Client Side Programming

James Goudy

Contents

1. JavaScript Introduction

JavaScript (JS) is a versatile programming language that plays a crucial role in making the web dynamic and interactive. It's often referred to as a **scripting language** due to its lightweight nature and ability to be embedded within web pages. Here's a breakdown of its key characteristics:

What it does:

- Adds interactivity to web pages: Think of things like dynamic content updates, animations, user input handling, and form validation.
- Powers web applications: From complex single-page applications (SPAs) like Gmail to data-driven dashboards, JS is at the core of many modern web experiences.
- Goes beyond the web: Node.js allows JS to run outside browsers, powering server-side applications and desktop/mobile tools.

Key advantages:

- **Cross-platform:** Runs consistently on all major browsers and platforms, ensuring wider reach.
- **Beginner-friendly:** Relatively easy to learn compared to other programming languages, making it popular for beginners.
- **Versatile:** Adaptable to various tasks, from simple scripting to full-fledged application development.
- **Large community and ecosystem:** Extensive resources, libraries, and frameworks available for support and development.

Where you encounter it:

- Nearly every website you visit likely uses some JavaScript for interactivity.
- Popular web applications like Gmail, Facebook, and Netflix rely heavily on JS.
- Many mobile apps and desktop tools are built with JavaScript frameworks.

Different flavors:

- **Client-side JavaScript:** Runs directly in the user's web browser.
- **Server-side JavaScript:** Runs on web servers using Node.js, handling server-side logic and database interactions.

2. Short History Of JavaScript

2.1 A Short History of JavaScript: From Mocha to Modern Marvel

JavaScript, a ubiquitous language powering the web, has gone through a fascinating journey since its conception in 1995. Here's a quick overview:

1995: Birth of Mocha - A 10-Day Wonder:

- Brendan Eich, at Netscape, created Mocha (later LiveScript) in just 10 days to add interactivity to web pages.
- Initially focused on simple tasks like form validation and image rollovers.

1995: LiveScript Emerges, then Rebrands:

- Netscape renamed the language to LiveScript for marketing purposes.
- Soon after, "JavaScript" was chosen to capitalize on the popularity of Java (though the languages are distinct).

1996-1997: Standardization and Global Reach:

- JavaScript gained widespread adoption and Netscape submitted it to ECMA for standardization.
- ECMA released the first standardized version, ECMA-262 (or ECMAScript), in 1997.

1998-2000: Browser Wars and JavaScript's Expansion:

- Competition between Netscape and Internet Explorer fueled rapid JavaScript development.
- New features like DOM manipulation and dynamic HTML enhanced interactivity.

2001-2008: The Rise of AJAX and Web 2.0:

- JavaScript combined with Asynchronous JavaScript and XML (AJAX) enabled dynamic web applications.
- Web 2.0 era saw JavaScript powering interactive platforms like YouTube and Gmail.

2009-Present: Modern JavaScript and Beyond:

- ECMAScript standards evolved rapidly, adding features like modules, classes, and arrow functions.
- Node.js emerged, enabling JavaScript for server-side development.
- JavaScript frameworks like React and Angular revolutionized web development.

Looking Ahead:

- Continued evolution of ECMAScript standards with features like WebAssembly for performance optimization.
- JavaScript increasingly gaining ground in desktop and mobile app development.

3. Document Object Model

In a web page, the DOM—short for **Document Object Model**—is the browser's *living* map of your HTML. Your HTML file is the blueprint you hand to the browser; the DOM is what the browser builds from it and keeps in memory so it can be inspected, updated, and interacted with while the page is running. If HTML is a script, the DOM is the stage set everyone actually walks around on.

The DOM is organized like a **tree**, because elements in HTML are nested inside other elements. At the top sits the `document` object, which represents the entire page. Under that is the `<html>` element, then `<head>` and `<body>`, and then everything visible on the page tends to live under `<body>`. Each item in this structure is called a **node**. Most nodes are **element nodes** (tags like `<div>` or `<p>`), but the characters inside an element are represented too (as **text nodes**), and details like attributes (`id`, `class`, `href`) are stored as part of the node's information.

Here's the basic shape, like a family tree for your page:

```
document
└── html
    ├── head
    │   ├── title
    │   └── meta / link / script ...
    └── body
        ├── h1
        │   └── "Welcome"
        ├── p
        │   └── "This is a page."
        └── div#main.container
            ├── button
            │   └── "Click me"
            └── img (src="photo.jpg")
```

Why this matters is the part that unlocks modern web pages: **JavaScript doesn't "edit the HTML file."** It works with the DOM. When you tell JavaScript to change the text of a heading, or hide a section, or add a new item to a list, what you're really doing is changing the DOM node (or nodes). After the DOM changes, the browser updates what you see so the screen matches the current DOM. That's the core loop of interactivity: something happens, code runs, the DOM updates, the display follows.

You interact with the DOM through the `document` object. Conceptually, you do three things over and over: you **select** a node, you **read or change** its data (text, attributes, classes, styles), and you **listen for events** on it. Events are the DOM's way of announcing: "A click happened," "a key was pressed," "a form field changed," "the mouse moved." When you attach an event listener, you're basically giving the page instructions on how to respond when something happens—like wiring a doorbell to a chime.

Here's a simple "what happens when you click" illustration:

```
User clicks button
      ↓
Browser creates a "click" event
      ↓
Event travels to the button's DOM node
      ↓
Your event listener runs (JavaScript)
      ↓
Your code updates the DOM (maybe changes text)
      ↓
Browser re-renders the page to match the new DOM
```

One last detail that saves people a lot of confusion: the DOM is **live**, and it can drift away from the original HTML source file. If JavaScript adds a new `<li>` to a list, the page you're

looking at now contains that list item in the DOM—even if the original HTML file never had it. The DOM is the *current state of the page*; the HTML file is the *starting point*.

If you want one sentence to tattoo on your brain (temporary ink is fine): **The DOM is the browser's object-based tree representation of a web page, and JavaScript makes pages interactive by reading from and writing to that tree.**

4. JavaScript Placement

You can include JavaScript code in an HTML document using the dedicated `<script>` tag. The placement of this tag depends on when you want the JavaScript to load:

1. **Inside the `<head>` section**:
   - Insert the `<script>` tag between the opening `<head>` and closing `</head>` tags. Place your JavaScript code inside it.
   - This ensures that the script is loaded and executed when the page loads. However, it can cause a delay in rendering the page because the browser waits for the script to download and execute before continuing with other HTML parsing.
2. **Inside the `<body>` section**:
   - You can place the `<script>` tags containing your JavaScript anywhere within the `<body>` tags.
   - While this approach allows the page to render faster, it may still cause a slight delay if the script is large or resource-intensive.
3. **Just before the `</body>` close tag**:
   - Some developers prefer placing the `<script>` tags just before the closing `</body>` tag.
   - This ensures that the entire HTML content is parsed and displayed before the JavaScript is loaded.
   - However, be cautious: if the script modifies the page content (e.g., via `document.write()`), it can disrupt the rendering process.

Choose the placement based on your specific requirements and performance considerations.

---

5. Four Ways To Display Output

there are **four primary ways** to display output:

1. **Using `innerHTML` Property**:
    - You can modify the content of an HTML element using the `innerHTML` property.
    - For example:

```
<p id="demo"></p>
<script>
  document.getElementById("demo").innerHTML = 5 + 6;
</script>
```

This sets the content of the paragraph with the ID “demo” to the result of

```
5 + 6
```

, which is 11.

1. **Using `document.write()` Method**:
    - For testing purposes, you can use `document.write()` to directly write content to the HTML output.
    - Example:

```
<script>
  document.write(5 + 6);
</script>
```

> **Caution:**
>
> Note that using
>
> document.write()
>
> ```
> after the HTML document is loaded will replace all existing HTML
> content.
> ```

1. **Using `window.alert()`**:
    - You can display data in an alert box using `window.alert()`.
    - Example:

<script> // window.alert(5 + 6); </script>

```
 (Note: remove the comment when code is in production)
```

You can also skip the

```
 window
```

keyword, as it’s optional:

```
 ```text
 <script>
   // alert(5 + 6);
 </script>
 ```
```

1. **Using `console.log()`**:
    - For debugging purposes, you can log data to the browser console using `console.log()`.
    - Example:

```
<script>
  console.log(5 + 6);
</script>
```

This won't display anything on the page but will log the result to the browser console.

5.1 Lecture Code

```
<!DOCTYPE html>
<!--
Four ways to output information.
Example by James Goudy
-->
<html>
    <head>
        <title>Four Ways To Output</title>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">


    </head>
    <body>
        <h1>Four Ways To Output In JavaScript</h1>

        <script>
            /*
             * In this example, we cannont put the script here. The body
             * runs top down.  Since our script examples writes to specific
             * tags - at this moment in time or location of the code,
             * the h3, p and span tags have not been created yet,
             * so the script would fail - not knowing where to put the tags..
             */
        </script>


        <br>
        <h3 id="myH3"></h3>
        <p id="myParagraph"></p>
        <p>The message is the following: "<span id="mySpan"></span>"</p>


        <script>

            var myMessage = "JavaSript is Fun!";

            //First way is by a popup window using alert
            alert(myMessage);

            //Second way is by using document doc write.
            //NOTE: This way destroys html - use is with caution.
            //It writes excatly where the physical location of the script
            //tag is located in the html.
            document.write(myMessage);

            //The third way is using innerHTML.  This puts the output
            //in between two html tags like span, paragraph, headings, etc.
            //NOTE: You have to identify the tag by id when usign innerHTML
            document.getElementById("myH3").innerHTML = myMessage;
            document.getElementById("myParagraph").innerHTML = myMessage;
            document.getElementById("mySpan").innerHTML = myMessage;

            /*
             * NOTE: innerHTML works with getElementByID. getElementByID
             * identifies the exact tag to use with innerHTML. Also the
             * page runs top down.  With the script tag located here, the
             * h3, p and span tags have already been created so the
```

```
             * script knows where to put the code.
             */


            //The fourth way to output information is to the console.
            //This is handy if you wish to test values, but not
            //display them on the page. In most browsers you can bring
            //the console up by clicking the F12 key and then clicking
            //the console tab.

            console.log("In order to read this you have to open the console");

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

6. Variables

In JavaScript, variables act like named storage containers for data. You can use them to hold all sorts of information you need throughout your program, like numbers, text, or even more complex things.

Here's a breakdown of how JavaScript variables work:

Declaring Variables:

To create a variable, you first need to declare it using one of these keywords:

- `let`: This is the modern and preferred way to declare variables. It creates a variable with block-level scope (more on that later).
- `const`: Similar to `let`, but the value assigned to the variable cannot be changed after it's declared. Use `const` for values that should stay constant throughout your code.
- `var`: This is the older way of declaring variables. It's generally not recommended due to some potential scoping issues, but it's still good to be aware of it in case you encounter older code.

Here's an example of how to declare variables with `let` and `const`:

```
let name = "Alice";
const age = 30;
```

Naming Conventions:

- Variable names can contain letters, numbers, and the underscore character (_).
- They must start with a letter or underscore.
- They are case-sensitive, so `name` and `Name` are considered different variables.
- Avoid using JavaScript keywords as variable names (like `if`, `for`, etc.).

Assigning Values:

Once you declare a variable, you can assign a value to it using the equal sign (=). The value can be a number, a string of text, or even another variable.

```
let message = "Hello!";
let count = 10;
```

Using Variables:

After assigning a value, you can use the variable name throughout your code to refer to that value. This makes your code more readable and easier to maintain.

For example, you could display the value of the `message` variable using an alert box:

```
let message = "Hello!";
alert(message); // This will display "Hello!"
```

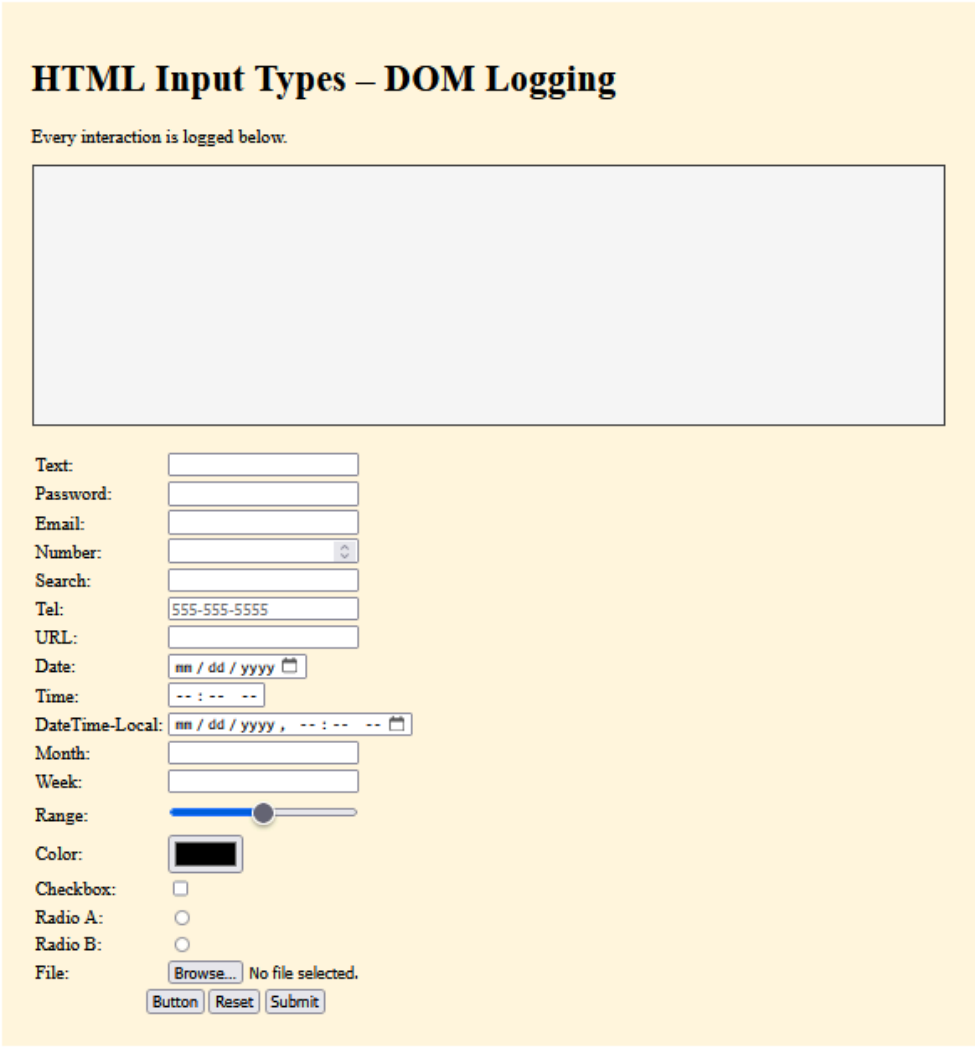
Scope:

The scope of a variable determines where it can be accessed in your code. `let` and `const` variables have block-level scope, meaning they are only accessible within the block of code where they are declared (like an `if` statement or a loop). `var` variables, on the other hand, have function-level scope, meaning they can be accessed from anywhere within the function they are declared in.

In Conclusion:

JavaScript variables are fundamental building blocks for storing and managing data in your programs. By understanding how to declare, name, assign values, and use variables effectively, you'll be well on your way to writing clean and efficient JavaScript code.

7. HTML Input Tag



7.1 Using Inline (inline event handlers)

In the **Inline** version, behavior is attached directly inside the HTML using attributes like `oninput="..."` and `onchange="..."`. That means the HTML doesn't just describe structure—it also tells the browser *what code to run* when something happens. It's immediate and readable for beginners: you can point to a single line and say "this fires the function."

Best use cases: tiny demos, learning exercises, quick one-off prototypes where clarity beats architecture. **Worst use cases:** real applications, shared codebases, or anything you expect to maintain. Inline handlers get messy fast, duplicate logic, and blur separation of concerns. It's also harder to refactor because your behavior is scattered across markup.

