards`)
`iterations`	Number of times the animation repeats
`direction`	`normal`, `reverse`, `alternate`, etc.
`delay`	Wait time before animation starts (in ms)

12.3.4 Controlling Animations Programmatically

Calling `animate()` returns an **Animation** object. You can use this to control the animation:

```
const animation = box.animate(
  [{ opacity: 0 }, { opacity: 1 }],
  { duration: 1000 }
);

animation.pause();
animation.play();
animation.reverse();
animation.cancel();
```

You can also **listen to animation events**:

```
animation.onfinish = () => {
  console.log('Animation complete!');
};
```

12.3.5 Chaining Animations

You can sequence animations using promises:

```
const fade = box.animate(
  [{ opacity: 0 }, { opacity: 1 }],
  { duration: 500, fill: 'forwards' }
);

fade.finished.then(() => {
  return box.animate(
    [{ transform: 'scale(1)' }, { transform: 'scale(1.2)' }],
    { duration: 300, direction: 'alternate', iterations: 2 }
  ).finished;
});
```

12.3.6 Comparison: Web Animations API vs CSS Animations

Feature	Web Animations API	CSS Animations
Defined in	JavaScript	CSS
Runtime flexibility	High	Low
Dynamic keyframes	Yes	No
Playback control	Full (pause, reverse, etc)	Limited (via CSS states)
Event hooks	`onfinish`, promises	`animationend` events
Use cases	Dynamic, interactive UIs	Simple transitions

12.3.7 Practical Example: Slide and Fade In

CSS:

```
<style>
.box {
  width: 150px;
  height: 150px;
  background: teal;
  margin: 20px;
}
</style>
```

HTML:

```
<div class="box" id="box"></div>
<button id="animateBtn">Animate</button>
```

Javascript:

```
  const box = document.getElementById('box');
  const button = document.getElementById('animateBtn');

  button.addEventListener('click', () => {
    const animation = box.animate(
      [
        { opacity: 0, transform: 'translateX(-100px)' },
        { opacity: 1, transform: 'translateX(0)' }
      ],
      {
        duration: 600,
        easing: 'ease-out',
        fill: 'forwards'
      }
    );

    animation.onfinish = () => {
      console.log('Slide-in complete.');
    };
  });
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Web Animations API Demo</title>
  <style>
    body {
      font-family: Arial, sans-serif;
      padding: 30px;
    }
    .box {
      width: 150px;
      height: 150px;
      background: teal;
      margin: 20px 0;
      opacity: 1;
      transform: translateX(0);
    }
    #animateBtn {
      padding: 10px 20px;
      font-size: 16px;
      cursor: pointer;
    }
  </style>
</head>
<body>

  <div class="box" id="box"></div>
  <button id="animateBtn">Animate</button>

  <script>
    const box = document.getElementById('box');
    const button = document.getElementById('animateBtn');
```

```
    button.addEventListener('click', () => {
      const animation = box.animate(
        [
          { opacity: 0, transform: 'translateX(-100px)' },
          { opacity: 1, transform: 'translateX(0)' }
        ],
        {
          duration: 600,
          easing: 'ease-out',
          fill: 'forwards'
        }
      );

      animation.onfinish = () => {
        console.log('Slide-in complete.');
      };
    });
  </script>

</body>
</html>
```

12.3.8 Summary

The Web Animations API bridges the gap between declarative CSS animations and imperative JavaScript control. It's powerful, efficient, and ideal for interactive UIs requiring dynamic motion.

Use WAAPI when:

- You need precise timing or playback control.
- Animations must respond to user input.
- You want to avoid CSS class toggling or inline styles.

Next, we'll build a **fully animated modal dialog** using these concepts to see the Web Animations API in action!

12.4 Practical Example: Animated Modal Dialog

In this section, we'll create a **modal dialog** that opens and closes with smooth animations using the **Web Animations API**. This is a real-world example that demonstrates how to combine DOM manipulation, event handling, and animation to enhance user experience.

12.4.1 What Well Build

- A centered modal dialog that fades and scales into view.
- A semi-transparent overlay behind the modal.
- Dismissal options via a close button or clicking outside the modal.
- JavaScript-powered animation using `element.animate()`.

12.4.2 HTML Structure

CSS:

```
<style>
  body {
    font-family: sans-serif;
    margin: 0;
    padding: 2rem;
  }

  #overlay {
    position: fixed;
    top: 0;
    left: 0;
    width: 100%;
    height: 100%;
    background: rgba(0, 0, 0, 0.5);
    display: none;
    z-index: 10;
  }

  #modal {
    position: fixed;
    top: 50%;
    left: 50%;
    transform: translate(-50%, -50%) scale(0.8);
    background: #fff;
    padding: 2rem;
    border-radius: 8px;
    min-width: 300px;
    box-shadow: 0 0 20px rgba(0, 0, 0, 0.2);
    opacity: 0;
    z-index: 20;
  }

  button {
    margin-top: 1rem;
  }
</style>
```

HTML:

```
<button id="openBtn">Open Modal</button>
```

```html
<div id="overlay"></div>

<div id="modal" role="dialog" aria-modal="true" aria-hidden="true">
  <h2>Animated Modal</h2>
  <p>This modal appears with a smooth animation!</p>
  <button id="closeBtn">Close</button>
</div>
```

12.4.3 JavaScript: Open/Close Logic with Animations

```javascript
const openBtn = document.getElementById('openBtn');
const closeBtn = document.getElementById('closeBtn');
const overlay = document.getElementById('overlay');
const modal = document.getElementById('modal');

// Utility: Animate fade/scale in
function showModal() {
  overlay.style.display = 'block';
  modal.style.display = 'block';
  modal.setAttribute('aria-hidden', 'false');

  // Animate overlay
  overlay.animate([{ opacity: 0 }, { opacity: 1 }], {
    duration: 300,
    fill: 'forwards'
  });

  // Animate modal appearance
  modal.animate(
    [
      { opacity: 0, transform: 'translate(-50%, -50%) scale(0.8)' },
      { opacity: 1, transform: 'translate(-50%, -50%) scale(1)' }
    ],
    {
      duration: 300,
      easing: 'ease-out',
      fill: 'forwards'
    }
  );

  modal.focus(); // for accessibility
}

// Utility: Animate fade/scale out
function hideModal() {
  const fadeOutOverlay = overlay.animate([{ opacity: 1 }, { opacity: 0 }], {
    duration: 200,
    fill: 'forwards'
  });

  const fadeOutModal = modal.animate(
    [
```

```
      { opacity: 1, transform: 'translate(-50%, -50%) scale(1)' },
      { opacity: 0, transform: 'translate(-50%, -50%) scale(0.8)' }
    ],
    {
      duration: 200,
      easing: 'ease-in',
      fill: 'forwards'
    }
  );

  fadeOutModal.onfinish = () => {
    modal.style.display = 'none';
    overlay.style.display = 'none';
    modal.setAttribute('aria-hidden', 'true');
  };
}

// Event listeners
openBtn.addEventListener('click', showModal);
closeBtn.addEventListener('click', hideModal);
overlay.addEventListener('click', hideModal); // Click outside modal to close
```

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<meta name="viewport" content="width=device-width, initial-scale=1" />
<title>Animated Modal Demo</title>
<style>
  body {
    font-family: sans-serif;
    margin: 0;
    padding: 2rem;
  }

  #overlay {
    position: fixed;
    top: 0;
    left: 0;
    width: 100%;
    height: 100%;
    background: rgba(0, 0, 0, 0.5);
    display: none;
    z-index: 10;
  }

  #modal {
    position: fixed;
    top: 50%;
    left: 50%;
    transform: translate(-50%, -50%) scale(0.8);
    background: #fff;
    padding: 2rem;
    border-radius: 8px;
    min-width: 300px;
    box-shadow: 0 0 20px rgba(0, 0, 0, 0.2);
    opacity: 0;
```

```
    z-index: 20;
    display: none;
  }

  button {
    margin-top: 1rem;
    padding: 8px 16px;
    font-size: 1rem;
    cursor: pointer;
  }
</style>
</head>
<body>

<button id="openBtn">Open Modal</button>

<div id="overlay"></div>

<div id="modal" role="dialog" aria-modal="true" aria-hidden="true" tabindex="-1">
  <h2>Animated Modal</h2>
  <p>This modal appears with a smooth animation!</p>
  <button id="closeBtn">Close</button>
</div>

<script>
  const openBtn = document.getElementById('openBtn');
  const closeBtn = document.getElementById('closeBtn');
  const overlay = document.getElementById('overlay');
  const modal = document.getElementById('modal');

  // Animate fade/scale in
  function showModal() {
    overlay.style.display = 'block';
    modal.style.display = 'block';
    modal.setAttribute('aria-hidden', 'false');

    overlay.animate([{ opacity: 0 }, { opacity: 1 }], {
      duration: 300,
      fill: 'forwards'
    });

    modal.animate(
      [
        { opacity: 0, transform: 'translate(-50%, -50%) scale(0.8)' },
        { opacity: 1, transform: 'translate(-50%, -50%) scale(1)' }
      ],
      {
        duration: 300,
        easing: 'ease-out',
        fill: 'forwards'
      }
    );

    modal.focus();
  }

  // Animate fade/scale out
  function hideModal() {
```

```
    const fadeOutOverlay = overlay.animate([{ opacity: 1 }, { opacity: 0 }], {
      duration: 200,
      fill: 'forwards'
    });

    const fadeOutModal = modal.animate(
      [
        { opacity: 1, transform: 'translate(-50%, -50%) scale(1)' },
        { opacity: 0, transform: 'translate(-50%, -50%) scale(0.8)' }
      ],
      {
        duration: 200,
        easing: 'ease-in',
        fill: 'forwards'
      }
    );

    fadeOutModal.onfinish = () => {
      modal.style.display = 'none';
      overlay.style.display = 'none';
      modal.setAttribute('aria-hidden', 'true');
    };
  }

  openBtn.addEventListener('click', showModal);
  closeBtn.addEventListener('click', hideModal);
  overlay.addEventListener('click', hideModal);
</script>

</body>
</html>
```

12.4.4 How It Works

- **Display logic**: We manually set `display: block` for visibility, since the Web Animations API doesn't control that directly.
- **Animation setup**: The `animate()` method is used to create smooth transitions for both the modal and the overlay.
- **Cleanup**: After closing, we hide the elements again using `display: none` inside the `onfinish` callback.
- **Accessibility**: We use `aria-hidden` and `role="dialog"` to support screen readers. Focus is placed inside the modal on open.

12.4.5 Try Extending This

Here are some ways to expand this modal example:

- Add **keyboard accessibility** (e.g., close on `Escape` key).
- Trap focus inside the modal when open.
- Add a **form inside** the modal with live validation.
- Save modal open state to `localStorage` to restore later.
- Support **animation preferences**, e.g., detect `prefers-reduced-motion`.

12.4.6 Summary

This modal dialog example shows how to:

- Use `Element.animate()` for smooth fade/scale effects.
- Handle UI interactions and animation lifecycles cleanly.
- Manage accessibility attributes and focus.
- Build an elegant and reusable interactive UI component.

Combining structure, interaction, and animation results in a polished and professional user experience — all achievable with native browser APIs and clean code.

Chapter 13.

Accessibility and ARIA in DOM Programming

1. Importance of Accessible DOM Manipulation
2. Working with ARIA Attributes
3. Keyboard Navigation and Focus Management
4. Practical Example: Accessible Tab Interface

13 Accessibility and ARIA in DOM Programming

13.1 Importance of Accessible DOM Manipulation

As modern web applications grow increasingly dynamic, **accessibility (a11y)** must be a top priority in DOM programming. Accessibility ensures that people with disabilities—such as those who use screen readers, rely on keyboard navigation, or have cognitive or motor impairments—can use and interact with web content effectively.

13.1.1 Why Accessibility Matters

Inclusive Design

- Over **1 billion people worldwide** have some form of disability. Many rely on **assistive technologies** like screen readers or speech recognition tools.
- Accessible DOM manipulation allows these users to perceive and interact with content just as effectively as others.

Legal & Ethical Responsibility

- **Laws** like the Americans with Disabilities Act (ADA) or Web Content Accessibility Guidelines (WCAG) require accessibility in many countries.
- Ethically, building inclusive web experiences is simply the right thing to do.

13.1.2 Common Accessibility Pitfalls in Dynamic DOM Manipulation

Dynamic DOM changes (adding/removing elements, updating content) can **break accessibility** if not handled correctly. Here are some examples:

Pitfall	Impact
Inserting content without focus management	Screen reader users may not be aware of new content
Using `display: none` to hide important messages without ARIA	Content is not announced to screen readers
Missing or incorrect semantic roles	Assistive tech cannot interpret component purpose
Non-keyboard-accessible interactions (e.g., click-only modals)	Keyboard users cannot navigate or activate components

13.1.3 Example: Accessible vs Inaccessible Alert

Inaccessible Alert

```
const alertBox = document.createElement('div');
alertBox.textContent = "Your session is about to expire!";
document.body.appendChild(alertBox);
```

- Problem: A screen reader won't know this alert appeared.
- The user might miss critical information.

Accessible Alert with ARIA

```
const alertBox = document.createElement('div');
alertBox.setAttribute('role', 'alert');
alertBox.setAttribute('aria-live', 'assertive');
alertBox.textContent = "Your session is about to expire!";
document.body.appendChild(alertBox);
```

- With `role="alert"` or `aria-live="assertive"`, screen readers will announce the alert automatically.
- This ensures timely communication for all users.

13.1.4 Dynamic Content and Screen Readers

If your script updates the DOM (e.g., inserting new form errors, toggling visibility, injecting tooltips), consider the **timing and visibility** for screen reader users:

```
<!-- Screen-reader friendly error message -->
<div id="email-error" role="alert" aria-live="polite" class="visually-hidden">
  Please enter a valid email address.
</div>
```

Then toggle visibility **while preserving semantic meaning**:

```
document.getElementById('email-error').textContent = 'Invalid email';
document.getElementById('email-error').classList.remove('visually-hidden');
```

13.1.5 Best Practices for Accessible DOM Manipulation

- Use semantic HTML first (`<button>`, `<nav>`, `<section>`, etc.).
- Announce dynamic changes with ARIA roles like `role="alert"`, `aria-live`, `aria-hidden`.

- Manage keyboard focus when inserting or revealing elements (`element.focus()`).
- Provide alternatives to pointer input (mouse/touch) with keyboard support (`keydown`, `keyup`).
- Never rely solely on visual cues like color—use text or icons too.

13.1.6 Real-World Impacts

Without Accessibility	With Accessibility
Modal appears, but screen reader stays on background	Focus shifts to modal, user is informed
Form validation messages appear visually but not announced	ARIA live regions announce errors
Tabs change content, but screen readers are unaware	Proper `aria-controls` and `aria-selected` guide assistive tools

13.1.7 Summary

Accessible DOM manipulation ensures **equal access**, improves **usability for all**, and aligns your project with **legal standards**. As you work with the DOM dynamically—creating modals, tabs, alerts, menus—always consider how your changes are communicated **not just visually**, but **semantically**.

In the next section, we'll explore how **ARIA attributes** provide the tools to bridge these gaps and enrich your DOM manipulations with clear, accessible meaning.

13.2 Working with ARIA Attributes

As web interfaces become more interactive and component-based, it's crucial to ensure that custom elements are accessible to all users. **ARIA (Accessible Rich Internet Applications)** is a set of roles, states, and properties designed to bridge the accessibility gaps in dynamic content and custom widgets.

13.2.1 What Is ARIA?

ARIA attributes are used when native HTML elements **can't provide sufficient semantic meaning** or behavior by themselves.

ARIA provides:

- **Roles**: Describe what an element is (`button`, `dialog`, `tablist`, etc.)
- **States**: Represent current conditions (`aria-expanded`, `aria-checked`, etc.)
- **Properties**: Provide relationships (`aria-labelledby`, `aria-controls`, etc.)

  YES Use **semantic HTML whenever possible**, and **ARIA only as a supplement**.

13.2.2 Manipulating ARIA Attributes in JavaScript

You can dynamically add or modify ARIA attributes using standard DOM methods:

```
const menu = document.getElementById('menu');
menu.setAttribute('aria-expanded', 'true');

menu.removeAttribute('aria-hidden');

const isExpanded = menu.getAttribute('aria-expanded'); // "true"
```

This approach ensures screen readers stay informed as your UI changes dynamically.

13.2.3 Common ARIA Roles and States

Role	Purpose
`button`	Identifies an element as a button (useful if not using `<button>`)
`dialog`	Marks a modal dialog that interrupts workflow
`tablist` / `tab` / `tabpanel`	Used together to structure tabbed interfaces
`alert`	Announces important, time-sensitive messages
`navigation`	Denotes a navigational region

State / Property	Description
`aria-expanded`	Indicates if a section (like a dropdown) is expanded
`aria-hidden`	Hides elements from assistive tech
`aria-selected`	Used in tabs, listboxes to mark selection

State / Property	Description
`aria-labelledby` / `aria-describedby`	Associate elements for labels or descriptions

13.2.4 Practical Example: Expandable Menu

HTML:

```
<button id="menuToggle" aria-controls="menu" aria-expanded="false">Menu</button>
<ul id="menu" hidden>
  <li><a href="#">Home</a></li>
  <li><a href="#">About</a></li>
</ul>
```

JavaScript:

```
const toggleBtn = document.getElementById('menuToggle');
const menu = document.getElementById('menu');

toggleBtn.addEventListener('click', () => {
  const expanded = toggleBtn.getAttribute('aria-expanded') === 'true';
  toggleBtn.setAttribute('aria-expanded', String(!expanded));
  menu.hidden = expanded;
});
```

YES **ARIA in Action**:

- Screen readers now know that `#menuToggle` controls an expandable menu.
- The `aria-expanded` attribute reflects the visible state of the menu.

13.2.5 Example: Dialog with ARIA

HTML:

```
<div id="modal" role="dialog" aria-modal="true" aria-labelledby="dialog-title" hidden>
  <h2 id="dialog-title">Subscribe</h2>
  <p>Join our newsletter.</p>
</div>
```

JavaScript:

```
const modal = document.getElementById('modal');
modal.hidden = false;
modal.setAttribute('aria-hidden', 'false');
```

- `role="dialog"` tells screen readers this is a modal.
- `aria-modal="true"` ensures focus is kept inside the dialog.
- `aria-labelledby="dialog-title"` links the heading to the dialog.

13.2.6 Best Practices

- YES Prefer native elements: use `<button>`, `<a>`, `<dialog>`, etc., when possible.
- YES Always keep ARIA states in sync with visual state.
- NO Don't misuse ARIA to change visual behavior (e.g., don't rely on `aria-hidden="true"` to visually hide an element).
- YES Test with screen readers and accessibility tools.

13.2.7 Summary

ARIA attributes are powerful tools for making dynamic interfaces **perceivable and operable** by users with assistive technologies. By using JavaScript to manage roles and states like `aria-expanded`, `aria-hidden`, and `aria-labelledby`, you ensure your applications communicate effectively—beyond just the visual layer.

In the next section, we'll explore how **keyboard navigation and focus management** complement ARIA to provide a fully accessible experience.

13.3 Keyboard Navigation and Focus Management

Keyboard navigation is a vital part of building accessible web interfaces. Users who rely on keyboards—whether due to motor disabilities, screen readers, or personal preference—must be able to navigate, interact with, and control dynamic DOM elements as easily as those using a mouse.

13.3.1 Why Focus Management Matters

Many dynamic UI components—modals, dropdowns, custom tabs—are built with divs and spans instead of native HTML controls. These often lack proper **keyboard interaction** and **focus behavior**, making them unusable for keyboard and screen reader users unless managed explicitly.

13.3.2 The Role of `tabindex`

The `tabindex` attribute controls how elements participate in keyboard tabbing.

Value	Behavior
`0`	Makes a normally non-focusable element focusable (in tab order)
`-1`	Makes an element programmatically focusable, but not tabbable
`> 0`	Rarely used; creates custom tab order (not recommended)

Example:

```
<div tabindex="0">Focusable div</div>
```

This lets users reach the div using the `Tab` key.

13.3.3 Setting Focus Programmatically

You can set focus using JavaScript with `.focus()`:

```
const dialog = document.getElementById('dialog');
dialog.focus(); // Move keyboard focus to the dialog
```

This is useful for:

- Moving focus into modals
- Returning focus to a trigger after closing a menu or dialog
- Guiding users after an error or validation message

13.3.4 Trapping Focus in a Modal

When a modal dialog is open, keyboard users should **not** be able to tab to content behind it. You can trap focus within the modal by cycling between its focusable children.

```
function trapFocus(container) {
  const focusable = container.querySelectorAll('button, [href], input, [tabindex]:not([tabindex="-1"])'
  const first = focusable[0];
  const last = focusable[focusable.length - 1];

  container.addEventListener('keydown', (e) => {
    if (e.key === 'Tab') {
      if (e.shiftKey && document.activeElement === first) {
        e.preventDefault();
        last.focus();
```

```
      } else if (!e.shiftKey && document.activeElement === last) {
        e.preventDefault();
        first.focus();
      }
    }
  });
}
```

Use this when opening a modal:

```
const modal = document.getElementById('modal');
modal.hidden = false;
modal.focus();
trapFocus(modal);
```

13.3.5 Handling Keyboard Events for Navigation

You can make custom components keyboard-accessible by listening for relevant keys:

```
document.addEventListener('keydown', (e) => {
  if (e.key === 'ArrowRight') {
    // move to next item
  } else if (e.key === 'Enter' || e.key === ' ') {
    // activate selected item
  }
});
```

Examples of common keys:

- `Tab` / `Shift + Tab` – Move forward/backward through focusable elements
- `Enter` / `Space` – Activate buttons, checkboxes, etc.
- `Arrow` keys – Navigate menus, sliders, or tablists
- `Escape` – Close modals or menus

13.3.6 Example: Custom Button with Keyboard Support

```
<div role="button" tabindex="0" id="myButton">Click Me</div>
```

```
const btn = document.getElementById('myButton');

btn.addEventListener('keydown', (e) => {
  if (e.key === 'Enter' || e.key === ' ') {
    e.preventDefault();
    console.log('Button activated!');
  }
```

```
});

btn.addEventListener('click', () => {
  console.log('Button clicked!');
});
```

This ensures both keyboard and mouse users can activate the element.

13.3.7 Best Practices

- Always **include `tabindex="0"`** on custom interactive elements.
- Use `.focus()` to guide user interaction (especially in modals or form errors).
- Trap focus within dialogs using keyboard listeners.
- Handle **Enter**, **Space**, and **Arrow** keys appropriately based on context.
- Test your UI **without a mouse** to identify accessibility issues.

13.3.8 Summary

Good keyboard navigation is essential for accessibility and usability. By combining `tabindex`, focus control, and key event handling, you ensure that users who rely on the keyboard can access and operate every part of your web application. In the next section, we'll apply these principles to build a fully **accessible tab interface** that supports both mouse and keyboard interaction.

13.4 Practical Example: Accessible Tab Interface

A **tabbed interface** is a common UI pattern used to show different content panels under labeled tabs. To make it accessible, we must combine correct ARIA roles, keyboard navigation, and focus management to support screen readers and keyboard-only users.

13.4.1 Key Accessibility Requirements

- **ARIA Roles**:
  - `role="tablist"` on the container
  - `role="tab"` on each tab
  - `role="tabpanel"` on each associated panel

- **ARIA States**:
  - `aria-selected`, `aria-controls`, `aria-labelledby`, and `tabindex`
- **Keyboard Support**:
  - Arrow keys to move between tabs
  - `Enter` or `Space` to activate a tab

13.4.2 Complete Example: Accessible Tabs

HTML

```
<div class="tabs" role="tablist" aria-label="Sample Tabs">
  <button role="tab" id="tab-1" aria-selected="true" aria-controls="panel-1" tabindex="0">Tab One</butt
  <button role="tab" id="tab-2" aria-selected="false" aria-controls="panel-2" tabindex="-1">Tab Two</bu
  <button role="tab" id="tab-3" aria-selected="false" aria-controls="panel-3" tabindex="-1">Tab Three</
</div>

<div id="panel-1" role="tabpanel" aria-labelledby="tab-1">Content for Tab One</div>
<div id="panel-2" role="tabpanel" aria-labelledby="tab-2" hidden>Content for Tab Two</div>
<div id="panel-3" role="tabpanel" aria-labelledby="tab-3" hidden>Content for Tab Three</div>
```

CSS (for visual feedback)

```
[role="tab"][aria-selected="true"] {
  font-weight: bold;
  border-bottom: 2px solid #333;
}
[role="tabpanel"] {
  margin-top: 1rem;
}
```

JavaScript

```
const tabs = document.querySelectorAll('[role="tab"]');
const tabList = document.querySelector('[role="tablist"]');

tabList.addEventListener('keydown', (e) => {
  const keys = ['ArrowLeft', 'ArrowRight', 'Home', 'End'];
  if (!keys.includes(e.key)) return;

  const currentTab = document.activeElement;
  let newIndex;

  const tabArray = Array.from(tabs);
  const currentIndex = tabArray.indexOf(currentTab);

  switch (e.key) {
```

```
    case 'ArrowLeft':
      newIndex = currentIndex === 0 ? tabArray.length - 1 : currentIndex - 1;
      break;
    case 'ArrowRight':
      newIndex = currentIndex === tabArray.length - 1 ? 0 : currentIndex + 1;
      break;
    case 'Home':
      newIndex = 0;
      break;
    case 'End':
      newIndex = tabArray.length - 1;
      break;
  }

  tabArray[newIndex].focus();
});

tabs.forEach(tab => {
  tab.addEventListener('click', () => activateTab(tab));
  tab.addEventListener('keydown', (e) => {
    if (e.key === 'Enter' || e.key === ' ') {
      e.preventDefault();
      activateTab(tab);
    }
  });
});

function activateTab(tab) {
  // Update tab attributes
  tabs.forEach(t => {
    t.setAttribute('aria-selected', false);
    t.setAttribute('tabindex', -1);
  });

  tab.setAttribute('aria-selected', true);
  tab.setAttribute('tabindex', 0);
  tab.focus();

  // Show/hide panels
  const selectedPanelId = tab.getAttribute('aria-controls');
  document.querySelectorAll('[role="tabpanel"]').forEach(panel => {
    panel.hidden = panel.id !== selectedPanelId;
  });
}
```

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<meta name="viewport" content="width=device-width, initial-scale=1" />
<title>Accessible Tabs Example</title>
<style>
  [role="tab"][aria-selected="true"] {
    font-weight: bold;
    border-bottom: 2px solid #333;
  }
  [role="tab"] {
```

```
    cursor: pointer;
    padding: 0.5rem 1rem;
    border: none;
    background: none;
    font-size: 1rem;
  }
  [role="tab"]:focus {
    outline: 2px solid #007acc;
    outline-offset: 2px;
  }
  [role="tabpanel"] {
    margin-top: 1rem;
    padding: 0.5rem;
    border: 1px solid #ccc;
  }
  .tabs {
    display: flex;
    gap: 1rem;
  }
</style>
</head>
<body>

<div class="tabs" role="tablist" aria-label="Sample Tabs">
  <button role="tab" id="tab-1" aria-selected="true" aria-controls="panel-1" tabindex="0">Tab One</butt
  <button role="tab" id="tab-2" aria-selected="false" aria-controls="panel-2" tabindex="-1">Tab Two</bu
  <button role="tab" id="tab-3" aria-selected="false" aria-controls="panel-3" tabindex="-1">Tab Three</
</div>

<div id="panel-1" role="tabpanel" aria-labelledby="tab-1">Content for Tab One</div>
<div id="panel-2" role="tabpanel" aria-labelledby="tab-2" hidden>Content for Tab Two</div>
<div id="panel-3" role="tabpanel" aria-labelledby="tab-3" hidden>Content for Tab Three</div>

<script>
  const tabs = document.querySelectorAll('[role="tab"]');
  const tabList = document.querySelector('[role="tablist"]');

  tabList.addEventListener('keydown', (e) => {
    const keys = ['ArrowLeft', 'ArrowRight', 'Home', 'End'];
    if (!keys.includes(e.key)) return;

    const currentTab = document.activeElement;
    let newIndex;

    const tabArray = Array.from(tabs);
    const currentIndex = tabArray.indexOf(currentTab);

    switch (e.key) {
      case 'ArrowLeft':
        newIndex = currentIndex === 0 ? tabArray.length - 1 : currentIndex - 1;
        break;
      case 'ArrowRight':
        newIndex = currentIndex === tabArray.length - 1 ? 0 : currentIndex + 1;
        break;
      case 'Home':
        newIndex = 0;
        break;
      case 'End':
```

```
        newIndex = tabArray.length - 1;
        break;
    }

    tabArray[newIndex].focus();
    e.preventDefault();
  });

  tabs.forEach(tab => {
    tab.addEventListener('click', () => activateTab(tab));
    tab.addEventListener('keydown', (e) => {
      if (e.key === 'Enter' || e.key === ' ') {
        e.preventDefault();
        activateTab(tab);
      }
    });
  });

  function activateTab(tab) {
    // Update tab attributes
    tabs.forEach(t => {
      t.setAttribute('aria-selected', 'false');
      t.setAttribute('tabindex', '-1');
    });

    tab.setAttribute('aria-selected', 'true');
    tab.setAttribute('tabindex', '0');
    tab.focus();

    // Show/hide panels
    const selectedPanelId = tab.getAttribute('aria-controls');
    document.querySelectorAll('[role="tabpanel"]').forEach(panel => {
      panel.hidden = panel.id !== selectedPanelId;
    });
  }
</script>

</body>
</html>
```

13.4.3 Explanation and Accessibility Highlights

1. **ARIA roles/states**:
   - Each `button` uses `role="tab"` and links to its panel with `aria-controls`.
   - Panels reference the tab with `aria-labelledby`.
   - Only the active tab has `aria-selected="true"` and `tabindex="0"`.
2. **Keyboard navigation**:
   - Arrow keys move focus between tabs.
   - `Enter` or `Space` activates the tab and displays its content.

3. **Focus management**:
    - When a tab is activated, `tabindex` and `aria-selected` are updated, and the corresponding panel is shown.
4. **Screen readers**:
    - The use of `role`, `aria-labelledby`, and `aria-controls` enables screen readers to announce tab relationships correctly.

13.4.4 Ideas to Extend

Encourage readers to:

- Add screen reader announcements using `aria-live`.
- Style the selected tab more distinctly.
- Store selected tab in `localStorage` to persist state on reload.
- Make the tab interface responsive for small screens (e.g., converting to accordion).

13.4.5 Summary

This practical example brings together **ARIA semantics**, **focus control**, and **keyboard navigation** to deliver a robust, accessible tabbed interface. Building components like this improves usability for all users and ensures your applications are inclusive and compliant with accessibility standards.

Chapter 14.

Performance Optimization Techniques

1. Minimizing Reflows and Repaints
2. Using `requestAnimationFrame` for Smooth Updates
3. Batch DOM Changes Using Document Fragments
4. Practical Example: Virtual Scrolling for Large Lists

14 Performance Optimization Techniques

14.1 Minimizing Reflows and Repaints

14.1.1 What Are Reflows and Repaints?

When the browser renders a web page, it performs two key processes:

- **Reflow (Layout):** The browser calculates the position and size of elements in the document. Any change that affects the geometry of elements—such as adding, removing, or resizing DOM nodes—triggers a reflow.
- **Repaint (Redraw):** After layout is recalculated, the browser updates the pixels on the screen to reflect visual changes like color, visibility, or style changes that don't affect layout.

Reflows are usually more expensive than repaints because they involve recalculating the entire or partial layout tree.

14.1.2 How Do Reflows and Repaints Affect Performance?

Excessive or forced reflows and repaints can cause the page to lag or stutter, especially when manipulating the DOM frequently or dealing with complex layouts. This degrades user experience, particularly on slower devices or during animations.

14.1.3 Common DOM Operations That Trigger Reflows

- Adding or removing DOM elements (e.g., `appendChild`, `removeChild`)
- Changing layout-related CSS properties (e.g., `width`, `height`, `margin`, `padding`, `position`)
- Modifying the DOM tree structure
- Accessing certain layout-related properties forces synchronous reflow (e.g., `offsetWidth`, `scrollHeight`, `getComputedStyle()`)

14.1.4 Avoiding Forced Synchronous Layouts (Layout Thrashing)

Layout thrashing occurs when code repeatedly reads layout properties after making DOM changes, causing the browser to reflow multiple times unnecessarily.

Example of layout thrashing:

```
const list = document.querySelector('#list');
for (let i = 0; i < 100; i++) {
  list.style.height = `${i * 2}px`;          // DOM write triggers reflow
  console.log(list.offsetHeight);             // DOM read forces reflow
}
```

Here, the browser performs a reflow on every iteration because it must compute the updated layout before reading `offsetHeight`.

14.1.5 Tips to Minimize Reflows and Repaints

Batch DOM Writes and Reads Separately

Group all DOM **writes** (style changes, adding/removing nodes) together, then do all DOM **reads** afterward. This prevents the browser from having to reflow between each read/write.

Optimized example:

```
const list = document.querySelector('#list');

// Batch writes
for (let i = 0; i < 100; i++) {
  list.style.height = `${i * 2}px`;
}
// Then batch reads
console.log(list.offsetHeight);
```

Use Document Fragments for Bulk DOM Updates

When adding multiple elements, insert them into a `DocumentFragment` first, then append the fragment once to the DOM. This reduces the number of reflows.

```
const list = document.querySelector('#list');
const fragment = document.createDocumentFragment();

for (let i = 0; i < 100; i++) {
  const item = document.createElement('li');
  item.textContent = `Item ${i}`;
  fragment.appendChild(item);
}

list.appendChild(fragment);  // Single reflow triggered here
```

Avoid Layout-Triggering Property Access During Animation Loops

Properties like `offsetWidth` or `scrollTop` force layout calculations. Avoid accessing them inside tight loops or animations unless necessary.

Use CSS Classes to Trigger Multiple Style Changes

Rather than setting many individual styles via JavaScript, toggle classes that encapsulate multiple style changes.

```
element.classList.add('expanded'); // CSS handles multiple style changes
```

14.1.6 Inefficient vs Optimized Example: Adding List Items

Inefficient approach (many reflows):

```
const list = document.getElementById('list');

for (let i = 0; i < 50; i++) {
  const li = document.createElement('li');
  li.textContent = `Item ${i}`;
  list.appendChild(li);   // Triggers reflow every iteration
}
```

Optimized approach (single reflow):

```
const list = document.getElementById('list');
const fragment = document.createDocumentFragment();

for (let i = 0; i < 50; i++) {
  const li = document.createElement('li');
  li.textContent = `Item ${i}`;
  fragment.appendChild(li);
}

list.appendChild(fragment);  // Only one reflow triggered here
```

14.1.7 Summary

Reflows and repaints are necessary but costly parts of rendering dynamic pages. Understanding what triggers them and how to minimize their frequency is crucial for writing high-performance DOM manipulation code. By batching changes, using document fragments, avoiding layout thrashing, and leveraging CSS classes, you can keep your web applications fast and responsive.

14.2 Using `requestAnimationFrame` for Smooth Updates

14.2.1 What Is `requestAnimationFrame`?

`requestAnimationFrame` is a browser API designed to optimize and synchronize DOM updates and animations with the browser's repaint cycle. Instead of updating the UI at arbitrary intervals, this method lets you schedule your changes to happen right before the browser repaints the screen, resulting in smoother visual updates and better performance.

14.2.2 Why Use `requestAnimationFrame`?

- **Smooth animations:** It allows the browser to optimize animation timing, typically running at 60 frames per second (FPS), matching the display refresh rate.
- **Avoids jank and layout thrashing:** By batching DOM changes just before a repaint, it prevents unnecessary and costly reflows and repaints.
- **Energy efficient:** The browser can throttle or pause animations when tabs are inactive, saving CPU and battery.

14.2.3 How to Use `requestAnimationFrame`

The API accepts a callback function that runs before the next repaint:

```
function update() {
  // Perform DOM updates or animations here
  // Schedule the next update
  requestAnimationFrame(update);
}

// Start the animation loop
requestAnimationFrame(update);
```

14.2.4 Example: Animating an Element's Position with `requestAnimationFrame`

Let's animate a box moving horizontally across the screen.

```
<div id="box" style="width:50px; height:50px; background:blue; position:absolute; left:0;"></div>
```

```
const box = document.getElementById('box');
let pos = 0;
```

```
function animate() {
  pos += 2;                      // Move 2px per frame
  box.style.left = pos + 'px';

  if (pos < window.innerWidth - 50) {
    requestAnimationFrame(animate);
  }
}

requestAnimationFrame(animate);
```

This code moves the box smoothly by syncing updates to the browser's repaint schedule.

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<meta name="viewport" content="width=device-width, initial-scale=1" />
<title>Simple Animation</title>
<style>
  #box {
    width: 50px;
    height: 50px;
    background: blue;
    position: absolute;
    left: 0;
    top: 50px;
  }
</style>
</head>
<body>

<div id="box"></div>

<script>
  const box = document.getElementById('box');
  let pos = 0;

  function animate() {
    pos += 2; // Move 2px per frame
    box.style.left = pos + 'px';

    if (pos < window.innerWidth - 50) {
      requestAnimationFrame(animate);
    }
  }

  requestAnimationFrame(animate);
</script>

</body>
</html>
```

14.2.5 Contrasting with `setTimeout` or `setInterval`

Using `setTimeout` or `setInterval` to update animations can cause inconsistent frame rates and visual stuttering because these timers are not synchronized with repaints.

Example with `setTimeout`:

```
let pos = 0;
const box = document.getElementById('box');

function move() {
  pos += 2;
  box.style.left = pos + 'px';

  if (pos < window.innerWidth - 50) {
    setTimeout(move, 16);  // ~60fps but no guarantee of sync with repaint
  }
}

move();
```

In contrast, `requestAnimationFrame` automatically adjusts to the display refresh rate and pauses animations when the tab is inactive.

14.2.6 Using `requestAnimationFrame` for Throttling Expensive DOM Updates

Besides animations, `requestAnimationFrame` is useful to throttle expensive updates triggered by rapid events like `scroll` or `resize`.

Example:

```
let ticking = false;

window.addEventListener('scroll', () => {
  if (!ticking) {
    requestAnimationFrame(() => {
      // Perform costly DOM update here
      console.log('Scroll event processed');

      ticking = false;
    });

    ticking = true;
  }
});
```

This technique prevents multiple costly updates per frame by ensuring only one update runs before the next repaint.

14.2.7 Summary

Using `requestAnimationFrame` aligns DOM updates and animations with the browser's native repaint timing, resulting in smoother visuals and better performance. It should be preferred over timers for animation and heavy UI updates, helping avoid jank and improve user experience.

14.3 Batch DOM Changes Using Document Fragments

14.3.1 Why Use Document Fragments for Batch DOM Updates?

When you add or modify elements directly in the DOM, each insertion can trigger **reflows** and **repaints**, which are costly operations affecting page performance. If you're adding many elements one by one, these layout recalculations happen repeatedly and can slow down your page significantly.

Document Fragments offer an elegant solution: they are lightweight, off-DOM containers that let you build a subtree of nodes in memory without causing any immediate reflows or repaints. Only when you append the entire fragment to the DOM will all the changes apply at once, triggering a single reflow and repaint.

14.3.2 How Document Fragments Minimize Reflows

- **Manipulate off-DOM:** Changes inside a fragment don't affect the live document, so the browser skips layout calculations during these operations.
- **Single insertion point:** When you append the fragment, its children are inserted into the DOM at once, resulting in a single reflow.
- **Improves responsiveness:** Especially useful when rendering large lists or complex UI components dynamically.

14.3.3 Creating and Using a Document Fragment: Example

Suppose you want to add 1000 list items to a `<ul>` element. Here's the naive approach, which triggers many reflows:

```
const ul = document.getElementById('myList');

for (let i = 0; i < 1000; i++) {
  const li = document.createElement('li');
```

```
  li.textContent = `Item ${i + 1}`;
  ul.appendChild(li);  // Appending directly to DOM each iteration
}
```

This causes the browser to recalculate layout 1000 times—very inefficient.

14.3.4 Optimized Approach with Document Fragment

```
const ul = document.getElementById('myList');
const fragment = document.createDocumentFragment();

for (let i = 0; i < 1000; i++) {
  const li = document.createElement('li');
  li.textContent = `Item ${i + 1}`;
  fragment.appendChild(li);  // Append to fragment off-DOM
}

ul.appendChild(fragment);  // Single DOM insertion triggers only one reflow
```

In this version, only **one** reflow occurs when the entire fragment is appended, greatly improving performance.

14.3.5 Practical Performance Gains

- **Less CPU work:** The browser avoids unnecessary style recalculations and layout thrashing.
- **Faster UI updates:** Especially noticeable on slower devices or with complex DOM trees.
- **Smoother user experience:** Minimizes jank during dynamic content rendering.

14.3.6 Summary

Using **Document Fragments** is a simple yet powerful technique to batch multiple DOM changes. By preparing nodes off-DOM and appending them in bulk, you significantly reduce reflows and repaints, leading to faster, smoother web pages.

14.4 Practical Example: Virtual Scrolling for Large Lists

When working with large lists, rendering all items at once can severely degrade performance, causing slow page loads and laggy scrolling. **Virtual scrolling** is a technique that renders only the items visible in the viewport — dynamically updating the DOM as the user scrolls — vastly improving performance.

In this section, we'll build a simple virtual scrolling list using the concepts you've learned: batching DOM updates with document fragments, using `requestAnimationFrame` for smooth scroll handling, and efficient DOM reuse.

14.4.1 How Virtual Scrolling Works

- **Calculate visible range:** Based on scroll position, determine which items are visible.
- **Render only visible items:** Create DOM nodes for visible items and remove off-screen ones.
- **Reuse DOM nodes:** Instead of destroying and recreating, reuse nodes when possible for efficiency.
- **Batch updates:** Use document fragments to update the DOM in bulk.
- **Throttle scroll events:** Use `requestAnimationFrame` to prevent excessive DOM updates during fast scrolling.

14.4.2 Example Code

Style:

```
<style>
  #scroll-container {
    height: 300px;
    overflow-y: auto;
    border: 1px solid #ccc;
    position: relative;
  }
  #list {
    position: relative;
    margin: 0;
    padding: 0;
    list-style: none;
  }
  #list li {
    height: 30px;
    box-sizing: border-box;
    padding: 5px 10px;
    border-bottom: 1px solid #eee;
    background: #fafafa;
```

```
  }
</style>
```

HTML:

```
<div id="scroll-container">
  <ul id="list"></ul>
</div>
```

Javascrit:

```
<script>
  const container = document.getElementById('scroll-container');
  const list = document.getElementById('list');

  const totalItems = 10000;
  const itemHeight = 30; // Must match CSS li height + padding/border
  const containerHeight = container.clientHeight;
  const visibleCount = Math.ceil(containerHeight / itemHeight) + 1;

  // Set total height to allow scrollbar
  list.style.height = `${totalItems * itemHeight}px`;

  // Pool of reused <li> elements
  const pool = [];

  // Track current start index
  let startIndex = 0;

  // Create initial pool of visible nodes
  for (let i = 0; i < visibleCount; i++) {
    const li = document.createElement('li');
    pool.push(li);
    list.appendChild(li);
  }

  function updateList() {
    const scrollTop = container.scrollTop;
    startIndex = Math.floor(scrollTop / itemHeight);

    // Position the <ul> relative to scroll
    list.style.transform = `translateY(${startIndex * itemHeight}px)`;

    // Batch update visible list items
    const fragment = document.createDocumentFragment();
    for (let i = 0; i < visibleCount; i++) {
      const itemIndex = startIndex + i;
      const li = pool[i];
      if (itemIndex < totalItems) {
        li.textContent = `Item #${itemIndex + 1}`;
        fragment.appendChild(li);
      }
    }
    list.innerHTML = ''; // Clear existing
    list.appendChild(fragment);
  }
```

```
  // Initial render
  updateList();

  // Efficiently handle scroll events
  let ticking = false;
  container.addEventListener('scroll', () => {
    if (!ticking) {
      window.requestAnimationFrame(() => {
        updateList();
        ticking = false;
      });
      ticking = true;
    }
  });
</script>
```

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<meta name="viewport" content="width=device-width, initial-scale=1" />
<title>Virtual Scrolling List</title>
<style>
  #scroll-container {
    height: 300px;
    overflow-y: auto;
    border: 1px solid #ccc;
    position: relative;
  }
  #list {
    position: relative;
    margin: 0;
    padding: 0;
    list-style: none;
  }
  #list li {
    height: 30px;
    box-sizing: border-box;
    padding: 5px 10px;
    border-bottom: 1px solid #eee;
    background: #fafafa;
  }
</style>
</head>
<body>

<div id="scroll-container">
  <ul id="list"></ul>
</div>

<script>
  const container = document.getElementById('scroll-container');
  const list = document.getElementById('list');

  const totalItems = 10000;
  const itemHeight = 30; // Must match CSS li height + padding/border
  const containerHeight = container.clientHeight;
```

```
  const visibleCount = Math.ceil(containerHeight / itemHeight) + 1;

  // Set total height to allow scrollbar
  list.style.height = `${totalItems * itemHeight}px`;

  // Pool of reused <li> elements
  const pool = [];

  // Track current start index
  let startIndex = 0;

  // Create initial pool of visible nodes
  for (let i = 0; i < visibleCount; i++) {
    const li = document.createElement('li');
    pool.push(li);
    list.appendChild(li);
  }

  function updateList() {
    const scrollTop = container.scrollTop;
    startIndex = Math.floor(scrollTop / itemHeight);

    // Position the <ul> relative to scroll
    list.style.transform = `translateY(${startIndex * itemHeight}px)`;

    // Batch update visible list items
    for (let i = 0; i < visibleCount; i++) {
      const itemIndex = startIndex + i;
      const li = pool[i];
      if (itemIndex < totalItems) {
        li.textContent = `Item #${itemIndex + 1}`;
        li.style.display = '';
      } else {
        li.style.display = 'none';
      }
    }
  }

  // Initial render
  updateList();

  // Efficiently handle scroll events
  let ticking = false;
  container.addEventListener('scroll', () => {
    if (!ticking) {
      window.requestAnimationFrame(() => {
        updateList();
        ticking = false;
      });
      ticking = true;
    }
  });
</script>

</body>
</html>
```

14.4.3 Explanation

- The container's height is fixed with vertical scrolling enabled.
- The `<ul>` height is set to the full height of all items, allowing the scrollbar to represent the full list size.
- A fixed number of `<li>` elements are created in the **pool** corresponding to visible items plus one extra.
- On scroll, we calculate the **startIndex** of visible items and shift the `<ul>` vertically using `transform`.
- We update the pool's text content dynamically, reusing nodes instead of creating new ones.
- Scroll events are throttled using `requestAnimationFrame` for smooth updates.