```
<!DOCTYPE html>
<!--
  This document demonstrates many HTML <input> types.
  Each input triggers a JavaScript function that logs
  user interaction directly to the DOM.
-->

<head>
  <!-- Page title shown in the browser tab -->
  <title>Input types</title>

  <style>
    /*
      Simple container styling to center the demo
```

```
      and make it visually distinct from the page background
    */
    .div1 {
      width: 75%;
      background-color: cornsilk;
      padding: 25px;
      margin-left: auto;
      margin-right: auto;
      margin-top: 50px;
      margin-bottom: 50px;
    }
  </style>
</head>

<body>

  <!-- Main demo container -->
  <div class="div1">

    <!-- Page heading -->
    <h1>HTML Input Types – DOM Logging</h1>
    <p>Every interaction is logged below.</p>

    <!--
      Log area:
      JavaScript dynamically inserts messages here
      instead of using console.log()
    -->
    <div id="log" style="border:1px solid #444;
      padding:10px;
      height:200px;
      overflow-y:auto;
      margin-bottom: 20px;
      background:#f9f9f9;">
    </div>

    <!--
      Demo form:
      Each input element demonstrates a different input type.
      Event handlers call JavaScript functions on change or input.
    -->
    <form id="demoForm">
      <table>

        <!-- Text input (fires on every keystroke) -->
        <tr>
          <td><label for="textInput">Text:</label></td>
          <td><input type="text" id="textInput"
oninput="handleText(this.value)"></td>
        </tr>

        <!-- Password input (value intentionally hidden in logs) -->
        <tr>
          <td><label for="passwordInput">Password:</label></td>
          <td><input type="password" id="passwordInput"
onchange="handlePassword()"></td>
        </tr>

        <!-- Email input with built-in browser validation -->
        <tr>
          <td><label for="emailInput">Email:</label></td>
          <td><input type="email" id="emailInput"
onchange="handleEmail(this.value)"></td>
        </tr>

        <!-- Numeric input -->
```

```
        <tr>
          <td><label for="numberInput">Number:</label></td>
          <td><input type="number" id="numberInput"
onchange="handleNumber(this.value)"></td>
        </tr>

        <!-- Search input -->
        <tr>
          <td><label for="searchInput">Search:</label></td>
          <td><input type="search" id="searchInput"
oninput="handleSearch(this.value)"></td>
        </tr>

        <!-- Telephone input with a validation pattern -->
        <tr>
          <td><label for="telInput">Tel:</label></td>
          <td>
            <input type="tel" id="telInput"
              onchange="handleTel(this.value)"
              placeholder="555-555-5555"
              pattern="[0-9]{3}-[0-9]{3}-[0-9]{4}">
          </td>
        </tr>

        <!-- URL input with HTTPS pattern enforcement -->
        <tr>
          <td><label for="urlInput">URL:</label></td>
          <td><input type="url" id="urlInput" onchange="handleURL(this.value)"
pattern="https://.*"></td>
        </tr>

        <!-- Date picker -->
        <tr>
          <td><label for="dateInput">Date:</label></td>
          <td><input type="date" id="dateInput"
onchange="handleDate(this.value)"></td>
        </tr>

        <!-- Time picker -->
        <tr>
          <td><label for="timeInput">Time:</label></td>
          <td><input type="time" id="timeInput"
onchange="handleTime(this.value)"></td>
        </tr>

        <!-- Combined date and time picker -->
        <tr>
          <td><label for="datetimeInput">DateTime-Local:</label></td>
          <td><input type="datetime-local" id="datetimeInput"
onchange="handleDateTime(this.value)"></td>
        </tr>

        <!-- Month picker -->
        <tr>
          <td><label for="monthInput">Month:</label></td>
          <td><input type="month" id="monthInput"
onchange="handleMonth(this.value)"></td>
        </tr>

        <!-- Week picker -->
        <tr>
          <td><label for="weekInput">Week:</label></td>
          <td><input type="week" id="weekInput"
onchange="handleWeek(this.value)"></td>
        </tr>
```

```
        <!-- Range slider -->
        <tr>
          <td><label for="rangeInput">Range:</label></td>
          <td><input type="range" id="rangeInput" min="0" max="100"
oninput="handleRange(this.value)"></td>
        </tr>

        <!-- Color picker -->
        <tr>
          <td><label for="colorInput">Color:</label></td>
          <td>
            <input type="color" id="colorInput"
              onchange="handleColor(this.value)">
          </td>
        </tr>

        <!-- Checkbox -->
        <tr>
          <td><label for="checkboxInput">Checkbox:</label></td>
          <td><input type="checkbox" id="checkboxInput"
onchange="handleCheckbox(this.checked)"></td>
        </tr>

        <!-- Radio buttons (grouped by name attribute) -->
        <tr>
          <td><label for="radioA">Radio A:</label></td>
          <td><input type="radio" id="radioA" name="group1" value="A"
onchange="handleRadio(this.value)"></td>
        </tr>
        <tr>
          <td><label for="radioB">Radio B:</label></td>
          <td><input type="radio" id="radioB" name="group1" value="B"
onchange="handleRadio(this.value)"></td>
        </tr>

        <!-- File upload -->
        <tr>
          <td><label for="fileInput">File:</label></td>
          <td><input type="file" id="fileInput"
onchange="handleFile(this.files)"></td>
        </tr>

        <!-- Hidden field (not visible to users, still submitted) -->
        <tr style="display: none;">
          <td colspan="2">
            <input type="hidden" id="hiddenInput" value="classified">
          </td>
        </tr>

        <!-- Form control buttons -->
        <tr>
          <td colspan="2" style="text-align: center;">
            <input type="button" value="Button" onclick="handleButton()">
            <input type="reset" value="Reset" onclick="handleReset()">
            <input type="submit" value="Submit">
          </td>
        </tr>

      </table>
    </form>
  </div>

  <script>
    /*
      Core logging function.
      Appends messages to the log div and auto-scrolls.
```

```
    */
    function log(message) {

      // Clear log when the form is reset
      if (message === "Form reset") {
        document.getElementById("log").innerHTML = "";
      }

      const logDiv = document.getElementById("log");
      const entry = document.createElement("div");
      entry.textContent = message;
      logDiv.appendChild(entry);

      // Keep newest log entry visible
      logDiv.scrollTop = logDiv.scrollHeight;
    }

    // Individual handlers for each input type
    function handleText(value) { log("Text: " + value); }
    function handlePassword() { log("Password changed (value hidden, on
purpose)"); }
    function handleEmail(value) { log("Email: " + value); }
    function handleNumber(value) { log("Number: " + value); }
    function handleSearch(value) { log("Search: " + value); }
    function handleTel(value) { log("Tel: " + value); }
    function handleURL(value) { log("URL: " + value); }
    function handleDate(value) { log("Date: " + value); }
    function handleTime(value) { log("Time: " + value); }
    function handleDateTime(value) { log("DateTime: " + value); }
    function handleMonth(value) { log("Month: " + value); }
    function handleWeek(value) { log("Week: " + value); }
    function handleRange(value) { log("Range: " + value); }
    function handleColor(value) { log("Color: " + value); }
    function handleCheckbox(checked) { log("Checkbox checked: " + checked); }
    function handleRadio(value) { log("Radio selected: " + value); }

    // File input handler (optional chaining avoids errors)
    function handleFile(files) {
      log("File selected: " + (files[0]?.name || "none"));
    }

    function handleButton() { log("Button clicked"); }
    function handleReset() { log("Form reset"); }

    /*
      Form submission handler.
      Prevents actual submission and logs hidden input data.
    */
    document.getElementById("demoForm").onsubmit = function (e) {
      e.preventDefault();
      log("Form submitted");
      log("Hidden value: " + document.getElementById("hiddenInput").value);
    };
  </script>

</body>
```

7.2 Using Listeners (addEventListener)

In the **Listeners** version, the HTML stays clean and JavaScript does the wiring with `addEventListener`. This is the "grown-up" baseline: structure in HTML, behavior in JS. It also avoids common inline pitfalls like mixing quoting rules, creating hidden dependencies, and accidentally triggering code before the DOM is ready.

Best use cases: most projects, especially when you want maintainable code and clear separation of concerns. Great for small-to-medium forms where each input has unique logic.

Worst use cases: very large forms with many similar elements, where attaching a separate listener to every element becomes repetitive and noisy (and sometimes slightly wasteful).

```
<!DOCTYPE html>
<!--
  Demonstration of HTML <input> types using JavaScript event listeners.
  All events are attached in JavaScript to reinforce separation of concerns.

  Below is the same behavior, same UI, same logging—but
  all inline event handlers are removed and replaced with addEventListener.
The comments now explicitly call out why this approach matters.
-->

<head>
  <title>Input types</title>

  <style>
    /* Visual container for the demo */
    .div1 {
      width: 75%;
      background-color: cornsilk;
      padding: 25px;
      margin: 50px auto;
    }
  </style>
</head>

<body>

  <div class="div1">
    <h1>HTML Input Types — DOM Logging</h1>
    <h2>Adding Event Listeners</h2>
    <p>Every interaction is logged below.</p>

    <!-- Log output area -->
    <div id="log" style="border:1px solid #444;
      padding:10px;
      height:200px;
      overflow-y:auto;
      margin-bottom:20px;
      background:#f9f9f9;">
    </div>

    <!--
      Form contains examples of many HTML input types.
      No inline JavaScript is used here.
    -->
    <form id="demoForm">
      <table>
        <tr><td>Text:</td><td><input type="text" id="textInput"></td></tr>
        <tr><td>Password:</td><td><input type="password" id="passwordInput"></
td></tr>
        <tr><td>Email:</td><td><input type="email" id="emailInput"></td></tr>
        <tr><td>Number:</td><td><input type="number" id="numberInput"></td></
tr>
        <tr><td>Search:</td><td><input type="search" id="searchInput"></td></
tr>
        <tr>
          <td>Tel:</td>
          <td>
            <input type="tel" id="telInput"
              placeholder="555-555-5555"
              pattern="[0-9]{3}-[0-9]{3}-[0-9]{4}">
          </td>
        </tr>
        <tr><td>URL:</td><td><input type="url" id="urlInput"
```

```
pattern="https://.*"></td></tr>
        <tr><td>Date:</td><td><input type="date" id="dateInput"></td></tr>
        <tr><td>Time:</td><td><input type="time" id="timeInput"></td></tr>
        <tr><td>DateTime:</td><td><input type="datetime-local"
id="datetimeInput"></td></tr>
        <tr><td>Month:</td><td><input type="month" id="monthInput"></td></tr>
        <tr><td>Week:</td><td><input type="week" id="weekInput"></td></tr>
        <tr><td>Range:</td><td><input type="range" id="rangeInput" min="0"
max="100"></td></tr>
        <tr><td>Color:</td><td><input type="color" id="colorInput"></td></tr>
        <tr><td>Checkbox:</td><td><input type="checkbox" id="checkboxInput"></
td></tr>
        <tr><td>Radio A:</td><td><input type="radio" name="group1" value="A"
id="radioA"></td></tr>
        <tr><td>Radio B:</td><td><input type="radio" name="group1" value="B"
id="radioB"></td></tr>
        <tr><td>File:</td><td><input type="file" id="fileInput"></td></tr>

        <!-- Hidden input demonstrates non-visible form data -->
        <tr style="display:none;">
          <td colspan="2"><input type="hidden" id="hiddenInput"
value="classified"></td>
        </tr>

        <tr>
          <td colspan="2" style="text-align:center;">
            <input type="button" id="buttonInput" value="Button">
            <input type="reset" value="Reset">
            <input type="submit" value="Submit">
          </td>
        </tr>
      </table>
    </form>
  </div>

  <script>
    /*
      Utility function to append messages to the log div.
      Keeps the UI responsive and readable.
    */
    function log(message) {
      if (message === "Form reset") {
        document.getElementById("log").innerHTML = "";
      }

      const logDiv = document.getElementById("log");
      const entry = document.createElement("div");
      entry.textContent = message;
      logDiv.appendChild(entry);
      logDiv.scrollTop = logDiv.scrollHeight;
    }

    /*
      Event listeners are attached AFTER the DOM loads.
      This avoids race conditions and inline JavaScript.
    */
    document.addEventListener("DOMContentLoaded", () => {

      document.getElementById("textInput")
        .addEventListener("input", e => log("Text: " + e.target.value));

      document.getElementById("passwordInput")
        .addEventListener("change", () => log("Password changed (value hidden,
on purpose)"));

      document.getElementById("emailInput")
```

```
        .addEventListener("change", e => log("Email: " + e.target.value));

      document.getElementById("numberInput")
        .addEventListener("change", e => log("Number: " + e.target.value));

      document.getElementById("searchInput")
        .addEventListener("input", e => log("Search: " + e.target.value));

      document.getElementById("telInput")
        .addEventListener("change", e => log("Tel: " + e.target.value));

      document.getElementById("urlInput")
        .addEventListener("change", e => log("URL: " + e.target.value));

      document.getElementById("dateInput")
        .addEventListener("change", e => log("Date: " + e.target.value));

      document.getElementById("timeInput")
        .addEventListener("change", e => log("Time: " + e.target.value));

      document.getElementById("datetimeInput")
        .addEventListener("change", e => log("DateTime: " + e.target.value));

      document.getElementById("monthInput")
        .addEventListener("change", e => log("Month: " + e.target.value));

      document.getElementById("weekInput")
        .addEventListener("change", e => log("Week: " + e.target.value));

      document.getElementById("rangeInput")
        .addEventListener("input", e => log("Range: " + e.target.value));

      document.getElementById("colorInput")
        .addEventListener("change", e => log("Color: " + e.target.value));

      document.getElementById("checkboxInput")
        .addEventListener("change", e => log("Checkbox checked: " +
e.target.checked));

      document.querySelectorAll("input[name='group1']")
        .forEach(radio =>
          radio.addEventListener("change", e => log("Radio selected: " +
e.target.value))
        );

      document.getElementById("fileInput")
        .addEventListener("change", e =>
          log("File selected: " + (e.target.files[0]?.name || "none"))
        );

      document.getElementById("buttonInput")
        .addEventListener("click", () => log("Button clicked"));

      document.getElementById("demoForm")
        .addEventListener("reset", () => log("Form reset"));

      document.getElementById("demoForm")
        .addEventListener("submit", e => {
          e.preventDefault();
          log("Form submitted");
          log("Hidden value: " +
document.getElementById("hiddenInput").value);
        });
    });
  </script>
</body>
```

7.3 Using Delegation (event delegation)

In the **Delegation** version, you attach *one* listener (or a few) to the parent container (here, the `<form>`). Events bubble up from the input to the form, and your code inspects `event.target` to figure out which input triggered it. This is the "one guard at the door checks everyone's ID" approach.

Best use cases: large dynamic UIs, long forms, tables of inputs, and interfaces where elements are added/removed after page load. It's also excellent when many inputs share similar handling. **Worst use cases:** complex forms where different inputs need radically different event timing or custom rules. Delegation can become a logic maze if you pile too many special cases into one handler.