14.4.4 Benefits and Extensions

- This technique dramatically reduces the number of DOM nodes rendered at once, improving memory use and responsiveness.
- You can extend this pattern for:
  - **Grid layouts** by calculating rows and columns visible.
  - **Infinite scrolling** by loading more data as the user scrolls.
  - **Dynamic item heights** with more complex calculations or libraries.

14.4.5 Summary

Virtual scrolling efficiently handles large lists by rendering only what's needed. By combining **DOM batching**, **efficient node reuse**, and **scroll event throttling**, you create smooth, performant interfaces even for massive datasets.

Experiment with this example by adding features like item selection, editing, or loading remote data asynchronously!

Chapter 15.

Debugging and Testing DOM Code

1. Using Browser Developer Tools Effectively
2. Debugging DOM Manipulation and Event Handling
3. Writing Unit Tests for DOM-Dependent Code
4. Practical Example: Automated Tests for Interactive Components

15 Debugging and Testing DOM Code

15.1 Using Browser Developer Tools Effectively

Modern browsers come equipped with powerful **Developer Tools** that are essential for inspecting, debugging, and optimizing DOM-based web applications. Learning to leverage these tools effectively can drastically improve your productivity and help you quickly resolve common DOM issues.

15.1.1 Overview of Key Features

1. DOM Inspection and Live Editing

- Inspect and explore the full DOM tree visually.
- View element attributes, inline styles, and computed styles.
- Edit HTML and CSS live in the browser to test fixes instantly without reloading.
- Toggle element visibility or modify classes to experiment with layouts and behavior.

2. Event Listener Breakpoints and Inspection

- View all event listeners attached to elements, categorized by event type (click, keydown, etc.).
- Set breakpoints on events like clicks or attribute modifications to pause execution exactly when changes occur.
- Trace event propagation paths to diagnose bubbling or capturing bugs.

3. Performance Profiling

- Analyze the performance impact of DOM operations and scripts using timeline and profiler tools.
- Identify costly reflows, repaints, and layout thrashing during interactions.
- Detect long-running JavaScript tasks that block UI responsiveness.

15.1.2 Debugging Common DOM Issues

- **Incorrect Selectors:** Use the Elements panel to verify your selectors target the intended nodes. Use `$0` to reference the currently selected element in the Console.

```
// Example: Select currently inspected element in console
console.log($0);
```

- **Event Propagation Bugs:** Use the Event Listeners pane to check if multiple listeners exist on the same element or ancestors. Set breakpoints on event handlers to observe bubbling/capturing behavior and call stacks. You can also call `event.stopPropagation()`

or `event.stopImmediatePropagation()` in the console to test fixes.

- **Style Problems:** Use the Styles and Computed tabs to track down why a style is not applied — often due to specificity or overridden rules. Use the "Force element state" option to test `:hover` or `:focus` styles manually.

15.1.3 Useful Console Commands and Shortcuts

- `$0`, `$1`, … — Reference recently inspected DOM elements.
- `$$('selector')` — Query multiple elements, similar to `document.querySelectorAll()`.
- `monitorEvents($0, 'click')` — Log events fired on the selected element.
- `$0.style` — Access and modify inline styles of the inspected element.
- **Keyboard Shortcuts:**
  - `Ctrl+Shift+C` / `Cmd+Shift+C` — Toggle element picker tool.
  - `F8` — Resume script execution after breakpoint.
  - `Ctrl+Shift+E` — Focus on Console panel.
  - `Ctrl+Shift+M` — Toggle device toolbar for responsive testing.

15.1.4 Pro Tips for Efficient Debugging

- Use **Live Expressions** in the Console to monitor variable or property values dynamically as you step through code.
- Use **DOM Breakpoints** to pause execution when an element's attributes, subtree, or node removal occurs.
- Combine **Source Maps** with minified JS to debug original source files instead of compiled code.
- Group console outputs with `console.group()` and `console.groupEnd()` to organize logs.

15.1.5 Summary

Browser developer tools offer an indispensable suite of features tailored for DOM programming: from inspecting and editing elements to profiling performance and tracing event flow. Mastering these tools lets you rapidly identify and fix bugs, experiment with live edits, and optimize your web applications for the best user experience.

Spend time exploring your browser's devtools — your future debugging self will thank you!

15.2 Debugging DOM Manipulation and Event Handling

Debugging DOM manipulation and event handling can be challenging due to the dynamic and interactive nature of web pages. Effective strategies and tools can help isolate issues, understand event flows, and ensure your code behaves as expected.

15.2.1 Common Problems in DOM Manipulation and Events

- **Event Handler Duplication:** Accidentally attaching multiple listeners to the same element can cause handlers to fire repeatedly, leading to unexpected behavior or performance issues.
- **Memory Leaks:** Persisting references to DOM nodes or event listeners after they're no longer needed can prevent garbage collection, resulting in growing memory use and slower performance over time.
- **Unexpected Reflows:** Manipulating the DOM inefficiently can cause excessive layout recalculations (reflows), making the UI sluggish.
- **Incorrect Event Propagation:** Events might bubble or capture unexpectedly, triggering unintended handlers, or fail to propagate due to incorrect use of `stopPropagation()` or `preventDefault()`.

15.2.2 Strategies for Isolating Bugs

Use Console Logging Thoughtfully

Add `console.log()` statements in your event handlers and DOM manipulation functions to track when and how they execute. For example:

```
button.addEventListener('click', (event) => {
  console.log('Button clicked:', event.target);
});
```

Logging helps confirm whether event listeners are being added or triggered as expected.

Set Breakpoints in Developer Tools

Use **JavaScript breakpoints** inside your event listener functions or manipulation code. When execution pauses, inspect variables, the call stack, and the current DOM state.

You can also add **DOM breakpoints** to pause when an element's attributes change, when nodes are added/removed, or when text content is modified. This is useful to catch unexpected mutations.

Take DOM Snapshots

Use the **Elements panel** to inspect the live DOM and observe changes before and after manipulations. Some browsers offer snapshot tools to compare DOM states over time, helping track down when a bug first appears.

Verify Event Propagation and Order

Use the **Event Listeners panel** in DevTools to see all listeners attached to an element and its ancestors. Set breakpoints on listeners or use `monitorEvents(element, 'click')` in the console to log events as they fire.

To test and control event flow:

- Use `event.stopPropagation()` to stop bubbling or capturing.
- Use `event.preventDefault()` to block default browser behavior, like link navigation or form submission.

Example:

```
link.addEventListener('click', (event) => {
  event.preventDefault(); // Stop the link from navigating
  console.log('Custom link handling');
});
```

15.2.3 Example Debugging Workflow

Suppose a button's click handler seems to fire multiple times unexpectedly:

1. Check your code to ensure the listener isn't added repeatedly (e.g., inside a loop or multiple renders).
2. Add a console log inside the handler to confirm how many times it triggers.
3. Set a breakpoint inside the listener to pause execution and inspect the call stack.
4. Look for duplicated listeners in the Event Listeners panel.
5. Remove redundant listeners using `removeEventListener()` where necessary.

15.2.4 Tips to Avoid Common Pitfalls

- Always **remove event listeners** when elements are removed or no longer needed to prevent leaks.
- Avoid manipulating the DOM inside tight loops without batching changes (see Chapter 14).
- Use **event delegation** to attach fewer listeners on parent elements instead of many on children.

- Regularly profile your page with performance tools to detect reflows and repaints caused by your code.

15.2.5 Summary

Debugging DOM manipulation and event handling is a mix of strategic logging, breakpoint inspection, and tool-assisted exploration. Understanding event propagation, listener attachment, and DOM mutations lets you pinpoint bugs and optimize interactions effectively. Combining these methods helps maintain robust, responsive, and maintainable web applications.

15.3 Writing Unit Tests for DOM-Dependent Code

Testing JavaScript code that manipulates the DOM is essential for ensuring correctness, catching regressions, and maintaining confidence during development. This section introduces popular approaches and tools, explains how to write meaningful tests that simulate user interactions, and provides best practices to keep your tests maintainable and efficient.

15.3.1 Approaches and Tools for DOM Testing

- **Jest with JSDOM:** Jest is a popular testing framework that includes JSDOM, a lightweight, in-memory DOM implementation. This allows you to run DOM tests in Node.js environments without a real browser. JSDOM simulates browser APIs enough for most DOM manipulation tests.
- **Testing Library (e.g., @testing-library/dom):** Testing Library provides utilities focused on testing user interactions and accessibility by querying the DOM in ways closer to how users experience the UI. It encourages writing tests that don't rely on implementation details.
- **Other Tools:** Framework-specific libraries like React Testing Library, Vue Test Utils, and Cypress for end-to-end testing extend these concepts for richer testing environments.

15.3.2 Writing Unit Tests That Simulate DOM Interactions

Unit tests for DOM code typically involve:

1. **Setting up the DOM environment:** Create or render HTML elements for the

test case, either by directly manipulating `document.body.innerHTML` or using helper functions.

2. **Performing actions:** Simulate user interactions like clicks, input changes, or keyboard events by dispatching events on elements.
3. **Asserting the results:** Verify changes to the DOM, element attributes, classes, or internal state using queries and assertions.

15.3.3 Sample Test Examples

Example: Testing a Button Click Handler

Suppose you have a simple function that toggles a class on a button when clicked:

```
// toggleButton.js
export function setupToggleButton(button) {
  button.addEventListener('click', () => {
    button.classList.toggle('active');
  });
}
```

Unit Test:

```
import { setupToggleButton } from './toggleButton';

describe('setupToggleButton', () => {
  let button;

  beforeEach(() => {
    // Set up DOM element for each test
    button = document.createElement('button');
    button.textContent = 'Click me';
    document.body.appendChild(button);
    setupToggleButton(button);
  });

  afterEach(() => {
    // Clean up after test
    button.remove();
  });

  test('toggles active class on click', () => {
    expect(button.classList.contains('active')).toBe(false);

    // Simulate click event
    button.click();

    expect(button.classList.contains('active')).toBe(true);

    button.click();
```

```
    expect(button.classList.contains('active')).toBe(false);
  });
});
```

Example: Testing Input Change and Validation

```
// validateInput.js
export function setupValidation(input, errorMsgEl) {
  input.addEventListener('input', () => {
    if (input.value.length < 3) {
      errorMsgEl.textContent = 'Input must be at least 3 characters.';
    } else {
      errorMsgEl.textContent = '';
    }
  });
}
```

Unit Test:

```
import { setupValidation } from './validateInput';

describe('setupValidation', () => {
  let input, errorMsg;

  beforeEach(() => {
    input = document.createElement('input');
    errorMsg = document.createElement('div');
    document.body.append(input, errorMsg);
    setupValidation(input, errorMsg);
  });

  afterEach(() => {
    input.remove();
    errorMsg.remove();
  });

  test('shows error for short input', () => {
    input.value = 'ab';
    input.dispatchEvent(new Event('input'));

    expect(errorMsg.textContent).toBe('Input must be at least 3 characters.');
  });

  test('clears error for valid input', () => {
    input.value = 'abcd';
    input.dispatchEvent(new Event('input'));

    expect(errorMsg.textContent).toBe('');
  });
});
```

15.3.4 Best Practices for DOM Tests

- **Keep tests focused:** Each test should verify one behavior or outcome for clarity and ease of debugging.
- **Isolate tests:** Reset DOM state before each test to avoid side effects.
- **Avoid testing implementation details:** Focus on user-visible effects rather than internal function calls.
- **Use semantic queries:** Prefer queries like `getByRole`, `getByLabelText`, or `getByText` (available in Testing Library) to simulate real user interactions.
- **Keep tests fast:** Avoid loading heavy dependencies or unnecessary setups. Use JSDOM or similar lightweight environments for unit tests.
- **Mock external dependencies:** If your code fetches data or interacts with APIs, mock those interactions to keep tests reliable and deterministic.

15.3.5 Summary

Testing DOM-dependent code ensures your UI behaves as expected under various user interactions. Leveraging tools like Jest with JSDOM and Testing Library, you can write tests that simulate clicks, inputs, and other events, while asserting the resulting DOM changes. Maintaining clean, focused, and fast tests fosters a healthy development workflow and helps catch bugs early.

15.4 Practical Example: Automated Tests for Interactive Components

In this section, we'll walk through creating automated tests for a simple interactive component—a **toggle button**—that changes its state and updates accessibility attributes when clicked. This example demonstrates how to test rendering, event handling, state changes, and ARIA attributes to ensure your component works correctly and is accessible.

15.4.1 Component: Toggle Button

Our toggle button toggles between "On" and "Off" states and updates `aria-pressed` accordingly.

```
<!-- index.html -->
<button id="toggle-btn" aria-pressed="false">Off</button>

<script>
  const button = document.getElementById('toggle-btn');

  button.addEventListener('click', () => {
    const isPressed = button.getAttribute('aria-pressed') === 'true';
    button.setAttribute('aria-pressed', String(!isPressed));
    button.textContent = isPressed ? 'Off' : 'On';
  });
</script>
```

15.4.2 Writing Automated Tests

We'll use **Jest** with **JSDOM** to simulate the DOM environment and test the component logic.

toggleButton.test.js

```
/**
 * @jest-environment jsdom
 */

describe('Toggle Button Component', () => {
  let button;

  beforeEach(() => {
    // Setup DOM
    document.body.innerHTML = `<button id="toggle-btn" aria-pressed="false">Off</button>`;
    button = document.getElementById('toggle-btn');

    // Setup event listener (same as in actual component)
    button.addEventListener('click', () => {
      const isPressed = button.getAttribute('aria-pressed') === 'true';
      button.setAttribute('aria-pressed', String(!isPressed));
      button.textContent = isPressed ? 'Off' : 'On';
    });
  });

  test('renders with initial state', () => {
    expect(button).toBeInTheDocument();
    expect(button.textContent).toBe('Off');
    expect(button.getAttribute('aria-pressed')).toBe('false');
  });

  test('toggles state on click', () => {
    // First click
    button.click();
    expect(button.textContent).toBe('On');
    expect(button.getAttribute('aria-pressed')).toBe('true');

    // Second click
```

```
    button.click();
    expect(button.textContent).toBe('Off');
    expect(button.getAttribute('aria-pressed')).toBe('false');
  });

  test('is accessible with correct ARIA attributes', () => {
    expect(button.getAttribute('aria-pressed')).toMatch(/true|false/);
  });
});
```

15.4.3 Explanation

- **Setup:** Each test starts with a fresh DOM and event listener attached to the button. This isolation ensures no test interference.
- **Assertions:**
  - We verify the button is rendered correctly with expected initial content and ARIA attribute.
  - We simulate clicks with `.click()` and check that the state (`aria-pressed`) and displayed text update correctly.
  - We confirm accessibility attributes exist and have valid values.