```
<!DOCTYPE html>
<!--
  HTML input types demo using EVENT DELEGATION.
  A single event listener handles all inputs by inspecting the event target.

  What follows is event delegation: one listener on the form,
  the DOM bubbling events upward, and JavaScript calmly deciding what
happened.
  Fewer listeners, less repetition, clearer intent.
  The browser already did half the work.
-->

<head>
  <title>Input types</title>

  <style>
    .div1 {
      width: 75%;
      background-color: cornsilk;
      padding: 25px;
      margin: 50px auto;
    }
  </style>
</head>

<body>

  <div class="div1">
    <h1>HTML Input Types — Event Delegation</h1>
    <p>All input events are handled by one listener.</p>

    <!-- Log output -->
    <div id="log" style="border:1px solid #444;
      padding:10px;
      height:200px;
      overflow-y:auto;
      margin-bottom:20px;
      background:#f9f9f9;">
    </div>

    <!-- Form with many input types -->
    <form id="demoForm">
      <table>
        <tr><td>Text:</td><td><input type="text" id="textInput"></td></tr>
        <tr><td>Password:</td><td><input type="password" id="passwordInput"></
td></tr>
        <tr><td>Email:</td><td><input type="email" id="emailInput"></td></tr>
        <tr><td>Number:</td><td><input type="number" id="numberInput"></td></
tr>
        <tr><td>Search:</td><td><input type="search" id="searchInput"></td></
tr>
        <tr><td>Tel:</td><td><input type="tel" id="telInput"></td></tr>
```

```
        <tr><td>URL:</td><td><input type="url" id="urlInput"></td></tr>
        <tr><td>Date:</td><td><input type="date" id="dateInput"></td></tr>
        <tr><td>Time:</td><td><input type="time" id="timeInput"></td></tr>
        <tr><td>DateTime:</td><td><input type="datetime-local"
id="datetimeInput"></td></tr>
        <tr><td>Month:</td><td><input type="month" id="monthInput"></td></tr>
        <tr><td>Week:</td><td><input type="week" id="weekInput"></td></tr>
        <tr><td>Range:</td><td><input type="range" id="rangeInput" min="0"
max="100"></td></tr>
        <tr><td>Color:</td><td><input type="color" id="colorInput"></td></tr>
        <tr><td>Checkbox:</td><td><input type="checkbox" id="checkboxInput"></
td></tr>
        <tr>
          <td>Radio A:</td>
          <td><input type="radio" name="group1" value="A"></td>
        </tr>
        <tr>
          <td>Radio B:</td>
          <td><input type="radio" name="group1" value="B"></td>
        </tr>
        <tr><td>File:</td><td><input type="file" id="fileInput"></td></tr>

        <!-- Hidden input -->
        <tr style="display:none;">
          <td colspan="2"><input type="hidden" id="hiddenInput"
value="classified"></td>
        </tr>

        <tr>
          <td colspan="2" style="text-align:center;">
            <input type="button" id="buttonInput" value="Button">
            <input type="reset" value="Reset">
            <input type="submit" value="Submit">
          </td>
        </tr>
      </table>
    </form>
  </div>

  <script>
    /*
      Writes a message to the log div and auto-scrolls.
    */
    function log(message) {
      if (message === "Form reset") {
        document.getElementById("log").innerHTML = "";
      }

      const logDiv = document.getElementById("log");
      const entry = document.createElement("div");
      entry.textContent = message;
      logDiv.appendChild(entry);
      logDiv.scrollTop = logDiv.scrollHeight;
    }

    /*
      One listener. One decision point.
      The event bubbles up from the input to the form.
    */
    document.addEventListener("DOMContentLoaded", () => {

      const form = document.getElementById("demoForm");

      form.addEventListener("input", handleEvent);
      form.addEventListener("change", handleEvent);
```

```
  form.addEventListener("reset", () => log("Form reset"));

  form.addEventListener("submit", e => {
    e.preventDefault();
    log("Form submitted");
    log("Hidden value: " + document.getElementById("hiddenInput").value);
  });

  document.getElementById("buttonInput")
    .addEventListener("click", () => log("Button clicked"));
});

/*
  Centralized event handler.
  Determines behavior based on input type and attributes.
*/
function handleEvent(e) {

  const el = e.target;

  if (el.tagName !== "INPUT") return;

  switch (el.type) {

    case "text":
      log("Text: " + el.value);
      break;

    case "password":
      log("Password changed (value hidden, on purpose)");
      break;

    case "email":
      log("Email: " + el.value);
      break;

    case "number":
      log("Number: " + el.value);
      break;

    case "search":
      log("Search: " + el.value);
      break;

    case "tel":
      log("Tel: " + el.value);
      break;

    case "url":
      log("URL: " + el.value);
      break;

    case "date":
      log("Date: " + el.value);
      break;

    case "time":
      log("Time: " + el.value);
      break;

    case "datetime-local":
      log("DateTime: " + el.value);
      break;

    case "month":
      log("Month: " + el.value);
```

```
          break;

        case "week":
          log("Week: " + el.value);
          break;

        case "range":
          log("Range: " + el.value);
          break;

        case "color":
          log("Color: " + el.value);
          break;

        case "checkbox":
          log("Checkbox checked: " + el.checked);
          break;

        case "radio":
          log("Radio selected: " + el.value);
          break;

        case "file":
          log("File selected: " + (el.files[0]?.name || "none"));
          break;
      }
    }
  </script>
</body>
```

7.4 Using Objects (data-driven handlers)

In the **Objects** version, you keep delegation, but replace the long `switch` statement with an **object lookup table** that maps an input's type (like `"email"` or `"date"`) to the appropriate handler function. This is the cleanest scaling strategy: adding a new input often becomes "add one row to the handler table."

Best use cases: scalable systems, maintainable demos, codebases where you want to add new behavior without editing a monster conditional. This is also a stepping stone toward more advanced patterns (config-driven forms, reusable components, validation maps).
Worst use cases: when the type alone isn't enough to determine behavior. If two text inputs need different logic, mapping only by `type` can be too blunt—then you'll want to dispatch by `id`, `name`, or `data-*` attributes instead.

```
<!DOCTYPE html>
<!--
  HTML input types demo using:
    - Event delegation (one listener for many inputs)
    - Data-driven handlers (mapping input types to functions)

  same interface, same behavior,
  but now the JavaScript is data-driven.
  Instead of a long switch, we have a lookup table (an object)
  that maps input types to handler functions.
-->

<head>
  <title>Input types</title>

  <style>
    .div1 {
      width: 75%;
      background-color: cornsilk;
      padding: 25px;
      margin: 50px auto;
```

```
    }
  </style>
</head>

<body>

  <div class="div1">
    <h1>HTML Input Types – Data-Driven Event Handling</h1>
    <p>Input types are mapped to handler functions through a lookup table.</p>

    <!-- Log output -->
    <div id="log" style="border:1px solid #444;
      padding:10px;
      height:200px;
      overflow-y:auto;
      margin-bottom:20px;
      background:#f9f9f9;">
    </div>

    <!-- Form with many input types -->
    <form id="demoForm">
      <table>
        <tr><td>Text:</td><td><input type="text" id="textInput"></td></tr>
        <tr><td>Password:</td><td><input type="password" id="passwordInput"></
td></tr>
        <tr><td>Email:</td><td><input type="email" id="emailInput"></td></tr>
        <tr><td>Number:</td><td><input type="number" id="numberInput"></td></
tr>
        <tr><td>Search:</td><td><input type="search" id="searchInput"></td></
tr>
        <tr><td>Tel:</td><td><input type="tel" id="telInput"></td></tr>
        <tr><td>URL:</td><td><input type="url" id="urlInput"></td></tr>
        <tr><td>Date:</td><td><input type="date" id="dateInput"></td></tr>
        <tr><td>Time:</td><td><input type="time" id="timeInput"></td></tr>
        <tr><td>DateTime:</td><td><input type="datetime-local"
id="datetimeInput"></td></tr>
        <tr><td>Month:</td><td><input type="month" id="monthInput"></td></tr>
        <tr><td>Week:</td><td><input type="week" id="weekInput"></td></tr>
        <tr><td>Range:</td><td><input type="range" id="rangeInput" min="0"
max="100"></td></tr>
        <tr><td>Color:</td><td><input type="color" id="colorInput"></td></tr>
        <tr><td>Checkbox:</td><td><input type="checkbox" id="checkboxInput"></
td></tr>
        <tr>
          <td>Radio A:</td>
          <td><input type="radio" name="group1" value="A"></td>
        </tr>
        <tr>
          <td>Radio B:</td>
          <td><input type="radio" name="group1" value="B"></td>
        </tr>
        <tr><td>File:</td><td><input type="file" id="fileInput"></td></tr>

        <!-- Hidden input -->
        <tr style="display:none;">
          <td colspan="2"><input type="hidden" id="hiddenInput"
value="classified"></td>
        </tr>

        <tr>
          <td colspan="2" style="text-align:center;">
            <input type="button" id="buttonInput" value="Button">
            <input type="reset" value="Reset">
            <input type="submit" value="Submit">
          </td>
        </tr>
```

```
    </table>
  </form>
</div>

<script>
  /*
    Writes a message to the log div and auto-scrolls.
  */
  function log(message) {
    if (message === "Form reset") {
      document.getElementById("log").innerHTML = "";
    }

    const logDiv = document.getElementById("log");
    const entry = document.createElement("div");
    entry.textContent = message;
    logDiv.appendChild(entry);
    logDiv.scrollTop = logDiv.scrollHeight;
  }

  /*
    Handler table:
    Keys are input "type" values.
    Values are functions that know how to log that type.
    This replaces a long switch statement.
  */
  const inputHandlers = {
    text: el => log("Text: " + el.value),
    password: el => log("Password changed (value hidden, on purpose)"),
    email: el => log("Email: " + el.value),
    number: el => log("Number: " + el.value),
    search: el => log("Search: " + el.value),
    tel: el => log("Tel: " + el.value),
    url: el => log("URL: " + el.value),
    date: el => log("Date: " + el.value),
    time: el => log("Time: " + el.value),
    "datetime-local": el => log("DateTime: " + el.value),
    month: el => log("Month: " + el.value),
    week: el => log("Week: " + el.value),
    range: el => log("Range: " + el.value),
    color: el => log("Color: " + el.value),
    checkbox: el => log("Checkbox checked: " + el.checked),
    radio: el => log("Radio selected: " + el.value),
    file: el => log("File selected: " + (el.files[0]?.name || "none"))
  };

  /*
    Attach a few top-level listeners when the DOM is ready:
      - Delegated 'input' and 'change' on the form
      - 'reset' and 'submit' on the form
      - Separate click handler for the plain button
  */
  document.addEventListener("DOMContentLoaded", () => {

    const form = document.getElementById("demoForm");

    // Delegated listeners: any input inside the form bubbles up here
    form.addEventListener("input", handleDelegatedEvent);
    form.addEventListener("change", handleDelegatedEvent);

    form.addEventListener("reset", () => log("Form reset"));

    form.addEventListener("submit", e => {
      e.preventDefault();
      log("Form submitted");
      log("Hidden value: " + document.getElementById("hiddenInput").value);
```

```
    });

    document.getElementById("buttonInput")
      .addEventListener("click", () => log("Button clicked"));
  });

  /*
    Central delegated handler:
    - Makes sure the event came from an <input>
    - Looks up a handler by input type
    - Calls that handler if it exists
  */
  function handleDelegatedEvent(e) {
    const el = e.target;

    if (el.tagName !== "INPUT") return;

    const handler = inputHandlers[el.type];

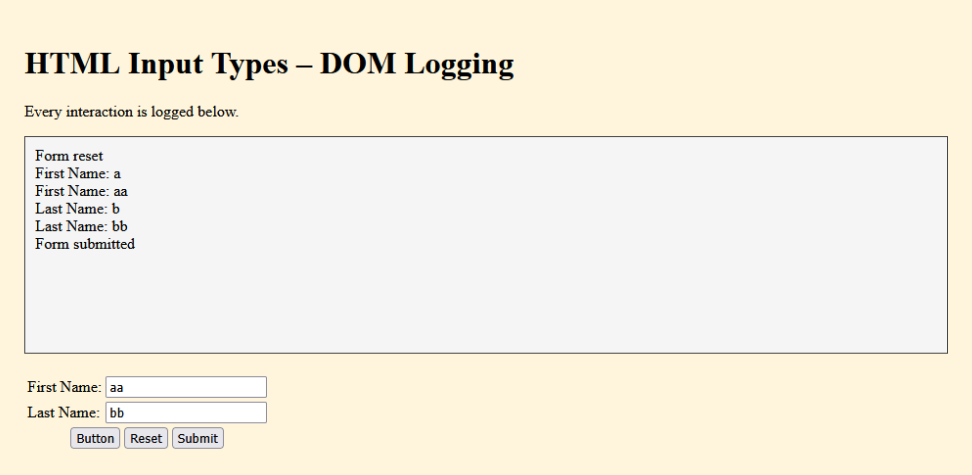
    if (handler) {
      handler(el, e); // pass element (and event if needed later)
    }
  }
</script>
</body>
```

7.5 Practical Summary: When Each Approach Wins (and Loses)

Inline events are a bicycle with no brakes: perfect for learning balance, a bad idea on a downhill commute. Event listeners are the standard, clean and professional. Delegation is the power tool you reach for when the number of inputs explodes or the DOM changes dynamically. Objects turn delegation into a system: fewer moving parts, clearer intent, and easier growth.

Inline shows *what events do* arrow.r listeners show *where code should live* arrow.r delegation shows *how to scale events* arrow.r objects show *how to scale logic.*

8. HTML Input Tag - Short Form



8.1 Using Inline (inline event handlers)

In the **Inline** version, behavior is attached directly inside the HTML using attributes like `oninput="..."` and `onchange="..."`. That means the HTML doesn't just describe structure—it also tells the browser *what code to run* when something happens. It's immediate and readable for beginners: you can point to a single line and say "this fires the function."

Best use cases: tiny demos, learning exercises, quick one-off prototypes where clarity beats architecture. **Worst use cases:** real applications, shared codebases, or anything you expect to maintain. Inline handlers get messy fast, duplicate logic, and blur separation of concerns. It's also harder to refactor because your behavior is scattered across markup.

```
<!DOCTYPE html>
<!--
  This document demonstrates using INLINE event handlers.
  Each input triggers a JavaScript function that logs
  user interaction directly to the DOM.
-->

<head>
  <!-- Page title shown in the browser tab -->
  <title>Input types</title>

  <style>
    /*
      Simple container styling to center the demo
      and make it visually distinct from the page background
    */
    .div1 {
      width: 75%;
      background-color: cornsilk;
      padding: 25px;
      margin-left: auto;
      margin-right: auto;
      margin-top: 50px;
      margin-bottom: 50px;
    }
  </style>
</head>

<body>

  <!-- Main demo container -->
  <div class="div1">

    <!-- Page heading -->
```

```
    <h1>HTML Input Types – DOM Logging</h1>
    <p>Every interaction is logged below.</p>

    <!--
      Log area:
      JavaScript dynamically inserts messages here
      instead of using console.log()
    -->
    <div id="log" style="border:1px solid #444;
      padding:10px;
      height:200px;
      overflow-y:auto;
      margin-bottom: 20px;
      background:#f9f9f9;">
    </div>

    <!--
      Demo form:
      Inline handlers call JavaScript functions on input.
    -->
    <form id="demoForm">
      <table>

        <!-- First Name input (fires on every keystroke) -->
        <tr>
          <td><label for="firstName">First Name:</label></td>
          <td><input type="text" id="firstName"
oninput="handleFirstName(this.value)"></td>
        </tr>

        <!-- Last Name input (fires on every keystroke) -->
        <tr>
          <td><label for="lastName">Last Name:</label></td>
          <td><input type="text" id="lastName"
oninput="handleLastName(this.value)"></td>
        </tr>

        <!-- Form control buttons (use <button> to keep the focus on input
fields) -->
        <tr>
          <td colspan="2" style="text-align: center;">
            <button type="button" onclick="handleButton()">Button</button>
            <button type="reset">Reset</button>
            <button type="submit">Submit</button>
          </td>
        </tr>

      </table>
    </form>
  </div>

  <script>
    /*
      Core logging function.
      Appends messages to the log div and auto-scrolls.
    */
    function log(message) {

      // Clear log when the form is reset
      if (message === "Form reset") {
        document.getElementById("log").innerHTML = "";
      }

      const logDiv = document.getElementById("log");
      const entry = document.createElement("div");
      entry.textContent = message;
```

```
      logDiv.appendChild(entry);

      // Keep newest log entry visible
      logDiv.scrollTop = logDiv.scrollHeight;
    }

    // Inline handlers for the two input fields
    function handleFirstName(value) { log("First Name: " + value); }
    function handleLastName(value) { log("Last Name: " + value); }

    function handleButton() { log("Button clicked"); }

    function handleReset() { log("Form reset"); }

    /*
      Form submission handler.
      Prevents actual submission and logs completion.
    */
    document.getElementById("demoForm").onsubmit = function (e) {
      e.preventDefault();
      log("Form submitted");
    };

    // Reset event (so the log clears when the reset button is used)
    document.getElementById("demoForm").onreset = function () {
      log("Form reset");
    };
  </script>

</body>
```

8.2 Using Listeners (addEventListener)

In the **Listeners** version, the HTML stays clean and JavaScript does the wiring with `addEventListener`. This is the "grown-up" baseline: structure in HTML, behavior in JS. It also avoids common inline pitfalls like mixing quoting rules, creating hidden dependencies, and accidentally triggering code before the DOM is ready.

Best use cases: most projects, especially when you want maintainable code and clear separation of concerns. Great for small-to-medium forms where each input has unique logic. **Worst use cases:** very large forms with many similar elements, where attaching a separate listener to every element becomes repetitive and noisy (and sometimes slightly wasteful).

```
<!DOCTYPE html>
<!--
  Demonstration of HTML <input> types using JavaScript event listeners.
  All events are attached in JavaScript to reinforce separation of concerns.

  Below is the same behavior, same UI, same logging—but
  all inline event handlers are removed and replaced with addEventListener.
The comments now explicitly call out why this approach matters.
-->

<head>
  <title>Input types</title>

  <style>
    /* Visual container for the demo */
    .div1 {
      width: 75%;
      background-color: cornsilk;
      padding: 25px;
      margin: 50px auto;
    }
  </style>
```

```
</head>

<body>

  <div class="div1">
    <h1>HTML Input Types – DOM Logging</h1>
    <h2>Adding Event Listeners</h2>
    <p>Every interaction is logged below.</p>

    <!-- Log output area -->
    <div id="log" style="border:1px solid #444;
      padding:10px;
      height:200px;
      overflow-y:auto;
      margin-bottom:20px;
      background:#f9f9f9;">
    </div>

    <!--
      Form contains examples of input fields.
      No inline JavaScript is used here.
    -->
    <form id="demoForm">
      <table>
        <tr>
          <td><label for="firstName">First Name:</label></td>
          <td><input type="text" id="firstName"></td>
        </tr>
        <tr>
          <td><label for="lastName">Last Name:</label></td>
          <td><input type="text" id="lastName"></td>
        </tr>

        <tr>
          <td colspan="2" style="text-align:center;">
            <button type="button" id="buttonInput">Button</button>
            <button type="reset">Reset</button>
            <button type="submit">Submit</button>
          </td>
        </tr>
      </table>
    </form>
  </div>

  <script>
    /*
      Utility function to append messages to the log div.
      Keeps the UI responsive and readable.
    */
    function log(message) {
      if (message === "Form reset") {
        document.getElementById("log").innerHTML = "";
      }

      const logDiv = document.getElementById("log");
      const entry = document.createElement("div");
      entry.textContent = message;
      logDiv.appendChild(entry);
      logDiv.scrollTop = logDiv.scrollHeight;
    }

    /*
      Event listeners are attached AFTER the DOM loads.
      This avoids race conditions and inline JavaScript.
    */
    document.addEventListener("DOMContentLoaded", () => {
```

```
      document.getElementById("firstName")
        .addEventListener("input", e => log("First Name: " + e.target.value));

      document.getElementById("lastName")
        .addEventListener("input", e => log("Last Name: " + e.target.value));

      document.getElementById("buttonInput")
        .addEventListener("click", () => log("Button clicked"));

      document.getElementById("demoForm")
        .addEventListener("reset", () => log("Form reset"));

      document.getElementById("demoForm")
        .addEventListener("submit", e => {
          e.preventDefault();
          log("Form submitted");
        });
    });
  </script>
</body>
```

8.3 Using Delegation (event delegation)

In the **Delegation** version, you attach *one* listener (or a few) to the parent container (here, the `<form>`). Events bubble up from the input to the form, and your code inspects `event.target` to figure out which input triggered it. This is the "one guard at the door checks everyone's ID" approach.

Best use cases: large dynamic UIs, long forms, tables of inputs, and interfaces where elements are added/removed after page load. It's also excellent when many inputs share similar handling. **Worst use cases:** complex forms where different inputs need radically different event timing or custom rules. Delegation can become a logic maze if you pile too many special cases into one handler.

```
<!DOCTYPE html>
<!--
  HTML input demo using EVENT DELEGATION.
  A single event listener handles all inputs by inspecting the event target.

  What follows is event delegation: one listener on the form,
  the DOM bubbling events upward, and JavaScript calmly deciding what
happened.
  Fewer listeners, less repetition, clearer intent.
  The browser already did half the work.
-->

<head>
  <title>Input types</title>

  <style>
    .div1 {
      width: 75%;
      background-color: cornsilk;
      padding: 25px;
      margin: 50px auto;
    }
  </style>
</head>

<body>

  <div class="div1">
    <h1>HTML Input Types – Event Delegation</h1>
    <p>All input events are handled by one listener.</p>
```

```
  <!-- Log output -->
  <div id="log" style="border:1px solid #444;
    padding:10px;
    height:200px;
    overflow-y:auto;
    margin-bottom:20px;
    background:#f9f9f9;">
  </div>

  <!-- Form with input fields -->
  <form id="demoForm">
    <table>
      <tr>
        <td><label for="firstName">First Name:</label></td>
        <td><input type="text" id="firstName"></td>
      </tr>
      <tr>
        <td><label for="lastName">Last Name:</label></td>
        <td><input type="text" id="lastName"></td>
      </tr>

      <tr>
        <td colspan="2" style="text-align:center;">
          <button type="button" id="buttonInput">Button</button>
          <button type="reset">Reset</button>
          <button type="submit">Submit</button>
        </td>
      </tr>
    </table>
  </form>
</div>

<script>
  /*
    Writes a message to the log div and auto-scrolls.
  */
  function log(message) {
    if (message === "Form reset") {
      document.getElementById("log").innerHTML = "";
    }

    const logDiv = document.getElementById("log");
    const entry = document.createElement("div");
    entry.textContent = message;
    logDiv.appendChild(entry);
    logDiv.scrollTop = logDiv.scrollHeight;
  }

  /*
    One listener. One decision point.
    The event bubbles up from the input to the form.
  */
  document.addEventListener("DOMContentLoaded", () => {

    const form = document.getElementById("demoForm");

    form.addEventListener("input", handleEvent);

    form.addEventListener("reset", () => log("Form reset"));

    form.addEventListener("submit", e => {
      e.preventDefault();
      log("Form submitted");
    });
```

```
      document.getElementById("buttonInput")
        .addEventListener("click", () => log("Button clicked"));
    });

    /*
      Centralized event handler.
      Since both fields are text inputs, we dispatch by id.
    */
    function handleEvent(e) {

      const el = e.target;

      if (el.tagName !== "INPUT") return;

      switch (el.id) {

        case "firstName":
          log("First Name: " + el.value);
          break;

        case "lastName":
          log("Last Name: " + el.value);
          break;
      }
    }
  </script>
</body>
```

8.4 Using Objects (data-driven handlers)

In the **Objects** version, you keep delegation, but replace the long `switch` statement with an **object lookup table** that maps an input's type (like `"email"` or `"date"`) to the appropriate handler function. This is the cleanest scaling strategy: adding a new input often becomes "add one row to the handler table."

Best use cases: scalable systems, maintainable demos, codebases where you want to add new behavior without editing a monster conditional. This is also a stepping stone toward more advanced patterns (config-driven forms, reusable components, validation maps).
Worst use cases: when the type alone isn't enough to determine behavior. If two text inputs need different logic, mapping only by `type` can be too blunt—then you'll want to dispatch by `id`, `name`, or `data-*` attributes instead.

```
<!DOCTYPE html>
<!--
  HTML input demo using:
    - Event delegation (one listener for many inputs)
    - Data-driven handlers (mapping inputs to functions)

  same interface, same behavior,
  but now the JavaScript is data-driven.
  Instead of a long switch, we have a lookup table (an object)
  that maps inputs to handler functions.
-->

<head>
  <title>Input types</title>

  <style>
    .div1 {
      width: 75%;
      background-color: cornsilk;
      padding: 25px;
      margin: 50px auto;
    }
  </style>
```

```
</head>

<body>

  <div class="div1">
    <h1>HTML Input Types – Data-Driven Event Handling</h1>
    <p>Input types are mapped to handler functions through a lookup table.</p>

    <!-- Log output -->
    <div id="log" style="border:1px solid #444;
      padding:10px;
      height:200px;
      overflow-y:auto;
      margin-bottom:20px;
      background:#f9f9f9;">
    </div>

    <!-- Form with input fields -->
    <form id="demoForm">
      <table>
        <tr>
          <td><label for="firstName">First Name:</label></td>
          <td><input type="text" id="firstName"></td>
        </tr>
        <tr>
          <td><label for="lastName">Last Name:</label></td>
          <td><input type="text" id="lastName"></td>
        </tr>

        <tr>
          <td colspan="2" style="text-align:center;">
            <button type="button" id="buttonInput">Button</button>
            <button type="reset">Reset</button>
            <button type="submit">Submit</button>
          </td>
        </tr>
      </table>
    </form>
  </div>

  <script>
    /*
      Writes a message to the log div and auto-scrolls.
    */
    function log(message) {
      if (message === "Form reset") {
        document.getElementById("log").innerHTML = "";
      }

      const logDiv = document.getElementById("log");
      const entry = document.createElement("div");
      entry.textContent = message;
      logDiv.appendChild(entry);
      logDiv.scrollTop = logDiv.scrollHeight;
    }

    /*
      Handler table:
      Keys are input identifiers.
      Values are functions that know how to log that input.
      This replaces a long conditional.
    */
    const inputHandlers = {
      firstName: el => log("First Name: " + el.value),
      lastName: el => log("Last Name: " + el.value)
    };
```

```
    /*
      Attach a few top-level listeners when the DOM is ready:
        - Delegated 'input' on the form
        - 'reset' and 'submit' on the form
        - Separate click handler for the plain button
    */
    document.addEventListener("DOMContentLoaded", () => {

      const form = document.getElementById("demoForm");

      // Delegated listener: any input inside the form bubbles up here
      form.addEventListener("input", handleDelegatedEvent);

      form.addEventListener("reset", () => log("Form reset"));

      form.addEventListener("submit", e => {
        e.preventDefault();
        log("Form submitted");
      });

      document.getElementById("buttonInput")
        .addEventListener("click", () => log("Button clicked"));
    });

    /*
      Central delegated handler:
      - Makes sure the event came from an <input>
      - Looks up a handler by input id
      - Calls that handler if it exists
    */
    function handleDelegatedEvent(e) {
      const el = e.target;

      if (el.tagName !== "INPUT") return;

      const handler = inputHandlers[el.id];

      if (handler) {
        handler(el, e); // pass element (and event if needed later)
      }
    }
  </script>
</body>
```

8.5 Practical Summary: When Each Approach Wins (and Loses)

Inline events are a bicycle with no brakes: perfect for learning balance, a bad idea on a downhill commute. Event listeners are the standard, clean and professional. Delegation is the power tool you reach for when the number of inputs explodes or the DOM changes dynamically. Objects turn delegation into a system: fewer moving parts, clearer intent, and easier growth.

Inline shows *what events do* arrow.r listeners show *where code should live* arrow.r delegation shows *how to scale events* arrow.r objects show *how to scale logic.*

9. If Statement

A **JavaScript `if` statement** is a fundamental construct that allows you to make decisions in your code based on specified conditions. It enables you to execute a block of code **only if a particular condition is true**. Here's how it works:

1. **Syntax**:
    - The basic structure of an `if` statement looks like this:

```
if (condition) {
   // Block of code to be executed if the condition is true
}
```

- The `condition` is an expression that evaluates to either `true` or `false`.

1. **Execution Flow**:
    - When your program encounters an `if` statement, it checks whether the specified condition is true.
    - If the condition is true, the code inside the curly braces `{ ... }` is executed.
    - If the condition is false, the code block is skipped, and the program moves on to the next statement.
2. **Example**: Let's say we want to greet users based on the time of day. Here's how we can use an `if` statement:

JavaScript