15.4.4 Running Tests and Continuous Integration

- **Run Locally:** Run tests with the command:
  ```
  npx jest toggleButton.test.js
  ```
  or simply `npm test` if Jest is configured in your project.
- **CI Integration:** Add test commands to your CI pipeline (GitHub Actions, Travis CI, CircleCI, etc.) to automatically run tests on push or pull requests, ensuring regressions are caught before deployment.

15.4.5 Extending Tests

- **Edge Cases:** Test rapid clicking, disabled states, or integration with other components.
- **Error Handling:** Simulate failure modes like missing elements or invalid ARIA attributes.
- **Accessibility:** Integrate automated accessibility testing tools (e.g., axe-core) to catch

ARIA violations or contrast issues.

- **Visual Regression:** Use snapshot tests or visual testing tools to catch unexpected style changes.

15.4.6 Summary

Automated testing of interactive DOM components helps maintain quality and accessibility throughout your application lifecycle. By simulating real user actions and verifying state and accessibility attributes, your tests become powerful documentation and guardrails that keep your UI reliable and inclusive.

Chapter 16.

Advanced Topics and Integrations

1. Working with Mutation Observers
2. Integrating with Third-Party Libraries (e.g., jQuery, React Basics)
3. Using the Intersection Observer API
4. Practical Example: Lazy Loading Images and Content

16 Advanced Topics and Integrations

16.1 Working with Mutation Observers

16.1.1 What Are Mutation Observers?

Mutation Observers are a modern, efficient way to watch for changes in the DOM tree. Unlike older techniques such as polling with timers or deprecated DOM mutation events, Mutation Observers provide a performant, event-driven mechanism to detect when nodes are added, removed, or modified in the DOM. This allows your JavaScript to react immediately to changes without wasting CPU cycles constantly checking the DOM.

16.1.2 Why Use Mutation Observers?

Many web applications dynamically update the DOM—for example, adding new content, modifying attributes, or removing elements based on user interaction or asynchronous data. Mutation Observers enable you to:

- Detect when new elements are inserted, enabling dynamic initialization (e.g., attaching event listeners).
- Monitor attribute changes to respond to styling or state updates.
- Track subtree changes to react when any descendant node changes.
- Avoid inefficient polling or repeated DOM queries.

16.1.3 The Mutation Observer API: Key Concepts

A Mutation Observer watches a specific DOM node (called the **target**) and reports **mutations** matching your configuration options.

Creating a Mutation Observer

You create a Mutation Observer by passing a callback function that receives an array of mutation records whenever changes occur:

```
const observer = new MutationObserver((mutationsList, observer) => {
  // mutationsList is an array of MutationRecord objects
  mutationsList.forEach(mutation => {
    console.log(mutation);
  });
});
```

Configuring What to Observe

Use the `.observe()` method to specify the target node and what kinds of changes to watch for:

```
observer.observe(targetNode, {
  childList: true,      // Watch for added or removed child nodes
  attributes: true,     // Watch for attribute changes
  subtree: true,        // Watch entire subtree, not just direct children
  characterData: true,  // Watch for changes in text content
});
```

You can select any combination of these options depending on your needs.

Disconnecting the Observer

When you no longer need to watch the DOM, call:

```
observer.disconnect();
```

to clean up and stop receiving mutation notifications.

16.1.4 MutationRecord Properties

Each mutation record passed to the callback contains detailed information:

- `type`: `"childList"`, `"attributes"`, or `"characterData"`
- `target`: The node affected by the mutation
- `addedNodes`: A `NodeList` of nodes added (only for `childList`)
- `removedNodes`: A `NodeList` of nodes removed (only for `childList`)
- `attributeName`: The attribute name that changed (only for `attributes`)
- `oldValue`: Previous value before the change, if requested

16.1.5 Practical Example: Watching for Added Elements

This example monitors a container and logs whenever a new `<p>` element is added dynamically:

```
<div id="container">
  <p>Initial paragraph.</p>
</div>

<button id="add-btn">Add Paragraph</button>

<script>
  const container = document.getElementById('container');
  const addBtn = document.getElementById('add-btn');
```

```
  // Create a MutationObserver instance
  const observer = new MutationObserver((mutationsList) => {
    mutationsList.forEach(mutation => {
      if (mutation.type === 'childList' && mutation.addedNodes.length > 0) {
        mutation.addedNodes.forEach(node => {
          if (node.nodeName === 'P') {
            console.log('New paragraph added:', node.textContent);
            // You could initialize something here (e.g., attach events)
          }
        });
      }
    });
  });

  // Start observing child node additions in the container
  observer.observe(container, { childList: true });

  // Add paragraphs dynamically when button clicked
  addBtn.addEventListener('click', () => {
    const newPara = document.createElement('p');
    newPara.textContent = 'This is a dynamically added paragraph.';
    container.appendChild(newPara);
  });

  // Optional: disconnect observer when no longer needed
  // observer.disconnect();
</script>
```

16.1.6 Observing Attribute Changes

To detect when attributes change, you can enable the `attributes` option:

```
observer.observe(container, {
  attributes: true,
  attributeOldValue: true,
  attributeFilter: ['class', 'style'], // optional: watch only specific attributes
  subtree: true // watch in descendants too
});
```

Example callback detecting class changes:

```
const observer = new MutationObserver((mutationsList) => {
  mutationsList.forEach(mutation => {
    if (mutation.type === 'attributes') {
      console.log(`Attribute ${mutation.attributeName} changed on`, mutation.target);
      console.log('Old value:', mutation.oldValue);
      console.log('New value:', mutation.target.getAttribute(mutation.attributeName));
    }
  });
});
```

```
<div id="container">
  <p>Initial paragraph.</p>
</div>

<button id="add-btn">Add Paragraph</button>

<script>
  const container = document.getElementById('container');
  const addBtn = document.getElementById('add-btn');

  // Create a MutationObserver instance
  const observer = new MutationObserver((mutationsList) => {
    mutationsList.forEach(mutation => {
      if (mutation.type === 'childList' && mutation.addedNodes.length > 0) {
        mutation.addedNodes.forEach(node => {
          if (node.nodeName === 'P') {
            console.log('New paragraph added:', node.textContent);
            // You could initialize something here (e.g., attach events)
          }
        });
      }
      if (mutation.type === 'attributes') {
        console.log(`Attribute ${mutation.attributeName} changed on`, mutation.target);
        console.log('Old value:', mutation.oldValue);
        console.log('New value:', mutation.target.getAttribute(mutation.attributeName));
      }
    });
  });

  // Start observing child node additions and attribute changes in the container
  observer.observe(container, {
    childList: true,
    attributes: true,
    attributeOldValue: true,
    attributeFilter: ['class', 'style'], // optional: watch only specific attributes
    subtree: true // watch in descendants too
  });

  // Add paragraphs dynamically when button clicked
  addBtn.addEventListener('click', () => {
    const newPara = document.createElement('p');
    newPara.textContent = 'This is a dynamically added paragraph.';
    container.appendChild(newPara);
  });

  // Example: change container's class on click to test attribute observer
  container.addEventListener('click', () => {
    container.classList.toggle('highlight');
  });
</script>
```

16.1.7 Summary

Mutation Observers are a powerful and efficient tool for reacting to changes in the DOM without resorting to inefficient polling. By configuring observers carefully and disconnecting them when done, you can enhance dynamic behavior in your web applications with minimal performance overhead.

Feel free to experiment with different observer options and callbacks to suit your use cases such as live content updates, custom component frameworks, or debugging dynamic UI changes!

16.2 Integrating with Third-Party Libraries (e.g., jQuery, React Basics)

16.2.1 Coexisting with jQuery and React

JavaScript's native DOM API is powerful and flexible, but many developers also use third-party libraries and frameworks to simplify DOM manipulation, event handling, and UI rendering. Understanding how these tools integrate with or differ from vanilla DOM code is important for writing maintainable and efficient applications.

16.2.2 Using jQuery with Vanilla DOM

jQuery has been a hugely popular library for simplifying DOM traversal, event handling, animations, and Ajax calls. It wraps native DOM elements in a jQuery object, providing an easy-to-use API.

Pros of jQuery

- Simplifies complex selectors and cross-browser issues.
- Chainable methods enable concise code.
- Rich event handling and animation utilities.
- Large plugin ecosystem.

Cons of jQuery

- Adds dependency and increases bundle size.
- Can encourage less performant code if misused.
- Overlaps with modern browser APIs that have improved.

jQuery vs. Vanilla JS Examples

Task	jQuery	Vanilla JavaScript
Selecting elements	`$('#myButton')`	`document.getElementById('myButton')`
Adding event listener	`$('#myButton').on('click', handler)`	`button.addEventListener('click', handler)`
Hiding element	`$('#myDiv').hide()`	`document.getElementById('myDiv').style.display = 'none'`
Adding a class	`$('#myDiv').addClass('active')`	`element.classList.add('active')`

Using jQuery can reduce verbosity and simplify some tasks, especially in older projects. However, modern JavaScript often provides similarly straightforward methods, reducing the need for jQuery.

16.2.3 Integrating React and the Virtual DOM

React, a popular front-end framework, uses a **virtual DOM** — an in-memory representation of the real DOM — to efficiently update the UI. Instead of manipulating DOM elements directly, React updates the virtual DOM and applies only the minimal necessary changes to the actual DOM.

Key Differences from Vanilla DOM Manipulation

- **Declarative UI:** React components declare what the UI should look like for a given state, rather than imperatively changing DOM nodes.
- **Virtual DOM Diffing:** React calculates changes between virtual DOM trees and batches updates efficiently.
- **Component-Based:** UI is composed of reusable components with isolated logic and state.
- **Event Handling:** React normalizes events and attaches listeners at the document level for performance.

Example React Component vs. Vanilla DOM

React:

```
function Button({ onClick, label }) {
  return <button onClick={onClick}>{label}</button>;
}
```

Vanilla JS:

```
const button = document.createElement('button');
button.textContent = 'Click me';
button.addEventListener('click', () => alert('Clicked'));
```

```
document.body.appendChild(button);
```

React abstracts away direct DOM manipulation, focusing on state and UI description, which can simplify complex UIs but adds a learning curve and build tooling.

16.2.4 When to Use Libraries vs. Direct DOM Manipulation

Use Case	Recommendation
Small scripts or simple interactions	Vanilla JS for minimal dependencies
Legacy projects or quick prototyping	jQuery for fast, cross-browser support
Complex UIs with many state changes	React or similar frameworks for scalability
When working with frameworks ecosystems	Use the library's recommended patterns
Performance-critical fine-grained DOM control	Direct DOM manipulation may be needed

16.2.5 Best Practices for Integration

- Avoid mixing direct DOM manipulation on React-managed elements — React expects to manage its DOM subtree exclusively.
- When using jQuery in React projects, isolate jQuery effects to non-React-controlled parts.
- Prefer using native DOM APIs for modern browsers unless a library provides significant advantages.
- Leverage libraries' abstractions to improve code maintainability, but stay aware of underlying DOM operations for debugging and optimization.

16.2.6 Summary

Third-party libraries like jQuery and frameworks like React provide valuable tools for DOM manipulation and UI building but have distinct approaches and trade-offs compared to vanilla DOM code. Knowing when and how to integrate or choose one over the other helps you write more efficient, maintainable, and future-proof web applications.

16.3 Using the Intersection Observer API

16.3.1 What Is the Intersection Observer API?

The **Intersection Observer API** provides an efficient way to asynchronously detect when an element enters or leaves the viewport—or any other ancestor element's visible area. Unlike traditional scroll event listeners, which can be costly and trigger very frequently, Intersection Observer offloads work to the browser's optimized mechanisms, improving performance and battery life.

This API is widely used for features such as:

- **Lazy loading images or content** only when they come into view,
- **Infinite scrolling** by loading more data when the user nears the bottom,
- **Triggering animations or effects** when elements become visible.

16.3.2 How It Works

You create an `IntersectionObserver` instance with a callback function that runs whenever observed elements cross specified visibility thresholds. The observer watches one or more target elements relative to a root element (which defaults to the viewport).

Key configuration options:

Option	Description
`root`	The ancestor element to use as viewport. Defaults to the browser viewport if `null`.
`rootMargin`	Margin around the root, allowing early or delayed triggering (e.g., `"0px 0px 100px 0px"`).
`threshold`	Single number or array (0 to 1) indicating how much of the target must be visible to trigger.

16.3.3 Creating an Intersection Observer

Here is a simple example that logs when a target element becomes visible in the viewport:

```
// Callback function executed when intersections occur
const callback = (entries, observer) => {
  entries.forEach(entry => {
    if (entry.isIntersecting) {
      console.log(`Element ${entry.target.id} is visible.`);
      // You can unobserve if you only want a one-time notification
      observer.unobserve(entry.target);
```

```
    }
  });
};

// Create the observer with options
const options = {
  root: null,              // viewport
  rootMargin: '0px',
  threshold: 0.1           // trigger when 10% of element is visible
};

const observer = new IntersectionObserver(callback, options);

// Target elements to observe
const targets = document.querySelectorAll('.observe-me');
targets.forEach(el => observer.observe(el));
```

16.3.4 Practical Use Case: Lazy Loading Images

Lazy loading defers image loading until they are about to appear in the viewport, saving bandwidth and speeding up initial page load.

```
<img data-src="large-image.jpg" alt="Lazy loaded" class="lazy-image" />
```

```
const lazyImages = document.querySelectorAll('.lazy-image');

const lazyLoadCallback = (entries, observer) => {
  entries.forEach(entry => {
    if (entry.isIntersecting) {
      const img = entry.target;
      img.src = img.dataset.src;
      observer.unobserve(img); // stop observing once loaded
    }
  });
};

const lazyLoadObserver = new IntersectionObserver(lazyLoadCallback, {
  rootMargin: '100px 0px', // load before entering viewport
  threshold: 0.01
});

lazyImages.forEach(img => lazyLoadObserver.observe(img));
```

16.3.5 Additional Applications

- **Infinite Scrolling:** Observe a sentinel element at the bottom of the list and load more items when it becomes visible.

- **Animation Triggers:** Start animations or transitions when a section scrolls into view.
- **Ad or Analytics Tracking:** Detect when users have actually seen specific page elements.

16.3.6 Summary

The Intersection Observer API is a modern, efficient way to track element visibility changes without the overhead of scroll event listeners. By configuring thresholds and root margins, you can finely tune when your callbacks trigger, enabling smooth user experiences such as lazy loading and infinite scrolling.

16.4 Practical Example: Lazy Loading Images and Content

Lazy loading is a powerful technique to defer loading of offscreen images or content until just before they enter the viewport. This greatly improves initial page load time and reduces unnecessary network and CPU usage.

In this example, we'll use the **Intersection Observer API** to implement lazy loading for images. When an image is about to scroll into view, we will replace its placeholder `src` with the actual image URL stored in a `data-src` attribute.