```
let hour = new Date().getHours();

if (hour < 10) {
    console.log("Good morning");
} else if (hour < 20) {
    console.log("Good day");
} else {
    console.log("Good evening");
}
```

- In this example:
    - If the current hour is less than 10, it prints "Good morning."
    - If the hour is between 10 and 20 (exclusive), it prints "Good day."
    - Otherwise, it prints "Good evening."

1. **Additional Notes**:
    - You can use `else` to specify an alternative block of code to execute when the condition is false.
    - The `else if` statement allows you to test multiple conditions sequentially.
    - Remember that JavaScript is case-sensitive, so use lowercase `if`, `else`, and `else if`.
    -

9.1 Lecture Code

```
/*
<!DOCTYPE html >
<!--  If Examples by Jim Goudy -- >
<html >
   <head>
   <title> If Example </title>
   <meta charset = "UTF-8" >
   <meta name = "viewport" content = "width=device-width, initial-scale=1.0" >

   <style >
   .zbox {
      float: left;
```

```
      width: 200 px;
      height: 200 px;
      border - width: 3 px;
      border - style: solid;
      margin: 5 px;
      text - align: center;
      font - size: 24 pt;
   }
    #warningColor {
       font - style: italic;
       color: red;
    }
  </style>

<script >
   function setBoxColor() {
     //this demostrates an else if statement
      var userInput = Number(document.getElementById("inputC").value);
      var message = "Please enter a number between 0 and 100";

     //reset boxes, error message and inputbox
      resetBoxes();

     //example of else if
      if (userInput < 0) {
         document.getElementById("warningColor").innerHTML = message;

      } else if (userInput > 0 && userInput <= 50) {
         document.getElementById("boxGreen").style.backgroundColor = "green";
      } else if (userInput >= 51 && userInput <= 65) {
         document.getElementById("boxGreen").style.backgroundColor = "green";
         document.getElementById("boxYellow").style.backgroundColor =
"yellow";
      } else if (userInput >= 66 && userInput <= 74) {
         document.getElementById("boxGreen").style.backgroundColor = "green";
         document.getElementById("boxYellow").style.backgroundColor =
"yellow";
         document.getElementById("boxOrange").style.backgroundColor =
"orange";
      } else if (userInput >= 75 && userInput <= 100) {
         document.getElementById("boxGreen").style.backgroundColor = "green";
         document.getElementById("boxYellow").style.backgroundColor =
"yellow";
         document.getElementById("boxOrange").style.backgroundColor =
"orange";
         document.getElementById("boxRed").style.backgroundColor = "red";

      } else {
         document.getElementById("warningColor").innerHTML = message;
      }
   }

    function resetBoxes() {
     //get all of the boxs and store them into an array call boxes
      var boxes = document.getElementsByClassName("zbox");

     //loop through each box and change the color back to white
     //note: you cannot change them all at once
      for (var cntr = 0; cntr < boxes.length; cntr++) {
         boxes[cntr].style.backgroundColor = 'white';
      }
     //reset inputbox
      document.getElementById("inputC").value = "";

     //Reset the warning message
      document.getElementById("warningColor").innerHTML = "";
```

```
    }

    function checkAge() {
      //Store Age To variable
       var nAge = document.getElementById("inputAge").value;

       if (nAge > 17) {
          var ageMess = "  You are old enough to vote";
          document.getElementById("ageMessage").innerHTML = ageMess;
       }

    }

    function clearAge() {
       document.getElementById("inputAge").value = null;
       document.getElementById("ageMessage").innerHTML = null;

    }
  </script>
</head>
<body>
    <h1> If Example </h1>
    <p> This is an if example </p>
    <p> Enter Your Age
    <input type = "text" id = "inputAge" onfocus = "clearAge()"/ >
    <input type = "button" onclick = "checkAge()" value = "Check Age"/ >
    <span id = "ageMessage" > </span> </p >
    <h1> Else If </h1>
    <p >< br/ > Enter Value
  for boxes:
  <input type = "text" id = "inputC" size = "25"
  placeholder = "Enter a value between 0 and 100"/ >
  <input type = "button" value = "Enter" onclick = "setBoxColor()"/ >
  <input type = "button" value = "Reset" onclick = "resetBoxes()"/ >
  <span id = "warningColor" > </span> </p >
  <div id = "boxGreen" class = "zbox" > 0 - 50 </div>
  <div id = "boxYellow" class = "zbox" > 51 - 65 </div>
  <div id = "boxOrange" class = "zbox" > 66 - 74 </div>
  <div id = "boxRed" class = "zbox" > 75 - 100 </div>
  <p ></p>
  </body>
</html>
*/
```

End Of If Statement

10. Loops and Switch

1. For Loop:
    - Ideal for known iteration counts.

```
for (initialization; condition; afterthought) {
    // Code to execute
}
```

- Example:

```
for (let i = 0; i < 5; i++) {
    console.log("Walking east one step");
}
```

1. While Loop:
    - Best for unknown iteration counts based on a condition.

```
while (condition) {
    // Code to execute
}
```

1. Do...While Loop:
    - Executes at least once, even if the condition is initially false.

```
do {
    // Code to execute
} while (condition);
```

1. For...In Loop:
    - Perfect for iterating over object properties.

```
for (const key in object) {
    // Code to execute
}
```

10.1 Switch Statement

switch statement is a powerful construct used for making decisions in code based on different conditions. It provides an organized and concise alternative to using multiple `if-else` statements. Here's how it works:

1. **Syntax**:

```
switch (expression) {
    case value1:
        // Code block executed if expression matches value1
        break;
    case value2:
        // Code block executed if expression matches value2
        break;
    // More cases...
    default:
        // Code block executed if no case matches the expression
}
```

1. **How It Works**:
    - The `switch` expression is evaluated once.
    - The value of the expression is compared with the values of each `case`.
    - If there's a match, the associated block of code is executed.
    - If no match is found, the `default` code block is executed.

2. **Example**: Let's say we want to determine the day of the week based on the current day number (0 for Sunday, 1 for Monday, etc.). We can use the `switch` statement like this:

JavaScript

```
switch (new Date().getDay()) {
    case 0:
        day = "Sunday";
        break;
    case 1:
        day = "Monday";
        break;
    // ... other cases ...
    default:
        day = "Unknown day";
}
```

AI-generated code. Review and use carefully. More info on FAQ.

1. **Break Keyword**:
    - When JavaScript encounters a `break` keyword inside a `case`, it exits the `switch` block.
    - It's not necessary to break the last case; the block ends there anyway.
2. **Default Keyword**:
    - The `default` keyword specifies the code to run if no case matches.
    - It's like a fallback option.
3. **Common Code Blocks**:
    - Sometimes different `case` values share the same code block.
    - For example:

JavaScript

```
switch (new Date().getDay()) {
    case 4:
    case 5:
        text = "Soon it is Weekend";
        break;
    case 0:
    case 6:
        text = "It is Weekend";
        break;
    default:
        text = "Looking forward to the Weekend";
}
```

10.1.1 Lecture Code

```
<!DOCTYPE html>
<!--
Loops, if's and switch examples
-->
<html>
    <head>
        <title>TODO supply a title</title>
        <script>

            function loops()
            {
                //do loop
                let dostring = "do1 ";
                let cntr = 2;
                do {
                    dostring = dostring + "do" + cntr + " ";
```

```
        cntr++;
    } while (cntr <= 10);

    //while loop
    let whilestring = "while1 "
    cntr = 1;

    while (cntr <= 10)
    {
        whilestring = whilestring + "while" + cntr + " ";
        cntr++;
    }

    //for loop
    let forstring = "";
    for (let i = 1; i <= 10; i++)
    {
        forstring = forstring + "for" + i + " ";

    }
    document.getElementById("p1").innerHTML =
            dostring + "<br><br>" +
            whilestring + "<br><br>" +
            forstring + "<br><br>";
}

function rbutton()
{
    //checking radio buttons with an if
    let rbttns = document.getElementsByName("rbgroup1");
    for (let i = 0; i < rbttns.length; i++)
    {
        if (rbttns[i].checked)
        {
            document.getElementById("p2").innerHTML =
                    rbttns[i].value + " was selected <br>";
            document.getElementById("p2").style.color =
            rbttns[i].value;
        }
    }
}

function numrange()
{
    //example of an if
    let x;

    x = document.getElementById("nbox1").value;


    if (x <= 25)
    {
        document.getElementById("p3").innerHTML =
                "Your number is less than or equal to 25";
    }
    else if (x <= 50)
    {
        document.getElementById("p3").innerHTML =
                "Your number is less than or equal to 50";
    }
    else if (x <= 75)
    {
        document.getElementById("p3").innerHTML =
                "Your number is less than or equal to 75";
    }
    else if (x <= 100)
```

```
            {
                document.getElementById("p3").innerHTML =
                        "Your number is less than or equal to 100";
            }
            else
            {
                document.getElementById("p3").innerHTML =
                        "Your number is greater than 100";
            }

            //example of a range
            if ((x > 0 && x < 25) || (x > 75 && x < 100))
            {
                document.getElementById("p4").innerHTML =
                        "Your number must be near \n\
                        one of the ends of the range";
            }
            else
            {
                document.getElementById("p4").innerHTML = "";
            }

        }

        function switchexample()
        {
            //Example of a switch
            let x = document.getElementById('nbox2').value;

            switch (x)
            {
                case "1":
                    document.getElementById("p5").innerHTML =
                    "You chose 1 - you win a new car";
                    break;
                case "2":
                   document.getElementById("p5").innerHTML =
                    "You chose 2 - you win a new boat";
                    break;
                case "3":
                    document.getElementById("p5").innerHTML =
                    "You chose 3 - you win a vacation to Italy";
                    break;
                default:
                    document.getElementById("p5").innerHTML =
                    "You did not choose correctly - are destined to\n\
                    roam the world aimlessly";
                    break;
            }
        }

    </script>

</head>
<body>

    <br>
    <hr>
    <H2>Loops</H2>
    <input type="submit" value="Run" name="btnRun" onclick="loops()" />
    <p  id="p1"></p>
    <br>
    <hr>

    <h2>If example with radio buttons</h2>
    <input type="radio" name="rbgroup1" value="red" checked="checked"
```

```
                onclick="rbutton()" />
            Red<br>
            <input type="radio" name="rbgroup1" value="brown"
                onclick="rbutton()"/>
            Brown<br>
            <input type="radio" name="rbgroup1" value="blue"
                onclick="rbutton()"/>
            Blue<br>

            <p id="p2"></p>

            <hr>

            <h2>If else example</h2>
            Enter a number between 1- 100
            <input type="text" name="numberbox" value="" size="15" id="nbox1" />
            <input type="submit" value="Number Enter" name="bttnnumber"
                onclick="numrange()"/>
            <p id="p3"></p>
            <p id = "p4"></p>

            <hr>
            <h2>Switch statement</h2>
            Enter 1, 2 or 3<br>
            <input type="text" name="nbox2" value="" size="15" id="nbox2" />
            <input type="submit" value="Number Enter" name="bttnnumber2"
                onclick="switchexample()"/>
            <p id="p5"></p>

        </body>
    </html>
```

11. Radio and Check Boxes

Radio buttons and checkboxes are two of the most common ways a web page collects "choice" input, but they behave very differently: radios enforce a single decision inside a shared group (only one option can be selected at a time), while checkboxes allow multiple independent selections (zero, one, or many). This example demonstrates both patterns in one clean page: the radio section listens for `change` events and uses `document.querySelector('input[name="grp1"]:checked')` to grab the one currently-selected radio and immediately print a message, while the checkbox section waits for a button click, uses `document.querySelectorAll('input[type="checkbox"]:checked')` to gather every checked box, and then builds a readable summary by matching each checkbox to its `<label>` via `label[for="..."]`. In other words, the DOM becomes a searchable map: `querySelector` is your "find the first match" tool for single answers (the selected radio), and `querySelectorAll` is your "collect all matches" tool for plural answers (the checked checkboxes).

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>Radio + Checkbox Demo</title>

  <style>
    .div1 {
            width: 75%;
            background-color: cornsilk;
            padding: 20px;
            margin-left: auto;
            margin-right: auto;
            margin-top: 50px;
            margin-bottom: 50px;
    }
  </style>
</head>

<body>
  <div class="div1">
    <h1>Radio Buttons</h1>

    <input type="radio" id="rad1" name="grp1" value="10" />
    <label for="rad1">Option 1</label><br />

    <input type="radio" id="rad2" name="grp1" value="20" checked />
    <label for="rad2">Option 2</label><br />

    <input type="radio" id="rad3" name="grp1" value="30" />
    <label for="rad3">Option 3</label><br />

    <br />
    <hr />
    <br />

    <h1>Check Boxes</h1>

    <input type="checkbox" id="cb1" name="cbgrp" value="11" />
    <label for="cb1">check box 1</label><br />

    <input type="checkbox" id="cb2" name="cbgrp" value="22" />
    <label for="cb2">check box 2</label><br />

    <input type="checkbox" id="cb3" name="cbgrp" value="33" />
    <label for="cb3">check box 3</label><br /><br />
```

```
    <input type="button" value="Show selected checkboxes"
onclick="checkboxCheck()" />

    <br /><br />
    <hr />
    <br />

    <p id="radioMess"></p>
    <p id="checkMess"></p>
  </div>

  <script>

    function updateRadioMessage() {

      // Select the first thing that is named "grp1" that is checked
      const selected = document.querySelector('input[name="grp1"]:checked');
      const radioOut = document.getElementById("radioMess");

      if (!selected) {
        radioOut.textContent = "No radio option selected.";
        return;
      }

      let message;
      if (selected.value === "10") message = "rad1 selected (value 10)";
      else if (selected.value === "20") message = "rad2 selected (value 20)";
      else message = "rad3 selected (value 30)";

      radioOut.textContent = message;
    }

    function checkboxCheck() {

      // all of the selected checkboxes are collected and put into an array
checkedBoxes
      const checkedBoxes =
document.querySelectorAll('input[type="checkbox"]:checked');
      console.log(checkedBoxes);

      const checkOut = document.getElementById("checkMess");

      // nothing has been checked
      if (checkedBoxes.length === 0) {
        checkOut.textContent = "No checkboxes selected.";
        return;
      }

      let message = "Checked: ";

      for (let i = 0; i < checkedBoxes.length; i++) {
        const cb = checkedBoxes[i];

        const label = document.querySelector('label[for="' + cb.id + '"]');

        let labelText;
        if (label) {
          labelText = label.textContent.trim();
        } else {
          labelText = cb.id;
        }

        if (i > 0) {
          message += ", ";
        }
```

```
        message += labelText + " (value " + cb.value + ")";
      }

      checkOut.textContent = message;
    }

    // add listeners for radio buttons
    document.querySelectorAll('input[name="grp1"]').forEach(radio => {
      radio.addEventListener("change", updateRadioMessage);
    });

    // add listeners for check boxes
    document.querySelectorAll('input[name="cbgrp"]').forEach(checkbox=>{
            checkbox.addEventListener("change",checboxCheck);
    });

    updateRadioMessage();
  </script>
</body>
</html>
```

11.1 Document.querySelector

`document.querySelector(...)` is the browser's "find me the first matching thing" function.

It takes a **CSS selector string** (the same kind of selector you write in a stylesheet) and searches the page (the DOM). It returns **the first element that matches**, or `null` if nothing matches.

So it's like telling the DOM: "scan the page, stop at the first match, hand it to me."

11.1.1 What it can select

It uses **CSS selector syntax**, so it can match by:

ID:

```
document.querySelector("#radioMess")
```

Finds the first element with `id="radioMess"`.

Class:

```
document.querySelector(".div1")
```

Finds the first element with `class="div1"`.

Tag name:

```
document.querySelector("input")
```

Finds the first `<input>` element.

Attribute matches:

```
document.querySelector('input[name="grp1"]')
```

Finds the first `<input>` with `name="grp1"`.

Pseudo-classes (state-based selectors):

```
document.querySelector('input[name="grp1"]:checked')
```

Finds the first radio in that group that is currently selected.

11.1.2 Why it matters in your code

When your code does:

```
const selected = document.querySelector('input[name="grp1"]:checked');
```

it's not grabbing "rad1" or "rad2" directly. It's saying: "whichever radio in group `grp1` is checked right now—give me that one." That's the correct mental model for radios.

When your checkbox code does:

```
const label = document.querySelector('label[for="' + cb.id + '"]');
```

it's saying: "find the `<label>` whose `for` attribute points to this checkbox's id." That's how it pulls the human-readable text next to the checkbox.

11.1.3 querySelector vs querySelectorAll

`querySelector` returns **one element** (first match). `querySelectorAll` returns **all matches** as a NodeList.

So if you want "the selected radio" (only one), `querySelector` is perfect. If you want "every checked checkbox" (could be many), `querySelectorAll` is the right tool.

11.2 Quick Cheat Sheet

Think of each selector as a GPS address for elements in the DOM—CSS-style.

```
// By ID (finds the element with id="radioMess")
document.querySelector("#radioMess");

// By class (finds the first element with class="div1")
document.querySelector(".div1");

// By tag name (finds the first <input> on the page)
document.querySelector("input");

// By attribute (first input whose name is grp1)
document.querySelector('input[name="grp1"]');

// By attribute + state (the currently selected radio in the grp1 radio group)
document.querySelector('input[name="grp1"]:checked');

// All radios in the group (returns a NodeList of all three radios)
document.querySelectorAll('input[name="grp1"]');

// All checked checkboxes (returns a NodeList of whichever boxes are checked)
document.querySelectorAll('input[type="checkbox"]:checked');

// A specific checkbox by its id (same idea as #cb2)
document.querySelector("#cb2");

// The label attached to checkbox cb2 (matches <label for="cb2">...</label>)
document.querySelector('label[for="cb2"]');

// The label attached to a checkbox variable cb (dynamic version)
document.querySelector('label[for="' + cb.id + '"]');
```

A few "read it like English" translations:

`#radioMess` means "the element with id radioMess." `.div1` means "the first element with class div1." `input[name="grp1"]` means "an input whose name attribute is grp1." `:checked` means "only the one that's currently selected/checked." `label[for="cb2"]` means "a label whose for attribute is cb2."

One practical rule: use `querySelector` when you expect one thing (like the selected radio); use `querySelectorAll` when you expect many (like checked checkboxes). If you use `querySelectorAll`, you loop it. If you use `querySelector`, you null-check it.

12. querySelector vs getElement(s)By

`document.querySelector()` / `document.querySelectorAll()` and the old-school `document.getElementsBy...` family solve the same problem—"find me elements"—but they do it with different power tools, different return types, and different gotchas. Think of it like a modern search bar versus a set of specialized filing cabinets.

12.1 What each one is, in plain terms

`querySelector(cssSelector)` finds the **first** element that matches a **CSS selector string**.

`querySelectorAll(cssSelector)` finds **all** matching elements and returns a **static list**.

The `getElementsBy...` family are **specialized** finders:

- `getElementById(id)` arrow.r one element (or `null`)
- `getElementsByClassName(className)` arrow.r a **live HTMLCollection**
- `getElementsByTagName(tagName)` arrow.r a **live HTMLCollection**
- `getElementsByName(name)` arrow.r a **live NodeList** (mostly used for form controls)

12.2 Selector power: CSS vs "one trick per method"

This is the big difference.

With `querySelector(All)`, you can target almost anything you can describe in CSS:

- `'#menu .item.active'`
- `'input[name="grp1"]:checked'`
- `'ul > li:first-child'`
- `'.card[data-state="open"]'`

With `getElementsBy...`, you get only the built-in categories (id, class, tag, name). If your selection needs "class + inside this container + has attribute + currently checked," the `getElementsBy...` tools start to feel like trying to do calculus with a ruler.

12.3 Return types: the part that bites people

12.3.1 `querySelector()`

Returns a single `Element` (or `null`).

12.3.2 `querySelectorAll()`

Returns a `NodeList` that is **static**: it does **not** automatically update if the DOM changes after the call.

12.3.3 `getElementsByClassName()` / `getElementsByTagName()`

Return an `HTMLCollection` that is **live**: it **does** update as the DOM changes.

12.3.4 `getElementsByName()`

Returns a `NodeList` that is typically **live** as well (behavior can vary by context, but you should treat it as live in practice).

Why this matters: if you add/remove elements, a live collection can "change under your feet" while you loop it, which is a subtle way to accidentally skip items or double-handle them.

12.4 Looping and convenience

`querySelectorAll()`'s `NodeList` supports `forEach()` in modern browsers, which makes it pleasant.

`HTMLCollection` is array-like but not an array; it usually **doesn't** have `forEach()`. You can loop it with `for...of` (modern) or a classic index loop.

If you want to use array methods reliably:

```
const items = Array.from(document.getElementsByClassName("item"));
items.filter(/* ... */);
```

or

```
const items = [...document.querySelectorAll(".item")];
```

12.5 Performance: usually not your bottleneck (but here's the real story)

- `getElementById()` is extremely direct and typically fastest in micro-benchmarks.
- `getElementsByClassName()` / `getElementsByTagName()` can be fast, but the "live" behavior has overhead and can create surprising costs if you cause lots of DOM changes.
- `querySelector(All)` does more work because it parses a CSS selector, but for typical UI sizes it's "fast enough," and the clarity often wins.

If you're optimizing DOM selection in 2026, you're probably already doing something else wrong—like querying inside a loop or forcing layout repeatedly.

12.6 Practical "when to use what" (no fluff)

Use `querySelector()` when:

- you want a single thing and it's easiest to describe with CSS (`:checked`, `[data-x]`, descendant selectors, etc.)
- you want readable code that matches what you'd write in CSS

Use `querySelectorAll()` when:

- you need a snapshot of many elements and you don't want the list mutating while you work
- you want easy iteration

Use `getElementById()` when:

- you truly have an ID and it's the clearest way to say what you mean (also great for beginners)

Use `getElementsByClassName()` / `getElementsByTagName()` when:

- you specifically want a live collection that stays up to date (rare, but sometimes useful)
- you're maintaining older code and consistency matters

12.7 One concrete compare example

Say you want the currently-selected radio button in a group:

With `querySelector`:

```
const selected = document.querySelector('input[name="grp1"]:checked');
```

With `getElementsByName` you'd do more manual work:

```
const radios = document.getElementsByName("grp1");
let selected = null;
for (const r of radios) {
  if (r.checked) { selected = r; break; }
}
```

Same result. One is a laser pointer; the other is a flashlight and patience.

12.8 Common pitfalls

1. People expect `querySelectorAll()` to update when new elements are added. It won't. Call it again, or use event delegation.

2. People loop a live `HTMLCollection` while removing nodes and accidentally skip elements because the collection shrinks as they go.
3. People forget that `getElementById()` is singular and `getElementsBy...` is plural for a reason (and returns collections even if there's only one match).

13. Arrays

An array in JavaScript is an ordered collection of values. Arrays are used when you need to store multiple related items in a specific sequence and access them by position.

Each item in an array is called an element. Every element has an index, which represents its position in the array. JavaScript arrays are **zero-indexed**, meaning the first element is at index `0`, the second at index `1`, and so on. The last element is always at `array.length - 1`.

JavaScript arrays are **dynamic**. They can grow and shrink while the program is running. You can add elements, remove elements, and reorder elements at any time. Many array methods **modify the array itself**, and that fact is called out explicitly throughout this lesson.

Arrays can store mixed data types, but flexibility does not mean safety. JavaScript assumes you are paying attention.

Arrays are objects, but they are **not associative arrays**. They are designed for numeric indexing and ordered data. If you need named keys, use an object.

---

14. Creating Arrays

The preferred way to create an array is with an array literal. It is clear, readable, and avoids edge cases.

```
let cities = ["New York", "Los Angeles", "Chicago", "Houston"];
```

---

15. Accessing Array Elements

Array elements are accessed using bracket notation and an index.

```
cities[0]; // first element
cities[2]; // third element
```

If an index does not exist, JavaScript returns `undefined`. It does not throw an error.

---

16. Mutating vs Non-Mutating Operations

Some array methods **change the original array**. Others return new values and leave the original unchanged. This lesson demonstrates both, and each case is labeled clearly. Ignoring this distinction causes bugs later.

---

17. Lecture Code

This example demonstrates common array operations and displays their effects directly in the browser. The code is intentionally linear and explicit.

17.1 Complete Example (Using `querySelector`)

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width">
    <title>JavaScript Arrays</title>

    <style>
        .heading {
            color: blue;
            font-weight: bold;
        }
        .block {
            margin-bottom: 1em;
        }
    </style>

    <script>
        function write(tagId, html) {
            document.querySelector("#" + tagId).innerHTML = html;
        }

        function printArray(arr, tagId, heading) {
            const output = arr.join("<br>");

            write(tagId,
                "<div class='block'>" +
                "<div class='heading'>" + heading + "</div>" +
                output +
                "</div>"
            );
        }

        function findItem(arr, value) {
            return arr.indexOf(value);
        }
    </script>
</head>

<body>

<p id="p0"></p>
<p id="p1"></p>
<p id="p2"></p>
<p id="p3"></p>
<p id="p4"></p>
<p id="p5"></p>
<p id="p6"></p>
<p id="p7"></p>
<p id="p8"></p>
<p id="p9"></p>
<p id="p10"></p>
<p id="p11"></p>
<p id="p12"></p>
<p id="p13"></p>
<p id="p14"></p>
<p id="p15"></p>

<script>
```

```
// Create an array
let cities = [
    "New York",
    "Los Angeles",
    "Chicago",
    "Houston",
    "Phoenix",
    "San Diego",
    "Dallas",
    "Kalispell"
];

printArray(cities, "p0", "Original Array");

// Access individual elements
write("p1",
    cities[0] + "<br>" +
    cities[2] + "<br>" +
    cities[4]
);

// pop() removes the last element (mutates the array)
let removedCity = cities.pop();
printArray(cities, "p2", "After pop() — removed: " + removedCity);

// push() adds elements to the end (mutates the array)
cities.push("Toledo", "Big Fork");
printArray(cities, "p3", "After push()");

// sort() orders strings alphabetically (mutates the array)
cities.sort();
printArray(cities, "p4", "After sort()");

// reverse() reverses order (mutates the array)
cities.reverse();
printArray(cities, "p5", "After reverse()");

// shift() removes the first element (mutates the array)
cities.shift();
printArray(cities, "p6", "After shift()");

// unshift() adds elements to the beginning (mutates the array)
cities.unshift("Las Vegas");
printArray(cities, "p7", "After unshift()");

// join() creates a string from the array (does NOT mutate)
let cityString = cities.join(" | ");
write("p8",
    "<div class='heading'>Join Result</div>" +
    cityString
);

// split() creates an array from a string
let names = "Jim,Bob,George,Harry,Daryl,Daryl";
let nameArray = names.split(",");
printArray(nameArray, "p9", "Array Created with split()");

// splice() removes elements by index (mutates the array)
cities.splice(1, 1);
printArray(cities, "p10", "After splice() — remove one item");

// splice() can insert elements (mutates the array)
cities.splice(1, 0, "Dillon", "Helena");
printArray(cities, "p11", "After splice() — insert items");

// slice() creates a new array (does NOT mutate)
```

```
    let subset = cities.slice(2, 5);
    printArray(subset, "p12", "Result of slice() (new array)");
    printArray(cities, "p13", "Original Array After slice()");

    // indexOf() search – found
    let searchItem = "Chicago";
    let index = findItem(cities, searchItem);

    write("p14",
        "<div class='heading'>Search Result (found)</div>" +
        "Index: " + index + "<br>" +
        "Value: " + cities[index]
    );

    // indexOf() search – not found
    searchItem = "HooBoo";
    index = findItem(cities, searchItem);

    write("p15",
        "<div class='heading'>Search Result (not found)</div>" +
        "Returned value: " + index
    );
</script>

</body>
</html>
```

18.

19. Random Functions

JavaScript random always returns a number between 0 (inclusive) and 1 (exclusive).

```
Math.random()
```

`Math.floor` is a function commonly found in programming languages that deals with numbers. It's specifically used for rounding down a number to the nearest whole integer (whole number).

Here's what `math.floor` does:

- It takes a number (integer or decimal) as input.
- It ignores any digits after the decimal point.
- It returns the largest integer that is less than or equal to the input number.