16.4.1 Complete Example: Lazy Loading Images

CSS:

```
<style>
  body {
    font-family: Arial, sans-serif;
  }
  .image-container {
    max-width: 600px;
    margin: 20px auto;
  }
  img {
    display: block;
    width: 100%;
    height: auto;
    margin-bottom: 20px;
    background-color: #f0f0f0; /* placeholder background */
    transition: filter 0.3s ease;
    filter: blur(10px);
  }
```

```
  img.loaded {
    filter: blur(0);
  }
</style>
```

HTML:

```
<h2>Lazy Loading Images Example</h2>

<div class="image-container">
  <!-- Placeholder images with data-src -->
  <img class="lazy" data-src="https://picsum.photos/id/1015/600/400" alt="Mountain" src="https://via.pla
  <img class="lazy" data-src="https://picsum.photos/id/1025/600/400" alt="Dog" src="https://via.placeho
  <img class="lazy" data-src="https://picsum.photos/id/1035/600/400" alt="Forest" src="https://via.place
  <img class="lazy" data-src="https://picsum.photos/id/1045/600/400" alt="Sea" src="https://via.placeho
  <img class="lazy" data-src="https://picsum.photos/id/1055/600/400" alt="Desert" src="https://via.place
</div>
```

```
// Select all images with the 'lazy' class
const lazyImages = document.querySelectorAll('img.lazy');

// Intersection Observer callback
const onIntersection = (entries, observer) => {
  entries.forEach(entry => {
    if (entry.isIntersecting) {
      const img = entry.target;

      // Replace src with data-src to load actual image
      img.src = img.dataset.src;

      // Add event listener to remove blur filter when image loads
      img.onload = () => {
        img.classList.add('loaded');
      };

      // Stop observing this image as it has loaded
      observer.unobserve(img);
    }
  });
};

// Fallback for browsers without Intersection Observer support
const lazyLoadFallback = () => {
  lazyImages.forEach(img => {
    img.src = img.dataset.src;
    img.classList.add('loaded');
  });
};

if ('IntersectionObserver' in window) {
  const observerOptions = {
    root: null,           // viewport
    rootMargin: '100px',  // preload images 100px before entering viewport
    threshold: 0.01
  };
```

```
  const observer = new IntersectionObserver(onIntersection, observerOptions);
  lazyImages.forEach(img => observer.observe(img));
} else {
  // Fallback: Load all images immediately
  lazyLoadFallback();
}
```

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<meta name="viewport" content="width=device-width, initial-scale=1" />
<title>Lazy Loading with Intersection Observer</title>
<style>
  body {
    font-family: Arial, sans-serif;
  }
  .image-container {
    max-width: 600px;
    margin: 20px auto;
  }
  img {
    display: block;
    width: 100%;
    height: auto;
    margin-bottom: 20px;
    background-color: #f0f0f0; /* placeholder background */
    transition: filter 0.3s ease;
    filter: blur(10px);
  }
  img.loaded {
    filter: blur(0);
  }
</style>
</head>
<body>

<h2>Lazy Loading Images Example</h2>

<div class="image-container">
  <!-- Placeholder images with data-src -->
  <img class="lazy" data-src="https://picsum.photos/id/1015/600/400" alt="Mountain" src="https://via.pla
  <img class="lazy" data-src="https://picsum.photos/id/1025/600/400" alt="Dog" src="https://via.placeho
  <img class="lazy" data-src="https://picsum.photos/id/1035/600/400" alt="Forest" src="https://via.place
  <img class="lazy" data-src="https://picsum.photos/id/1045/600/400" alt="Sea" src="https://via.placeho
  <img class="lazy" data-src="https://picsum.photos/id/1055/600/400" alt="Desert" src="https://via.place
</div>

<script>
// Select all images with the 'lazy' class
const lazyImages = document.querySelectorAll('img.lazy');

// Intersection Observer callback
const onIntersection = (entries, observer) => {
  entries.forEach(entry => {
    if (entry.isIntersecting) {
      const img = entry.target;
```

```
      // Replace src with data-src to load actual image
      img.src = img.dataset.src;

      // Add event listener to remove blur filter when image loads
      img.onload = () => {
        img.classList.add('loaded');
      };

      // Stop observing this image as it has loaded
      observer.unobserve(img);
    }
  });
};

// Fallback for browsers without Intersection Observer support
const lazyLoadFallback = () => {
  lazyImages.forEach(img => {
    img.src = img.dataset.src;
    img.classList.add('loaded');
  });
};

if ('IntersectionObserver' in window) {
  const observerOptions = {
    root: null,           // viewport
    rootMargin: '100px',  // preload images 100px before entering viewport
    threshold: 0.01
  };

  const observer = new IntersectionObserver(onIntersection, observerOptions);
  lazyImages.forEach(img => observer.observe(img));
} else {
  // Fallback: Load all images immediately
  lazyLoadFallback();
}
</script>

</body>
</html>
```

16.4.2 Explanation of the Code

- **HTML Setup:** Each image has a small placeholder `src` and the real image URL stored in the `data-src` attribute.
- **CSS:** Images start blurred to indicate loading. Once the image fully loads, a `loaded` class is added that removes the blur smoothly.
- **Intersection Observer:**
  - Observes each lazy image.
  - When an image is about to enter the viewport (`rootMargin: '100px'` means

100px before visible), the actual image URL is assigned to the `src`.
  - The image loads and triggers the `onload` event, removing the blur.
  - The image is then unobserved to avoid unnecessary callbacks.
- **Fallback:** For older browsers without support, all images load immediately to ensure usability.

16.4.3 Extending This Pattern

- **Videos:** Use the same technique with `<video>` elements by setting `src` or `srcObject` dynamically.
- **Ads or Widgets:** Load heavy third-party scripts only when visible.
- **Infinite Scrolling:** Load additional content when the user scrolls near the bottom.
- **Accessibility:** Remember to include meaningful alt text for images to maintain accessibility.

16.4.4 Summary

Using Intersection Observer for lazy loading offscreen images or content greatly improves initial page speed and user experience. This method defers resource loading until truly needed, saving bandwidth and CPU time.

Experiment with different root margins, thresholds, and apply this pattern to other heavy resources for efficient, performant web apps.

Chapter 17.

Real-World Projects and Use Cases

1. Building a Dynamic To-Do List Application
2. Creating a Custom Modal Popup
3. Developing an Image Carousel with Keyboard Support
4. Interactive Form with Validation and Feedback

17 Real-World Projects and Use Cases

17.1 Building a Dynamic To-Do List Application

Creating a dynamic to-do list is a classic project that ties together many core DOM programming concepts: creating and removing elements, handling events efficiently, and persisting state. In this section, we'll build a fully functional to-do list with features including adding new items, editing existing ones, deleting items, marking items as completed, and saving the list in `localStorage` for persistence.

17.1.1 Features Overview

- **Add** new to-do items via an input form.
- **Edit** to-do items inline.
- **Delete** items.
- **Toggle completion** status.
- **Persist** list state across page reloads.
- **Accessible** with proper ARIA roles and keyboard support.
- **User feedback** with visual and textual cues.

17.1.2 HTML Structure

```
<div class="todo-app" role="application" aria-label="To-Do List Application">
  <h1>My To-Do List</h1>
  <form id="todo-form" aria-describedby="form-desc">
    <input type="text" id="todo-input" aria-label="New to-do item" placeholder="Add a new task" required
    <button type="submit">Add</button>
    <p id="form-desc">Type a task and press Add or Enter.</p>
  </form>
  <ul id="todo-list" role="list" aria-live="polite" aria-relevant="additions removals"></ul>
</div>
```

17.1.3 JavaScript: Modular and Commented

```
// Immediately Invoked Function Expression to avoid polluting global scope
(() => {
  const form = document.getElementById('todo-form');
  const input = document.getElementById('todo-input');
  const list = document.getElementById('todo-list');
```

```
// Load saved todos from localStorage or start with empty array
let todos = JSON.parse(localStorage.getItem('todos')) || [];

// Save todos array to localStorage
const saveTodos = () => {
  localStorage.setItem('todos', JSON.stringify(todos));
};

// Render the entire list from todos array
const renderTodos = () => {
  list.innerHTML = ''; // Clear existing list

  // Create a document fragment for performance
  const fragment = document.createDocumentFragment();

  todos.forEach((todo, index) => {
    const li = document.createElement('li');
    li.setAttribute('role', 'listitem');
    li.dataset.index = index;

    // Checkbox for completion toggle
    const checkbox = document.createElement('input');
    checkbox.type = 'checkbox';
    checkbox.checked = todo.completed;
    checkbox.setAttribute('aria-label', `Mark task "${todo.text}" as completed`);
    checkbox.addEventListener('change', toggleComplete);

    // Editable span for task text
    const span = document.createElement('span');
    span.textContent = todo.text;
    span.contentEditable = true;
    span.setAttribute('aria-label', `Edit task "${todo.text}"`);
    span.addEventListener('blur', editTodo);
    span.addEventListener('keydown', (e) => {
      if (e.key === 'Enter') {
        e.preventDefault();
        span.blur(); // Commit edit on Enter
      }
    });

    // Delete button
    const deleteBtn = document.createElement('button');
    deleteBtn.textContent = 'Delete';
    deleteBtn.setAttribute('aria-label', `Delete task "${todo.text}"`);
    deleteBtn.addEventListener('click', deleteTodo);

    // Styling for completed tasks
    if (todo.completed) {
      span.style.textDecoration = 'line-through';
      span.style.color = '#999';
    }

    li.appendChild(checkbox);
    li.appendChild(span);
    li.appendChild(deleteBtn);
    fragment.appendChild(li);
  });
```

```
    list.appendChild(fragment);
  };

  // Add new todo item
  const addTodo = (e) => {
    e.preventDefault();
    const text = input.value.trim();
    if (text === '') return;

    todos.push({ text, completed: false });
    saveTodos();
    renderTodos();
    input.value = '';
    input.focus();
  };

  // Toggle completion status
  const toggleComplete = (e) => {
    const index = e.target.parentElement.dataset.index;
    todos[index].completed = e.target.checked;
    saveTodos();
    renderTodos();
  };

  // Edit todo text on blur
  const editTodo = (e) => {
    const li = e.target.parentElement;
    const index = li.dataset.index;
    const newText = e.target.textContent.trim();

    if (newText === '') {
      // If empty, delete the todo
      todos.splice(index, 1);
    } else {
      todos[index].text = newText;
    }
    saveTodos();
    renderTodos();
  };

  // Delete todo
  const deleteTodo = (e) => {
    const index = e.target.parentElement.dataset.index;
    todos.splice(index, 1);
    saveTodos();
    renderTodos();
  };

  // Initialize app
  const init = () => {
    renderTodos();
    form.addEventListener('submit', addTodo);
  };

  init();
})();
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1" />
  <title>Accessible To-Do List</title>
  <style>
    body {
      font-family: sans-serif;
      padding: 2rem;
      max-width: 600px;
      margin: auto;
    }
    .todo-app {
      background: #f9f9f9;
      padding: 1.5rem;
      border-radius: 8px;
      box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
    }
    form {
      display: flex;
      gap: 0.5rem;
      margin-bottom: 1rem;
    }
    input[type="text"] {
      flex: 1;
      padding: 0.5rem;
      font-size: 1rem;
    }
    button {
      padding: 0.5rem 1rem;
      font-size: 1rem;
    }
    ul {
      list-style: none;
      padding: 0;
    }
    li {
      display: flex;
      align-items: center;
      gap: 0.5rem;
      padding: 0.5rem 0;
      border-bottom: 1px solid #ddd;
    }
    li span[contenteditable] {
      flex: 1;
      outline: none;
    }
    li button {
      background: #ff4d4d;
      color: white;
      border: none;
      border-radius: 4px;
      cursor: pointer;
    }
    li button:hover {
      background: #cc0000;
    }
```

```
  </style>
</head>
<body>

  <div class="todo-app" role="application" aria-label="To-Do List Application">
    <h1>My To-Do List</h1>
    <form id="todo-form" aria-describedby="form-desc">
      <input type="text" id="todo-input" aria-label="New to-do item" placeholder="Add a new task" requi
      <button type="submit">Add</button>
      <p id="form-desc">Type a task and press Add or Enter.</p>
    </form>
    <ul id="todo-list" role="list" aria-live="polite" aria-relevant="additions removals"></ul>
  </div>

  <script>
    (() => {
      const form = document.getElementById('todo-form');
      const input = document.getElementById('todo-input');
      const list = document.getElementById('todo-list');

      let todos = JSON.parse(localStorage.getItem('todos')) || [];

      const saveTodos = () => {
        localStorage.setItem('todos', JSON.stringify(todos));
      };

      const renderTodos = () => {
        list.innerHTML = '';
        const fragment = document.createDocumentFragment();

        todos.forEach((todo, index) => {
          const li = document.createElement('li');
          li.setAttribute('role', 'listitem');
          li.dataset.index = index;

          const checkbox = document.createElement('input');
          checkbox.type = 'checkbox';
          checkbox.checked = todo.completed;
          checkbox.setAttribute('aria-label', `Mark task "${todo.text}" as completed`);
          checkbox.addEventListener('change', toggleComplete);

          const span = document.createElement('span');
          span.textContent = todo.text;
          span.contentEditable = true;
          span.setAttribute('aria-label', `Edit task "${todo.text}"`);
          span.addEventListener('blur', editTodo);
          span.addEventListener('keydown', (e) => {
            if (e.key === 'Enter') {
              e.preventDefault();
              span.blur();
            }
          });

          const deleteBtn = document.createElement('button');
          deleteBtn.textContent = 'Delete';
          deleteBtn.setAttribute('aria-label', `Delete task "${todo.text}"`);
          deleteBtn.addEventListener('click', deleteTodo);
```

```
      if (todo.completed) {
        span.style.textDecoration = 'line-through';
        span.style.color = '#999';
      }

      li.appendChild(checkbox);
      li.appendChild(span);
      li.appendChild(deleteBtn);
      fragment.appendChild(li);
    });

    list.appendChild(fragment);
  };

  const addTodo = (e) => {
    e.preventDefault();
    const text = input.value.trim();
    if (text === '') return;

    todos.push({ text, completed: false });
    saveTodos();
    renderTodos();
    input.value = '';
    input.focus();
  };

  const toggleComplete = (e) => {
    const index = e.target.parentElement.dataset.index;
    todos[index].completed = e.target.checked;
    saveTodos();
    renderTodos();
  };

  const editTodo = (e) => {
    const li = e.target.parentElement;
    const index = li.dataset.index;
    const newText = e.target.textContent.trim();

    if (newText === '') {
      todos.splice(index, 1);
    } else {
      todos[index].text = newText;
    }
    saveTodos();
    renderTodos();
  };

  const deleteTodo = (e) => {
    const index = e.target.parentElement.dataset.index;
    todos.splice(index, 1);
    saveTodos();
    renderTodos();
  };

  const init = () => {
    renderTodos();
    form.addEventListener('submit', addTodo);
  };
```

```
      init();
    })();
  </script>
</body>
</html>
```

17.1.4 Accessibility Considerations

- Use of `role="application"` to indicate an interactive widget.
- ARIA labels on form input and buttons for screen readers.
- `aria-live="polite"` on the list to announce additions or removals.
- Keyboard support for editing with Enter to commit.
- Focus management: after adding, focus returns to input field for rapid entry.

17.1.5 User Feedback

- Visual strike-through and color change for completed items.
- Immediate updates on adding, editing, deleting, or toggling.
- Inline editing to reduce context switching.

17.1.6 Extending the Application

- **Add categories or priorities** for tasks.
- **Filter views**: show all, active, or completed tasks.
- **Syncing across tabs** using `storage` events.
- **Undo/redo** functionality.
- **Drag-and-drop reordering** with `dragstart`/`dragend` events.

17.1.7 Summary

This to-do list example demonstrates key DOM programming techniques like dynamic element creation, event delegation via listeners attached to created elements, persistent state management using `localStorage`, and accessibility best practices. The modular, commented code helps maintain and extend functionality easily.

Try building on this foundation to create more advanced task managers or interactive web apps!

17.2 Creating a Custom Modal Popup

A modal popup is a common UI pattern used to display content on top of the current page, requiring user interaction before returning to the main content. Building a reusable, accessible modal popup from scratch using DOM APIs involves creating the structure dynamically, managing visibility with animations, trapping focus inside the modal, and handling keyboard and mouse events for seamless user experience.

17.2.1 Key Features of Our Modal Popup

- Dynamic creation of modal elements.
- Show and hide modal with smooth CSS transitions.
- Focus trapping to keep keyboard navigation inside the modal.
- Close modal via close button, clicking outside modal content (overlay), and pressing ESC key.
- Accessibility: ARIA roles, focus management, and keyboard support.

17.2.2 Step 1: HTML Setup

We start with a simple page structure and a button to trigger the modal:

```
<button id="open-modal-btn" aria-haspopup="dialog" aria-controls="custom-modal">
  Open Modal
</button>

<!-- Modal container will be created dynamically -->
```

17.2.3 Step 2: CSS Styles for Modal and Transitions

Add styles to position the modal overlay and content, and animate opacity and scale for smooth show/hide:

```
/* Modal overlay */
.modal-overlay {
  position: fixed;
  top: 0;
  left: 0;
  width: 100vw;
  height: 100vh;
  background: rgba(0,0,0,0.5);
  display: flex;
```

```
  justify-content: center;
  align-items: center;
  opacity: 0;
  pointer-events: none;
  transition: opacity 0.3s ease;
  z-index: 1000;
}

/* Visible state */
.modal-overlay.active {
  opacity: 1;
  pointer-events: auto;
}

/* Modal content */
.modal-content {
  background: white;
  padding: 1.5rem;
  border-radius: 8px;
  max-width: 500px;
  width: 90%;
  transform: scale(0.9);
  transition: transform 0.3s ease;
}

/* Visible modal content */
.modal-overlay.active .modal-content {
  transform: scale(1);
}

/* Close button */
.modal-close-btn {
  position: absolute;
  top: 0.5rem;
  right: 0.75rem;
  background: transparent;
  border: none;
  font-size: 1.5rem;
  cursor: pointer;
}
```

17.2.4 Step 3: JavaScript for Modal Logic and Accessibility

```
(() => {
  // References
  const openBtn = document.getElementById('open-modal-btn');

  // Variables to hold modal elements and focusable elements inside
  let modalOverlay, modalContent, closeBtn;
  let focusableElements = [];
  let firstFocusable, lastFocusable;
```

```
// Keep track of last focused element before modal opened
let lastFocusedElement;

// Create modal structure dynamically
function createModal() {
  modalOverlay = document.createElement('div');
  modalOverlay.className = 'modal-overlay';
  modalOverlay.id = 'custom-modal';
  modalOverlay.setAttribute('role', 'dialog');
  modalOverlay.setAttribute('aria-modal', 'true');
  modalOverlay.setAttribute('aria-labelledby', 'modal-title');

  modalContent = document.createElement('div');
  modalContent.className = 'modal-content';
  modalContent.tabIndex = -1; // Make modal content focusable for focus trap

  // Close button
  closeBtn = document.createElement('button');
  closeBtn.className = 'modal-close-btn';
  closeBtn.setAttribute('aria-label', 'Close modal');
  closeBtn.innerHTML = '&times;';

  // Modal title and body content
  const title = document.createElement('h2');
  title.id = 'modal-title';
  title.textContent = 'Custom Modal Popup';

  const body = document.createElement('p');
  body.textContent = 'This is a reusable modal popup built with vanilla JavaScript.';

  // Assemble modal
  modalContent.appendChild(closeBtn);
  modalContent.appendChild(title);
  modalContent.appendChild(body);
  modalOverlay.appendChild(modalContent);
  document.body.appendChild(modalOverlay);
}

// Open modal and manage focus
function openModal() {
  lastFocusedElement = document.activeElement;
  modalOverlay.classList.add('active');
  updateFocusableElements();
  firstFocusable.focus();
  document.addEventListener('keydown', trapFocus);
  document.addEventListener('keydown', handleEsc);
}

// Close modal and restore focus
function closeModal() {
  modalOverlay.classList.remove('active');
  document.removeEventListener('keydown', trapFocus);
  document.removeEventListener('keydown', handleEsc);
  lastFocusedElement?.focus();
}

// Update focusable elements inside modal for focus trap
function updateFocusableElements() {
```

```
    const selectors = 'a[href], button:not([disabled]), textarea, input, select, [tabindex]:not([tabind
    focusableElements = modalOverlay.querySelectorAll(selectors);
    firstFocusable = focusableElements[0];
    lastFocusable = focusableElements[focusableElements.length - 1];
  }

  // Trap focus inside modal
  function trapFocus(e) {
    if (e.key !== 'Tab') return;

    if (e.shiftKey) {
      // Shift + Tab
      if (document.activeElement === firstFocusable) {
        e.preventDefault();
        lastFocusable.focus();
      }
    } else {
      // Tab
      if (document.activeElement === lastFocusable) {
        e.preventDefault();
        firstFocusable.focus();
      }
    }
  }

  // Close modal on ESC key
  function handleEsc(e) {
    if (e.key === 'Escape') {
      closeModal();
    }
  }

  // Close modal when clicking overlay outside content
  function handleOverlayClick(e) {
    if (e.target === modalOverlay) {
      closeModal();
    }
  }

  // Initialize modal component and event listeners
  function init() {
    createModal();

    openBtn.addEventListener('click', openModal);
    closeBtn.addEventListener('click', closeModal);
    modalOverlay.addEventListener('click', handleOverlayClick);
  }

  // Run initialization on DOM ready
  document.addEventListener('DOMContentLoaded', init);
})();
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1" />
```

```
<title>Accessible Modal Demo</title>
<style>
  body {
    font-family: sans-serif;
    padding: 2rem;
  }

  button {
    font-size: 1rem;
    padding: 0.5rem 1rem;
  }

  /* Modal overlay */
  .modal-overlay {
    position: fixed;
    top: 0;
    left: 0;
    width: 100vw;
    height: 100vh;
    background: rgba(0,0,0,0.5);
    display: flex;
    justify-content: center;
    align-items: center;
    opacity: 0;
    pointer-events: none;
    transition: opacity 0.3s ease;
    z-index: 1000;
  }

  /* Visible state */
  .modal-overlay.active {
    opacity: 1;
    pointer-events: auto;
  }

  /* Modal content */
  .modal-content {
    position: relative;
    background: white;
    padding: 1.5rem;
    border-radius: 8px;
    max-width: 500px;
    width: 90%;
    transform: scale(0.9);
    transition: transform 0.3s ease;
  }

  .modal-overlay.active .modal-content {
    transform: scale(1);
  }

  /* Close button */
  .modal-close-btn {
    position: absolute;
    top: 0.5rem;
    right: 0.75rem;
    background: transparent;
    border: none;
```

```
      font-size: 1.5rem;
      cursor: pointer;
    }
  </style>
</head>
<body>

  <button id="open-modal-btn" aria-haspopup="dialog" aria-controls="custom-modal">
    Open Modal
  </button>

  <script>
    (() => {
      const openBtn = document.getElementById('open-modal-btn');

      let modalOverlay, modalContent, closeBtn;
      let focusableElements = [];
      let firstFocusable, lastFocusable;
      let lastFocusedElement;

      function createModal() {
        modalOverlay = document.createElement('div');
        modalOverlay.className = 'modal-overlay';
        modalOverlay.id = 'custom-modal';
        modalOverlay.setAttribute('role', 'dialog');
        modalOverlay.setAttribute('aria-modal', 'true');
        modalOverlay.setAttribute('aria-labelledby', 'modal-title');

        modalContent = document.createElement('div');
        modalContent.className = 'modal-content';
        modalContent.tabIndex = -1;

        closeBtn = document.createElement('button');
        closeBtn.className = 'modal-close-btn';
        closeBtn.setAttribute('aria-label', 'Close modal');
        closeBtn.innerHTML = '&times;';

        const title = document.createElement('h2');
        title.id = 'modal-title';
        title.textContent = 'Custom Modal Popup';

        const body = document.createElement('p');
        body.textContent = 'This is a reusable modal popup built with vanilla JavaScript.';

        modalContent.appendChild(closeBtn);
        modalContent.appendChild(title);
        modalContent.appendChild(body);
        modalOverlay.appendChild(modalContent);
        document.body.appendChild(modalOverlay);
      }

      function openModal() {
        lastFocusedElement = document.activeElement;
        modalOverlay.classList.add('active');
        updateFocusableElements();
        firstFocusable.focus();
        document.addEventListener('keydown', trapFocus);
        document.addEventListener('keydown', handleEsc);
```

```
    }

    function closeModal() {
      modalOverlay.classList.remove('active');
      document.removeEventListener('keydown', trapFocus);
      document.removeEventListener('keydown', handleEsc);
      lastFocusedElement?.focus();
    }

    function updateFocusableElements() {
      const selectors = 'a[href], button:not([disabled]), textarea, input, select, [tabindex]:not([tab
      focusableElements = modalOverlay.querySelectorAll(selectors);
      firstFocusable = focusableElements[0];
      lastFocusable = focusableElements[focusableElements.length - 1];
    }

    function trapFocus(e) {
      if (e.key !== 'Tab') return;

      if (e.shiftKey) {
        if (document.activeElement === firstFocusable) {
          e.preventDefault();
          lastFocusable.focus();
        }
      } else {
        if (document.activeElement === lastFocusable) {
          e.preventDefault();
          firstFocusable.focus();
        }
      }
    }

    function handleEsc(e) {
      if (e.key === 'Escape') {
        closeModal();
      }
    }

    function handleOverlayClick(e) {
      if (e.target === modalOverlay) {
        closeModal();
      }
    }

    function init() {
      createModal();

      openBtn.addEventListener('click', openModal);
      closeBtn.addEventListener('click', closeModal);
      modalOverlay.addEventListener('click', handleOverlayClick);
    }

    document.addEventListener('DOMContentLoaded', init);
  })();
</script>

</body>
</html>
```

17.2.5 Explanation of Code

- **Dynamic creation**: The modal elements are created and appended to the document body only once, keeping the HTML clean.
- **Show/hide with animation**: The CSS `.active` class triggers smooth opacity and scale transitions.
- **Focus trapping**: Keyboard navigation is confined to modal elements with the `trapFocus` function to cycle through focusable elements.
- **Keyboard accessibility**: Pressing the ESC key closes the modal.
- **Overlay click**: Clicking outside the modal content closes the modal.
- **ARIA roles**: The modal container has `role="dialog"` and `aria-modal="true"` for screen reader semantics.
- **Focus management**: When opened, focus moves inside the modal; when closed, focus returns to the element that triggered it.

17.2.6 How to Use

Click the **Open Modal** button to show the popup. Try navigating with Tab/Shift+Tab, closing with the close button, clicking outside the modal, or pressing ESC.

17.2.7 Extending the Modal

- Add support for dynamic content injection.
- Implement animations with the Web Animations API.
- Add accessibility enhancements like announcing modal open/close via ARIA live regions.
- Include multiple modals or stacking modals.
- Integrate with forms inside the modal.

This modular, accessible modal popup provides a solid foundation to build advanced UI dialogs with full keyboard and screen reader support.

17.3 Developing an Image Carousel with Keyboard Support

Image carousels are popular UI components for displaying multiple images or pieces of content in a confined space. Creating an accessible, interactive carousel requires careful management of slide visibility, keyboard navigation, ARIA roles, and smooth transitions.

In this section, we will build a fully functional image carousel that supports:

- Navigation via **previous/next buttons**

- Navigation via **keyboard arrow keys**
- Swipe support on touch devices (basic handling)
- Proper **ARIA roles and attributes** for accessibility
- Smooth slide transitions and visible slide indication

17.3.1 Step 1: Basic HTML Structure

```
<div class="carousel" role="region" aria-label="Image Carousel">
  <button class="carousel-btn prev" aria-label="Previous Slide">&#9664;</button>

  <div class="carousel-track-container">
    <ul class="carousel-track">
      <li class="carousel-slide current-slide" tabindex="0" aria-hidden="false">
        <img src="https://picsum.photos/id/1011/600/300" alt="Mountain Landscape">
      </li>
      <li class="carousel-slide" tabindex="-1" aria-hidden="true">
        <img src="https://picsum.photos/id/1015/600/300" alt="Forest Trees">
      </li>
      <li class="carousel-slide" tabindex="-1" aria-hidden="true">
        <img src="https://picsum.photos/id/1016/600/300" alt="River Scene">
      </li>
    </ul>
  </div>

  <button class="carousel-btn next" aria-label="Next Slide">&#9654;</button>
</div>
```

Explanation:

- The outer `div.carousel` has a `role="region"` and descriptive `aria-label` for screen readers.
- Buttons have accessible labels and act as navigation controls.
- The slides are list items inside `.carousel-track`, with only one slide visible at a time.
- The visible slide has `tabindex="0"` to make it keyboard focusable, others have `tabindex="-1"`.
- `aria-hidden` attributes help screen readers ignore non-visible slides.

17.3.2 Step 2: CSS for Layout and Transitions

```
.carousel {
  position: relative;
  max-width: 600px;
  margin: auto;
  overflow: hidden;
}
```

```css
.carousel-track-container {
  overflow: hidden;
}

.carousel-track {
  display: flex;
  transition: transform 0.4s ease;
  padding: 0;
  margin: 0;
  list-style: none;
}

.carousel-slide {
  min-width: 100%;
  user-select: none;
  outline: none;
  opacity: 0;
  pointer-events: none;
  transition: opacity 0.4s ease;
  position: relative;
}

.carousel-slide.current-slide {
  opacity: 1;
  pointer-events: auto;
}

.carousel-slide img {
  width: 100%;
  display: block;
  border-radius: 6px;
}

.carousel-btn {
  position: absolute;
  top: 50%;
  transform: translateY(-50%);
  background: rgba(0,0,0,0.5);
  border: none;
  color: white;
  padding: 0.5rem 1rem;
  cursor: pointer;
  font-size: 1.5rem;
  user-select: none;
  z-index: 10;
  border-radius: 3px;
}

.carousel-btn.prev {
  left: 10px;
}

.carousel-btn.next {
  right: 10px;
}

.carousel-btn:focus {
  outline: 2px solid #fff;
```

```
}
```

17.3.3 Step 3: JavaScript for Interactivity and Accessibility

```
(() => {
  const track = document.querySelector('.carousel-track');
  const slides = Array.from(track.children);
  const prevButton = document.querySelector('.carousel-btn.prev');
  const nextButton = document.querySelector('.carousel-btn.next');

  let currentIndex = 0;

  // Update slide visibility, tabindex, and aria-hidden attributes
  function updateSlides() {
    slides.forEach((slide, index) => {
      if (index === currentIndex) {
        slide.classList.add('current-slide');
        slide.setAttribute('tabindex', '0');
        slide.setAttribute('aria-hidden', 'false');
        slide.focus();
      } else {
        slide.classList.remove('current-slide');
        slide.setAttribute('tabindex', '-1');
        slide.setAttribute('aria-hidden', 'true');
      }
    });
  }

  // Move to previous slide
  function movePrev() {
    currentIndex = (currentIndex - 1 + slides.length) % slides.length;
    updateSlides();
  }

  // Move to next slide
  function moveNext() {
    currentIndex = (currentIndex + 1) % slides.length;
    updateSlides();
  }

  // Keyboard navigation handler
  function handleKeydown(e) {
    switch (e.key) {
      case 'ArrowLeft':
        e.preventDefault();
        movePrev();
        break;
      case 'ArrowRight':
        e.preventDefault();
        moveNext();
        break;
    }
```

```
  }

  // Basic swipe support
  let startX = 0;

  function handleTouchStart(e) {
    startX = e.touches[0].clientX;
  }

  function handleTouchEnd(e) {
    const endX = e.changedTouches[0].clientX;
    const diff = endX - startX;
    if (Math.abs(diff) > 50) {
      if (diff > 0) {
        movePrev();
      } else {
        moveNext();
      }
    }
  }