```
function RandomIntMax(max)
{
    // to make the max inclusive we add one
    return Math.floor(Math.random() * (max + 1));
}

function RandomIntMinMax(min,max)
{
    // to make the max inclusive we add one
    return Math.floor(Math.random() * ((max+1) - min) ) + min;
}
```

20. Importance of Form Validation

HTML form validation is important for several reasons:

Improved User Experience:

- **Reduces frustration:** By catching errors on the client-side (user's browser) before submission, users are prevented from submitting invalid data that would otherwise cause errors or require them to refill the form.
- **Provides clear feedback:** Validation messages inform users exactly what's wrong with their input, allowing them to correct errors easily.

Enhanced Data Quality:

- **Ensures accurate data:** Validation helps prevent users from entering invalid or nonsensical data into forms. This is crucial for ensuring the integrity of data collected through forms.
- **Reduces processing errors:** When forms contain invalid data, it can lead to errors on the server-side when the data is processed. Validation helps prevent these errors by ensuring clean data reaches the server.

Security:

- **Prevents malicious attacks:** Certain form validation techniques can help mitigate security vulnerabilities like SQL injection attacks, where malicious code is injected into forms to manipulate data or gain unauthorized access.

Reduced Server Load:

- **Less server-side validation:** By validating on the client-side, you can reduce the amount of validation required on the server-side, improving server performance and efficiency.

However, it's important to remember that client-side validation is not a foolproof security measure. Malicious users can still tamper with the code or bypass client-side validation altogether. Therefore, server-side validation is still essential to ensure the security and integrity of your data.

In summary, HTML form validation plays a vital role in creating a smooth user experience, improving data quality, enhancing security, and reducing server load. It's a valuable tool for any web application that relies on user input through forms.

21. Form Validation

This example uses a "flag" approach.

21.1 Lecture Code

index.html

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <meta name="viewport" content="width=device-width">
        <title>Form Validation Example</title>
        <!-- Programmer: James Goudy -->

        <style>
            #theContent{
                width: 25%;
                margin: 100px auto; /* Combined margin properties */
                background-color: bisque;
                padding: 40px;
                color: black;
            }
            #errorDiv{
                background-color: khaki;
                color: navy;

            }
        </style>

    </head>
    <body>
        <div id="theContent">
        <h1>Form Validation Example</h1>

        <form name="demo" onsubmit="return validateFormOnSubmit(this)"
action="test.html">
            <table summary="Demonstration form">
                <tbody>
                    <tr>
                        <td><label for="username">Username:</label></td>
                        <td><input  type="text" id="username" name="username"
                                    size="35" maxlength="50" ></td>
                    </tr>
                    <tr>
                        <td><label for="pwd">Password:</label></td>
                        <td><input  type="password" id="pwd" name="pwd"
                                    size="35" maxlength="25" ></td>
                    </tr>
                    <tr>
                        <td><label for="email">Email:</label></td>
                        <td><input  type="text" id="email" name="email"
                                    size="35" maxlength="30"
                                    placeholder="xxx@domain.xxx"></td>
                    </tr>
                    <tr>
                        <td><label for="altemail">Alternate email:</label></
td>
                        <td><input  type="text" id="altemail" name="altemail"
                                    size="35" maxlength="30"
                                    pattern="[a-z0-9._%+\-]+@[a-z0-9.\-]+\.[a-
z]{2,}$"
                                    placeholder="xxx@domain.xxx" required></
td>
                    </tr>
                    <tr>
```

```
                        <td><label for="phone">Phone:</label></td>
                        <td><input  type="text" id="phone" name="phone"
size="35"
                                    maxlength="25" placeholder="(555)
555-5555"></td>
                    </tr>
                    <tr>
                        <td>
                            <label for="from">Where are you :</label></td>
                        <td><input  type="text" id="from" name="from"
                                    size="35" maxlength="50" ></td>
                    </tr>
                    <tr>
                        <td> </td>
                        <td><input name="Submit" value="Send" type="submit"
></td>
                        <td> </td>
                    </tr>
                </tbody>
            </table>
        </form>

        <div id="errorDiv">

        </div>

        </div>

        <script>
            /*
                This code defines a function that validates a form.
                The function breaks down the form validation into
                smaller functions, each of which checks
                a single form element. If an element is valid,
                the function checking it doesn't return any error message.
                Otherwise, it returns an error message
                describing the problem and highlights
                the element in yellow for the user to see.
             */

            function validateFormOnSubmit(theForm) {
                let reason = "";


                reason += validateUsername(theForm.username);
                reason += validatePassword(theForm.pwd);
                reason += validateEmail(theForm.email);
                reason += validatePhone(theForm.phone);
                reason += validateEmpty(theForm.from);

                if(reason != "")
                {
                    reason = "<h2>Errors</h2><ul>" + reason + "</ul><br>";
                    document.getElementById("errorDiv").innerHTML = reason;
                    return false;
                }

                document.getElementById("errordiv").innerHTML = "";
                return true;
            }

            /* *
             The function below checks if a required field has been left
             empty. If the required field is blank,
             returned is the error string to the main function.
             If it is not blank, the function returns an empty string.
```

```
 */

function validateEmpty(fld) {
    let error = "";

    if (fld.value.length == 0) {
        fld.style.background = 'Yellow';
        error = "<li>The required field has not been filled in.</
li>"
    } else {
        fld.style.background = 'White';
    }
    return error;
}

/*
It first makes sure the username field isn't left empty.
Then, it checks if the username's length is between 5 and 15
characters.
If it passes those checks, it uses a special pattern
(called a regular expression) to make sure the username
only contains letters, numbers, and underscores. No other symbols
or
characters are allowed.
*/
function validateUsername(fld) {
    let error = "";
    let illegalChars = /\W/; // allow letters, numbers, and
underscores

    if (fld.value == "") {
        fld.style.background = 'Yellow';
        error = "<li>You didn't enter a username.</li>";
    } else if ((fld.value.length < 5) || (fld.value.length > 15))
{
        fld.style.background = 'Yellow';
        error = "<li>The username is the wrong length.</li>";
    } else if (illegalChars.test(fld.value)) {
        fld.style.background = 'Yellow';
        error = "<li>The username contains illegal characters.</
li>";
    } else {
        fld.style.background = 'White';
    }
    return error;
}
/*
The function below checks the password field for blankness
 and allow only letters and numbers - no underscopes this time.
 So we should use a new regular expression to forbid underscopes.
 This one /[\W_]/ allow only letters and numbers.
 Next, we want to permit only passwords that contain letters
 and at least one numeral. For that we use the seacrh() method and
  two more regular expressions: /(a-z)+/ and /(0-9)/.
*/
function validatePassword(fld) {
    let error = "";
    let illegalChars = /[\W_]/; // allow only letters and numbers

    if (fld.value == "") {
        fld.style.background = 'Yellow';
        error = "<li>You didn't enter a password.</li>";
    } else if ((fld.value.length < 7) || (fld.value.length > 15))
{
        error = "<li>The password is the wrong length. </li>";
        fld.style.background = 'Yellow';
```

```
                } else if (illegalChars.test(fld.value)) {
                    error = "<li>The password contains illegal characters.</
li>";
                    fld.style.background = 'Yellow';
                } else if (!((fld.value.search(/(a-z)+/)) &&
(fld.value.search(/(0-9)+/)))) {
                    error = "<li>The password must contain at least one
numeral.</li>";
                    fld.style.background = 'Yellow';
                } else {
                    fld.style.background = 'White';
                }
                return error;
            }


            //
---------------------------------------------------------------------
            /*
            "The following steps are undertaken to validate the user-provided
email address:

            1. Empty Field Check: An initial check is performed to determine
if the user
            has entered any data in the email field.

            2. Regular Expression Validation: If the field is not empty, a
regular expression is
            employed in conjunction with the test() method to verify if the
email address
            adheres to the following format:

                    * Presence of an "@" symbol, but not as the leading
character.
                    * Presence of a dot (".") following the "@" symbol.

            It is acknowledged that this validation approach has limitations.
While it may not
            capture all non-compliant email addresses, it serves as a robust
foundation for
            initial validation purposes."
             *
             */

            function trim(s)
            {
                return s.replace(/^\s+|\s+$/, '');
            }

            function validateEmail(fld) {
                let error = "";
                // value of field with whitespace trimmed off
                let tfld = trim(fld.value);
                let emailFilter = /^[^@]+@[^@.]+\.[^@]*\w\w$/;
                let illegalChars = /[\(\)\<\>\,\;\:\\\"\[\]]/;

                if (fld.value == "") {
                    fld.style.background = 'Yellow';
                    error = "<li>You didn't enter an email address.</li>";
                } else if (!emailFilter.test(tfld)) {
                    //test email for illegal characters
                    fld.style.background = 'Yellow';
                    error = "<li>Please enter a valid email address.</li>";
                } else if (fld.value.match(illegalChars)) {
                    fld.style.background = 'Yellow';
                    error = "<li>The email address contains illegal
```

```
characters.</li>";
                } else {
                    fld.style.background = 'White';
                }
                return error;
            }

        //-------------------------------------------------------------------

            /*
            1. Clear Spacer Characters: It uses replace() to remove any extra
spaces
               or characters that might be in the phone number.
            2. Check for Numbers Only: It calls isNaN() to make sure the phone
number
               contains only numbers, not letters or symbols.
            3. Check Length: It verifies that the phone number has exactly 10
digits,
               as that's the standard length for many phone numbers.
             */

            function validatePhone(fld) {
                let error = "";
                let stripped = fld.value.replace(/[\(\)\.\-\ ]/g, '');
                console.log(stripped);
                if (fld.value === "") {
                    error = "<li>You didn't enter a phone number.</li>";
                    fld.style.background = 'Yellow';
                } else if (isNaN(stripped)) {
                    console.log(stripped);
                    error = "<li>The phone number contains illegal
characters.</li>";
                    fld.style.background = 'Yellow';
                } else if (!(stripped.length == 10)) {
                    error = "<li>The phone number is the wrong length. " +
                            "Make sure you included an area code.</li>";
                    fld.style.background = 'Yellow';
                }
        else
        {
          fld.style.background = 'White';
        }
                return error;
            }
        </script>

    </body>
</html>
```

test.html

```
<html>
<head>
<title>test success</title>
</head>
  <style>
    #theContent{
      width: 25%;
      margin: 100px auto; /* Combined margin properties */
      background-color: bisque;
      padding: 40px;
      color: black;
    }
  </style>
<body>
  <div id="theContent">
```

```
    <h1>Good test</h1>
    <p>There was good form validation</p>
  </div>

</body>
</html>
```

21.2 Explanation

The code you provided is a basic HTML form with Javascript validation. Let's break it down into sections:

HTML Structure:

- **DOCTYPE declaration:** This line specifies the document type as HTML.
- **HTML tag:** This is the root element of the HTML document.
- Head section:

This section contains meta information about the document.

- **Meta charset:** This specifies the character encoding used in the document (UTF-8 in this case).
- **Meta viewport:** This sets how the document should be scaled on different screen sizes (set to device width here).
- **Title:** This defines the title of the webpage displayed on the browser tab.
- Style section:

This section contains CSS code that styles the HTML elements.

- In this case, styles are defined for two divs with ids `theContent` and `errorDiv`. The content div has a width of 50%, centered with margin, background color, padding, and text color. The error div has a background color and text color.
- Body section:

This section contains the visible content of the webpage.

-

`Div` with `id` "theContent" This div contains the form and the error message area.
    - **Heading:** This displays the title "Form Validation Example".
    - Form:

This is the HTML form element that users will interact with to submit data.

- Form attributes:
    - name: "demo" (specifies a name for the form)
    - `onsubmit: "return validateFormOnSubmit(this)"` (calls the validateFormOnSubmit function when the form is submitted and prevents default form submission behavior).
    - `action: "test.html"` (specifies the file where the form data will be submitted, but not used in this example since validation is done with Javascript).
- Table:

This creates a table layout for the form elements.

- Each row represents an input field and its label.
- Input elements have different types (`text`, `password`) and attributes like size, maxlength, and placeholder.
- **Submit button:** This button triggers the form submission when clicked.
- `Div` with `id` `errorDiv`: This div will be used to display any error messages from the form validation.
- Script section:

This section contains Javascript code that adds interactivity to the webpage.

- The script defines several functions for form validation:
  - `validateFormOnSubmit`: This function is called when the form is submitted. It calls other validation functions for each form element and builds an error message string if any errors are found. If there are errors, it displays them in the error div and prevents the form from being submitted. Otherwise, it clears the error div and allows form submission.
  - `validateEmpty`: This function checks if a required field is empty. If empty, it sets the background color of the field to yellow and adds an error message to the error string.
  - `validateUsername`: This function checks the username field for validity. It checks for blank input, length restrictions, and illegal characters. It sets the background color of the field based on the validation result and adds an error message to the error string if needed.
  - `validatePassword`: This function checks the password field for validity. It checks for blank input, length restrictions, illegal characters, and presence of at least one numeral. It sets the background color of the field based on the validation result and adds an error message to the error string if needed.
  - `trim`: This function removes leading and trailing whitespace from a string (used in email validation).
  - `validateEmail`: This function checks the email field for validity. It checks for blank input, format using a regular expression, and illegal characters. It sets the background color of the field based on the validation result and adds an error message to the error string if needed.
  - `validatePhone`: This function checks the phone number field for validity. It removes non-numeric characters, checks if the remaining string contains only numbers, and checks the length of the number. It sets the background color of the field based on the validation result and adds an error message to the error string if needed.

Overall, this code demonstrates a simple form with client-side validation using Javascript. When the user submits the form, the Javascript code validates each field and provides feedback to the user if there are any errors.

22. Form Validation - Onblur

The `onblur` attribute in HTML is used to attach an event listener to an element. This event listener fires whenever the element loses focus. In simpler terms, the function specified in the `onblur` attribute gets executed when the user clicks away from the element.

Here's a breakdown of how it works:

1. **Attaching the Event Listener:** The `onblur` attribute is assigned to an HTML element (like `<input>`, `<textarea>`, etc.). It takes a JavaScript expression (usually a function call) as its value.
2. **Triggering the Event:** When the user interacts with the element (clicks on it, types text, etc.) and then clicks outside the element (clicks on another element or the background), the element loses focus.
3. **Executing the Function:** As soon as the element loses focus, the JavaScript expression specified in the `onblur` attribute gets executed. This function can perform various actions depending on the element and the desired functionality.

Here are some common use cases for the `onblur` event:

- **Validation:** You can use `onblur` to validate user input in form fields. For example, you could check if a field is empty, has the correct format (email, phone number, etc.), or meets specific criteria.
- **Dynamic Updates:** When the user leaves a field, you might use `onblur` to update other parts of the form or the webpage dynamically based on the entered value.
- **Hiding Elements:** You could use `onblur` to hide instructional text or error messages that appear when the user focuses on a field.