  // Initialize carousel
  function init() {
    updateSlides();

    prevButton.addEventListener('click', movePrev);
    nextButton.addEventListener('click', moveNext);
    document.addEventListener('keydown', handleKeydown);

    track.addEventListener('touchstart', handleTouchStart, { passive: true });
    track.addEventListener('touchend', handleTouchEnd);
  }

  document.addEventListener('DOMContentLoaded', init);
})();
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0"/>
  <title>Accessible Carousel</title>
  <style>
    .carousel {
      position: relative;
      max-width: 600px;
      margin: auto;
      overflow: hidden;
    }

    .carousel-track-container {
      overflow: hidden;
    }

    .carousel-track {
      display: flex;
      transition: transform 0.4s ease;
```

```
      padding: 0;
      margin: 0;
      list-style: none;
    }

    .carousel-slide {
      min-width: 100%;
      user-select: none;
      outline: none;
      opacity: 0;
      pointer-events: none;
      transition: opacity 0.4s ease;
      position: relative;
    }

    .carousel-slide.current-slide {
      opacity: 1;
      pointer-events: auto;
    }

    .carousel-slide img {
      width: 100%;
      display: block;
      border-radius: 6px;
    }

    .carousel-btn {
      position: absolute;
      top: 50%;
      transform: translateY(-50%);
      background: rgba(0,0,0,0.5);
      border: none;
      color: white;
      padding: 0.5rem 1rem;
      cursor: pointer;
      font-size: 1.5rem;
      user-select: none;
      z-index: 10;
      border-radius: 3px;
    }

    .carousel-btn.prev {
      left: 10px;
    }

    .carousel-btn.next {
      right: 10px;
    }

    .carousel-btn:focus {
      outline: 2px solid #fff;
    }
  </style>
</head>
<body>

  <div class="carousel" role="region" aria-label="Image Carousel">
    <button class="carousel-btn prev" aria-label="Previous Slide">&#9664;</button>
```

```
  <div class="carousel-track-container">
    <ul class="carousel-track">
      <li class="carousel-slide current-slide" tabindex="0" aria-hidden="false">
        <img src="https://picsum.photos/id/1011/600/300" alt="Mountain Landscape">
      </li>
      <li class="carousel-slide" tabindex="-1" aria-hidden="true">
        <img src="https://picsum.photos/id/1015/600/300" alt="Forest Trees">
      </li>
      <li class="carousel-slide" tabindex="-1" aria-hidden="true">
        <img src="https://picsum.photos/id/1016/600/300" alt="River Scene">
      </li>
    </ul>
  </div>

  <button class="carousel-btn next" aria-label="Next Slide">&#9654;</button>
</div>

<script>
  (() => {
    const track = document.querySelector('.carousel-track');
    const slides = Array.from(track.children);
    const prevButton = document.querySelector('.carousel-btn.prev');
    const nextButton = document.querySelector('.carousel-btn.next');

    let currentIndex = 0;

    function updateSlides() {
      slides.forEach((slide, index) => {
        if (index === currentIndex) {
          slide.classList.add('current-slide');
          slide.setAttribute('tabindex', '0');
          slide.setAttribute('aria-hidden', 'false');
          slide.focus();
        } else {
          slide.classList.remove('current-slide');
          slide.setAttribute('tabindex', '-1');
          slide.setAttribute('aria-hidden', 'true');
        }
      });
    }

    function movePrev() {
      currentIndex = (currentIndex - 1 + slides.length) % slides.length;
      updateSlides();
    }

    function moveNext() {
      currentIndex = (currentIndex + 1) % slides.length;
      updateSlides();
    }

    function handleKeydown(e) {
      switch (e.key) {
        case 'ArrowLeft':
          e.preventDefault();
          movePrev();
          break;
        case 'ArrowRight':
```

```
            e.preventDefault();
            moveNext();
            break;
        }
      }

      let startX = 0;

      function handleTouchStart(e) {
        startX = e.touches[0].clientX;
      }

      function handleTouchEnd(e) {
        const endX = e.changedTouches[0].clientX;
        const diff = endX - startX;
        if (Math.abs(diff) > 50) {
          if (diff > 0) {
            movePrev();
          } else {
            moveNext();
          }
        }
      }

      function init() {
        updateSlides();
        prevButton.addEventListener('click', movePrev);
        nextButton.addEventListener('click', moveNext);
        document.addEventListener('keydown', handleKeydown);
        track.addEventListener('touchstart', handleTouchStart, { passive: true });
        track.addEventListener('touchend', handleTouchEnd);
      }

      document.addEventListener('DOMContentLoaded', init);
    })();
  </script>

</body>
</html>
```

17.3.4 Explanation

- The **`updateSlides()`** function marks only the current slide as visible (`current-slide`), focuses it for keyboard users, and sets ARIA attributes appropriately.
- Navigation buttons update the `currentIndex` and call `updateSlides()`.
- Keyboard arrow keys (`ArrowLeft`, `ArrowRight`) trigger previous/next navigation with `preventDefault()` to avoid scrolling the page.
- Basic swipe support uses touchstart and touchend events to detect horizontal swipes, moving slides accordingly.
- All event listeners are registered after DOM content loads.

17.3.5 Accessibility Notes

- The visible slide is focusable (`tabindex=0`), while others are not (`tabindex=-1`), helping keyboard users navigate efficiently.
- `aria-hidden="true"` on hidden slides prevents screen readers from reading offscreen content.
- The carousel container has a meaningful `aria-label`.
- Navigation buttons have descriptive `aria-label`s for screen readers.
- Focus moves automatically to the visible slide on change, supporting keyboard users.

17.3.6 Extending the Carousel

- Add indicators (dots) for direct navigation.
- Implement autoplay with pause on hover or focus.
- Enhance swipe handling with velocity and inertia.
- Add responsive design for different screen sizes.
- Announce slide changes with ARIA live regions for screen reader feedback.

This accessible image carousel provides smooth, keyboard-friendly navigation and can be expanded to fit many UI requirements. Its modular code structure makes it easy to maintain and enhance.

17.4 Interactive Form with Validation and Feedback

Creating forms with real-time validation and dynamic user feedback is key to improving user experience and data quality. In this section, we will build an advanced interactive form that validates inputs as users type or change fields, shows helpful error messages, and provides success feedback on valid submission.

17.4.1 Key Features

- **Real-time validation** on input and change events
- Validation for **required fields**, **email format**, and **password rules**
- Visual feedback: error messages and input field highlighting
- Prevention of form submission if any input is invalid
- Clean form reset and state management

17.4.2 Step 1: HTML Markup

```
<form id="registrationForm" novalidate>
  <div class="form-group">
    <label for="username">Username (required)</label>
    <input type="text" id="username" name="username" required minlength="3" />
    <small class="error-message" aria-live="polite"></small>
  </div>

  <div class="form-group">
    <label for="email">Email (required)</label>
    <input type="email" id="email" name="email" required />
    <small class="error-message" aria-live="polite"></small>
  </div>

  <div class="form-group">
    <label for="password">Password (min 6 chars)</label>
    <input type="password" id="password" name="password" minlength="6" />
    <small class="error-message" aria-live="polite"></small>
  </div>

  <button type="submit">Register</button>
  <button type="reset">Reset</button>

  <p id="formMessage" role="alert" aria-live="assertive"></p>
</form>
```

17.4.3 Step 2: CSS for Visual Feedback

```
.form-group {
  margin-bottom: 1rem;
  position: relative;
}

input {
  display: block;
  width: 100%;
  padding: 0.5rem;
  font-size: 1rem;
  border: 2px solid #ccc;
  border-radius: 4px;
}

input.invalid {
  border-color: #e74c3c;
  background-color: #fdecea;
}

input.valid {
  border-color: #2ecc71;
}
```

```css
.error-message {
  color: #e74c3c;
  font-size: 0.875rem;
  margin-top: 0.25rem;
  height: 1.2em; /* Reserve space */
  visibility: hidden;
}

.error-message.visible {
  visibility: visible;
}

#formMessage {
  margin-top: 1rem;
  font-weight: bold;
}
```

17.4.4 Step 3: JavaScript Validation Logic and UI Updates

```javascript
(() => {
  const form = document.getElementById('registrationForm');
  const inputs = form.querySelectorAll('input');
  const formMessage = document.getElementById('formMessage');

  // Validate a single input field
  function validateInput(input) {
    const errorElement = input.nextElementSibling; // <small> for errors
    let valid = true;
    let message = '';

    if (input.validity.valueMissing) {
      valid = false;
      message = 'This field is required.';
    } else if (input.validity.tooShort) {
      valid = false;
      message = `Must be at least ${input.minLength} characters.`;
    } else if (input.type === 'email' && input.validity.typeMismatch) {
      valid = false;
      message = 'Please enter a valid email address.';
    }

    if (!valid) {
      input.classList.add('invalid');
      input.classList.remove('valid');
      errorElement.textContent = message;
      errorElement.classList.add('visible');
    } else {
      input.classList.remove('invalid');
      input.classList.add('valid');
      errorElement.textContent = '';
      errorElement.classList.remove('visible');
    }
```

```
    return valid;
  }

  // Validate all inputs, returns true if all valid
  function validateForm() {
    let allValid = true;
    inputs.forEach(input => {
      if (!validateInput(input)) allValid = false;
    });
    return allValid;
  }

  // Event handler for input events to validate on the fly
  inputs.forEach(input => {
    input.addEventListener('input', () => {
      validateInput(input);
      formMessage.textContent = '';
    });
  });

  // Handle form submission
  form.addEventListener('submit', (event) => {
    event.preventDefault(); // prevent actual submission

    if (validateForm()) {
      formMessage.textContent = 'Registration successful!';
      formMessage.style.color = '#2ecc71';

      // Optional: Clear form after submission or keep data
      // form.reset();
      // inputs.forEach(input => input.classList.remove('valid'));
    } else {
      formMessage.textContent = 'Please fix errors before submitting.';
      formMessage.style.color = '#e74c3c';
    }
  });

  // Clear validation states on reset
  form.addEventListener('reset', () => {
    inputs.forEach(input => {
      input.classList.remove('invalid', 'valid');
      const errorElement = input.nextElementSibling;
      errorElement.textContent = '';
      errorElement.classList.remove('visible');
    });
    formMessage.textContent = '';
  });
})();
```

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

```
  font-family: sans-serif;
  padding: 2rem;
  max-width: 600px;
  margin: auto;
}

.form-group {
  margin-bottom: 1rem;
  position: relative;
}

label {
  display: block;
  margin-bottom: 0.25rem;
  font-weight: bold;
}

input {
  display: block;
  width: 100%;
  padding: 0.5rem;
  font-size: 1rem;
  border: 2px solid #ccc;
  border-radius: 4px;
}

input.invalid {
  border-color: #e74c3c;
  background-color: #fdecea;
}

input.valid {
  border-color: #2ecc71;
}

.error-message {
  color: #e74c3c;
  font-size: 0.875rem;
  margin-top: 0.25rem;
  height: 1.2em;
  visibility: hidden;
}

.error-message.visible {
  visibility: visible;
}

button {
  margin-right: 1rem;
  padding: 0.5rem 1rem;
  font-size: 1rem;
  border: none;
  border-radius: 4px;
  background: #3498db;
  color: white;
  cursor: pointer;
}
```

```
    button[type="reset"] {
      background: #7f8c8d;
    }

    #formMessage {
      margin-top: 1rem;
      font-weight: bold;
    }
  </style>
</head>
<body>

  <form id="registrationForm" novalidate>
    <div class="form-group">
      <label for="username">Username (required)</label>
      <input type="text" id="username" name="username" required minlength="3" />
      <small class="error-message" aria-live="polite"></small>
    </div>

    <div class="form-group">
      <label for="email">Email (required)</label>
      <input type="email" id="email" name="email" required />
      <small class="error-message" aria-live="polite"></small>
    </div>

    <div class="form-group">
      <label for="password">Password (min 6 chars)</label>
      <input type="password" id="password" name="password" minlength="6" />
      <small class="error-message" aria-live="polite"></small>
    </div>

    <button type="submit">Register</button>
    <button type="reset">Reset</button>

    <p id="formMessage" role="alert" aria-live="assertive"></p>
  </form>

  <script>
    (() => {
      const form = document.getElementById('registrationForm');
      const inputs = form.querySelectorAll('input');
      const formMessage = document.getElementById('formMessage');

      function validateInput(input) {
        const errorElement = input.nextElementSibling;
        let valid = true;
        let message = '';

        if (input.validity.valueMissing) {
          valid = false;
          message = 'This field is required.';
        } else if (input.validity.tooShort) {
          valid = false;
          message = `Must be at least ${input.minLength} characters.`;
        } else if (input.type === 'email' && input.validity.typeMismatch) {
          valid = false;
          message = 'Please enter a valid email address.';
        }
```

```
        if (!valid) {
          input.classList.add('invalid');
          input.classList.remove('valid');
          errorElement.textContent = message;
          errorElement.classList.add('visible');
        } else {
          input.classList.remove('invalid');
          input.classList.add('valid');
          errorElement.textContent = '';
          errorElement.classList.remove('visible');
        }

        return valid;
      }

      function validateForm() {
        let allValid = true;
        inputs.forEach(input => {
          if (!validateInput(input)) allValid = false;
        });
        return allValid;
      }

      inputs.forEach(input => {
        input.addEventListener('input', () => {
          validateInput(input);
          formMessage.textContent = '';
        });
      });

      form.addEventListener('submit', (event) => {
        event.preventDefault();

        if (validateForm()) {
          formMessage.textContent = 'Registration successful!';
          formMessage.style.color = '#2ecc71';
        } else {
          formMessage.textContent = 'Please fix errors before submitting.';
          formMessage.style.color = '#e74c3c';
        }
      });

      form.addEventListener('reset', () => {
        inputs.forEach(input => {
          input.classList.remove('invalid', 'valid');
          const errorElement = input.nextElementSibling;
          errorElement.textContent = '';
          errorElement.classList.remove('visible');
        });
        formMessage.textContent = '';
      });
    })();
  </script>

</body>
</html>
```

17.4.5 Explanation

- Each input field is validated on every user input to provide immediate feedback.
- The built-in browser constraints (`required`, `type="email"`, `minlength`) are checked via the `validity` property.
- Error messages appear dynamically below each field using the `<small>` element with `aria-live="polite"` to announce updates to screen readers.
- The input's border changes color to visually indicate valid or invalid states.
- On form submission, validation runs on all fields; if invalid, submission is prevented and a summary message appears.
- The reset button clears all validation feedback and the form message.

17.4.6 Accessibility Considerations

- Error messages use `aria-live="polite"` so screen readers announce them without interrupting.
- The form message has `role="alert"` and `aria-live="assertive"` to immediately notify users about submission status.
- Inputs maintain semantic roles and use native validation properties to reduce complexity.
- Focus remains on inputs so keyboard users can continue correcting errors seamlessly.

17.4.7 Extending This Example

- Add **custom validation rules** like password strength or matching confirmation.
- Provide **tooltip hints** or icons for guidance.
- Improve **focus management** by moving focus to the first invalid field on submit.
- Integrate with **server-side validation** to handle asynchronous checks (e.g., username availability).
- Add **ARIA live regions** for screen reader announcements on success or failure.

Interactive forms with real-time validation dramatically improve user experience by catching errors early and guiding users to correct inputs smoothly. By combining native validation features with custom JavaScript logic and accessible feedback, you build robust, user-friendly forms ready for modern web applications.