22.1 Lecture Code

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Form 2b</title>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width">
  <!- By James Goudy -->

  <style>
    .error { /* Combined error styling for reusability */
      display: none;
      position: absolute;
      width: 200px;
      left: 160px;
      bottom: 2px;
      color: red;
      padding: 2px;
    }
    .input-container { /* Descriptive class name */
      position: relative;
    }
    #areaA {
      width: 50%;
      margin: 100px auto; /* Combined margin properties */
      background-color: navy;
      padding: 10px;
      color: white;
    }
  </style>
</head>
<body>
  <div id="areaA">
    <h1>Onblur Example</h1>
```

```
  <div class="input-container">
    <div class="error" id="nameError">Enter Your Name</div>
    Name <input id="textboxName" type="text" style="width: 100px"
                onfocus="hideError(this)" onblur="validateName(this)">
  </div>
  <br>
  <div class="input-container">
    <div class="error" id="zipError"></div>
    Zip <input id="textboxZip" type="text" style="width: 100px"
                onfocus="hideError(this)" onblur="validateZip(this)">
  </div>

  <form onsubmit="return validateForm()" action="test.html">
    <br>
    <input type="submit" value="Enter">
  </form>
</div>
<script>
  let nameValid = false;
  let zipValid = false;

  function hideError(element) {
    element.parentNode.querySelector('.error').style.display = 'none';
  }

  function validateName(element) {
    const nameInput = document.getElementById("textboxName");
    const nameError = document.getElementById("nameError");

    if (nameInput.value === "") {
      nameError.style.display = 'block';
      nameValid = false;
      return;
    }

    nameValid = true;
  }

  function validateZip(element) {
    const zipInput = document.getElementById("textboxZip");
    const zipError = document.getElementById("zipError");


    if (zipInput.value === "") {
      zipError.innerHTML = "Zip - is empty";
      zipError.style.display = 'block';
      zipValid = false;
      return;
    } else if (zipInput.value.length != 5) {
      zipError.innerHTML = "Zip - wrong length";
      zipError.style.display = 'block';
      zipValid = false;
      return;
    } else if (isNaN(zipInput.value)) {
      zipError.innerHTML = "Zip - numbers only";
      zipError.style.display = 'block';
      zipValid = false;
      return;
    }

    zipValid = true;
  }

  function validateForm() {
    return nameValid && zipValid;
  }
```

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

page2.html - if there was successful form validation

```
<!DOCTYPE html>
<html lang="en">
<head>
<title>test success</title>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width">
  <!-- By James Goudy -->

  <style>
  .areaA {
      width: 50%;
      margin: 100px auto; /* Combined margin properties */
      background-color: navy;
      padding: 10px;
      color: white;
    }
  </style>
</head>
<body>
  <div class="areaA">
  <h1>Good test<br>Form 2<br>On Blurr</h1>
  </div>
</body>
</html>
```

22.2 Explanation

The provided code implements a simple form with two input fields and a submit button. Here's a breakdown of its functionality:

HTML Structure:

- The code defines a basic HTML document with a title ("Form 2b").
- It sets the character encoding to UTF-8 and ensures proper responsiveness on various devices using the viewport meta tag.
- Styles are defined within a `<style>` tag.
- The body contains a container element with the ID "areaA".
    - Inside "areaA", there's a heading (`<h1>`) displaying "Onblur Example".
    - Two input fields are created using `<div>` elements with the class "input-container".
        - Each input container holds:
            - An error message element (`<div>`) with the class "error" and a unique ID (initially hidden).
            - A label ("Name" or "Zip") followed by a text input field (`<input>`) with an ID and event listeners.
    - A form element (`<form>`) is placed after the input fields.
        - The form has an "onsubmit" event listener that calls the `validateForm` function and sets the form action to "test.html" (where the form data might be submitted).
        - Inside the form, a submit button (`<input type="submit">`) is displayed with the value "Enter".

JavaScript Functionality:

- Two variables, `nameValid` and `zipValid`, are declared with initial values of `false`, indicating that neither field is validated by default.

- Three functions are defined:
  - `hideError(element)`: This function takes an element as input (presumably an input field). It finds the closest error element (using `querySelector`) within the same container (`parentNode`) and hides it (sets `display` to "none").
  - `validateName(element)` : This function validates the name input field.
    - It retrieves the name input element and its corresponding error element using their IDs.
    - It checks if the name input value is empty. If so, it displays the error message ("Enter Your Name"), sets the error element to visible (`display: block`), and sets `nameValid` to `false`.
    - If the name is not empty, it sets `nameValid` to `true`.
  - `validateZip(element)` : This function validates the zip code input field. It follows a similar logic to the `validateName` function:
  - It retrieves the zip code input element and its corresponding error element.
    - It checks if the zip code value is empty. If so, it displays an error message ("Zip - is empty"), sets the error visibility, and sets `zipValid` to `false`.
    - If the zip code is not empty, it further validates the length (must be 5 characters) and if it contains only numbers (using `isNaN`). If either validation fails, it displays an appropriate error message, sets error visibility, and sets `zipValid` to `false`. Otherwise, it sets `zipValid` to `true`.
- The `validateForm` function checks if both `nameValid` and `zipValid` are true. It returns `true` if both fields are valid, allowing form submission; otherwise, it returns `false`, preventing form submission until errors are corrected.

Event Listeners:

- The name and zip code input fields have two event listeners each:
  - `onfocus`: When the user focuses on an input field (clicks on it), the `hideError` function is called, presumably to hide any previous error messages for that field.
  - `onblur`: When the user leaves an input field (clicks elsewhere), the corresponding validation function (`validateName` or `validateZip`) is called to check the field's value and display any errors if necessary.

In summary, this code implements a simple form with error handling for name and zip code input fields. It ensures that the user enters a name and a valid zip code (5 digits, numbers only) before allowing form submission.

23. JavaScript Placement

You can include JavaScript code in an HTML document using the dedicated `<script>` tag. The placement of this tag depends on when you want the JavaScript to load:

1. **Inside the `<head>` section**:
   - Insert the `<script>` tag between the opening `<head>` and closing `</head>` tags. Place your JavaScript code inside it.
   - This ensures that the script is loaded and executed when the page loads. However, it can cause a delay in rendering the page because the browser waits for the script to download and execute before continuing with other HTML parsing.
2. **Inside the `<body>` section**:
   - You can place the `<script>` tags containing your JavaScript anywhere within the `<body>` tags.
   - While this approach allows the page to render faster, it may still cause a slight delay if the script is large or resource-intensive.
3. **Just before the `</body>` close tag**:
   - Some developers prefer placing the `<script>` tags just before the closing `</body>` tag.
   - This ensures that the entire HTML content is parsed and displayed before the JavaScript is loaded.
   - However, be cautious: if the script modifies the page content (e.g., via `document.write()`), it can disrupt the rendering process.

Choose the placement based on your specific requirements and performance considerations.

---

24. Objects By Class

JavaScript classes are a relatively new way to structure code using object-oriented programming (OOP) principles. While JavaScript itself is prototype-based, classes act like syntactic sugar on top of that foundation, making it easier for programmers coming from other OOP languages like Java or C++ to adapt.

Here's a breakdown of key concepts in JavaScript classes:

- **Structure:** Classes are defined using the `class` keyword. Inside the class, you can define properties (like variables) and methods (like functions) that act on the data.
- **Constructors:** A constructor is a special method called with the `new` keyword when creating a new object (called an instance) from the class. It's typically used to initialize the object's properties.
- **Inheritance:** Classes can inherit properties and methods from other classes, promoting code reuse.
- **Encapsulation (sort of):** While JavaScript doesn't have strict public/private access modifiers like some languages, you can simulate encapsulation using conventions like starting property names with underscores to discourage direct modification outside the class.

JavaScript introduced private members with the introduction of classes in ES6 (ECMAScript 2015).

Here's a breakdown of private members in JavaScript:

Declaring Private Members:

- Private members are denoted using a hash (#) symbol before the property name.
- This creates a special kind of property that cannot be accessed directly outside of the class.

Benefits of Private Members:

- **Encapsulation:** Protects internal state of the object from unintentional modification.
- **Abstraction:** Exposes only necessary functionalities through public methods.
- **Reduced naming conflicts:** Protects private members from clashes with public names.

Limitations:

- **Accessibility:** While private members are enforced by JavaScript, a determined developer can still access them through trickery. However, it's generally not recommended practice.
- **Constructors:** Constructors themselves cannot be private in JavaScript.

Overall, JavaScript classes provide a familiar way to structure object-oriented code, making it easier to reason about complex applications and maintain larger codebases.

Here are some resources to learn more about JavaScript classes:

- JavaScript Classes - MDN Web Docs: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/class
- JavaScript Classes – How They Work with Use Case Example: https://www.freecodecamp.org/news/javascript-classes-tutorial/

For the example below:

The `Person` class has the following constructor parameters:

- `fname`: The first name of the person.
- `lname`: The last name of the person.
- `city`: The city where the person lives.

The `Person` class also has the following methods:

- `getfname`: A getter method that returns the value of the `fname` property.
- `setfname`: A setter method that sets the value of the `fname` property.
- `getlname`: A getter method that returns the value of the `lname` property.
- `setlname`: A setter method that sets the value of the `lname` property.
- `getcity`: A getter method that returns the value of the `city` property.
- `setcity`: A setter method that sets the value of the `city` property.
- `fullNameCity`: A method that returns the full name of the person and the city where they live.

24.1 Lecture Code

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Classes Objectes</title>

  <script>
    // Define a Person class to represent a person with first name, last name,
and city
    class Person {

      // Constructor to initialize a Person object with given name and city
      constructor(fname, lname, city) {
        this.fname = fname;
        this.lname = lname;
        this.city = city;
      }

      // Getters and Setters for accessing and modifying properties
      get getfname() {
        return this.fname;
      }

      set setfname(fname) {
        this.fname = fname;
      }

      get getlname() {
        return this.lname;
      }

      set setlname(lname) {
        this.lname = lname;
      }

      get getcity() {
        return this.city;
      }

      set setcity(city) {
        this.city = city;
      }

      // Method to return the full name of the person along with their city
      fullNameCity() {
        let fullName = this.fname + " " + this.lname + ", " + this.city;
        return fullName;
      }
    }

    // Lists containing names and cities for random generation
    let firstList = ["John", "Mary", "Abe", "Jim", "Stanley", "Salley",
```

```
"Ann"];
    let lastList = ["Rodes", "Smith", "Miller", "Jones", "Stucci", "Kindreck",
"Patton"];
    let cityList = ["Kali", "Polson", "Libby", "Whitefish", "Dayton", "Noxon",
"Bozeman"];

    let people = new Array(); // Array to store Person objects

    // Function to generate a random integer between 0 (inclusive) and max
(inclusive)
    function RandomIntMax(max) {
      // + 1 to make the max inclusive
      return Math.floor(Math.random() * (max + 1));
    }

    // Function to generate a random integer between min (inclusive) and max
(inclusive)
    function RandomIntMinMax(min, max) {
      // + 1 to make the max inclusive
      return Math.floor(Math.random() * ((max + 1) - min) ) + min;
    }

    // Function to create a specified number of Person objects with random
names and cities
    function createPeople(numPeople) {

      let test = ""; // Temporary string for building output (not used in
final version)

      for (let i = 0; i < numPeople; i++) {
        let rfname = firstList[RandomIntMax(firstList.length - 1)];
        let rlname = lastList[RandomIntMax(lastList.length - 1)];
        let rcity = cityList[RandomIntMax(cityList.length - 1)];

        // Uncomment these lines to display temporary output during person
creation (for debugging purposes)
        //test = test + (rfname + " " + rlname + ", " + rcity +"\n<br>");

        xperson = new Person(rfname, rlname, rcity);
        people[i] = xperson;
      }

      // Uncomment this line to display temporary output after person creation
(for debugging purposes)
      //document.getElementById("test").innerHTML = test;
    }

    // Function to display the full names and cities of all Person objects in
the people array
    function displayPeople() {
      let strpeople = "";

      for (let i = 0; i < people.length; i++) {
        strpeople = strpeople + people[i].fname + " " + people[i].lname +
          " " + ", " + people[i].city + "\n<br>";
      }

      document.getElementById("test2").innerHTML = strpeople;
    }
  </script>

</head>
<body>

  <p id="test"></p>
  <br><hr><br>
```

```
  <p id="test2"></p>

  <script>
    createPeople(5);
    displayPeople();
  </script>
</body>
</html>
```

24.2 Explaination

This code creates a webpage that allows users to generate a random list of people with names and city locations. Here's a breakdown of the different sections:

HTML Structure (head and body):

- Defines the document type (`<!DOCTYPE html>`)
- Sets the language to English (`<html lang="en">`)
- Creates the

`<head>`

section containing document metadata:

- Character encoding (`<meta charset="utf-8">`)
- Viewport configuration for responsive design (`<meta name="viewport" content="width=device-width">`)
- Page title (`<title>Objects By Class</title>`)
- Styles for the content area (`<style>...</style>`)
- Creates the

`<body>`

section containing the visible content:

- A styled content area (

`<div class="theContent">...</div>`

)

- Page heading (`<h1>Objects By Class</h1>`)
- Line break, horizontal rule, and another line break for separation
- User input for number of people (

`<label>...<input>...<button>...`

)

- Label (`<label>`) explains the input field
- Input field (`<input>`) allows users to enter a number between 0 and 25
- Button (`<button>`) triggers the `createPeople()` function on click
- Paragraph element with an empty ID (`<p id="ppl"></p>`) to display the generated list
- Link to another webpage (`<a href="personObject.html">Objects By Function</a>`)

JavaScript Code (script):

- Defines a

`Person`

class to represent a person:

- Constructor (`constructor(fname, lname, city)`) initializes a new `Person` object with first name, last name, and city properties.
- Getter and setter methods for each property:
  - `getfname`, `setfname` for first name

    - `getlname`, `setlname` for last name
    - `getcity`, `setcity` for city
- `fullNameCity()` method that returns the full name of the person with their city (e.g., John Smith, Kali)
- Creates three lists containing names and locations for random generation:
    - `firstList`: Array of first names
    - `lastName`: Array of last names
    - `cityList`: Array of city names
- Creates an empty array called `people` to store `Person` objects.
- Defines two helper functions for generating random integers:
    - `RandomIntMax(max)` generates a random integer between 0 (inclusive) and `max` (inclusive).
    - `RandomIntMinMax(min, max)` generates a random integer between `min` (inclusive) and `max` (inclusive).
- Defines the

`createPeople()`

function:

- Clears the `people` array (`people.length = 0`).
- Gets the number of people entered by the user (`numPeople`).
- Limits the number of people to 25 if the user enters a higher value.
- Loops to create the specified number of

`Person`

objects:

- Randomly selects first name, last name, and city from the respective lists.
- Creates a new `Person` object with the random names and city.
- Adds the new `Person` object to the `people` array.
- Clears the user input field (`document.getElementById("numppl").value = ""`).
- Calls the `displayPeople()` function to show the generated list.
- Defines the

`displayPeople()`

function:

- Initializes an empty string (`strpeople`) to store the list.
- Loops through the

`people`

array:

- For each `Person` object, retrieves their full name and city using the `fullNameCity()` method.
- Appends the full name and city information to the `strpeople` string with line breaks for formatting.
- Sets the content of the paragraph element with ID “ppl” to the generated list (`document.getElementById("ppl").innerHTML = strpeople`).

Overall Functionality:

1. When the webpage loads, the user sees a heading, input field for number of people, a button labeled “Enter”, and an empty paragraph area.
2. The user enters a number between 0 and 25 (or keeps the default 0) and clicks the “Enter” button.
3. The `createPeople()` function is triggered.

4. The function generates the specified number of `Person` objects with random names and cities.
5. It then displays the full name and city information of each person in the paragraph area with line breaks for better readability.

25. Objects By Function

Create objects by use of a function.

25.1 Lecture Code

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width">
  <title>Objects By Function</title>
  <style>
    .theContent {
      width: 25%;
      margin: 100px auto; /* Combines margin properties for top/bottom and
left/right centering */
      background-color: bisque;
      padding: 40px;
      color: black;
    }

    #numppl {
      width: 50px;
    }
  </style>
  <script>
    function Person(fname, lname, city) {
      this.fname = fname;
      this.lname = lname;
      this.city = city;
      this.fullname = () => {
        return this.fname + " " + this.lname + ", " + this.city;
      }
    }

    // Lists containing names and cities for random generation
    let firstList = ["John", "Mary", "Abe", "Jim", "Stanley", "Salley",
"Ann"];
    let lastList = ["Rodes", "Smith", "Miller", "Jones", "Stucci", "Kindreck",
"Patton"];
    let cityList = ["Kali", "Polson", "Libby", "Whitefish", "Dayton", "Noxon",
"Bozeman"];

    let people = new Array(); // Array to store Person objects

    // Function to generate a random integer between 0 (inclusive) and max
(inclusive)
    function RandomIntMax(max) {
      // + 1 to make the max inclusive
      return Math.floor(Math.random() * (max + 1));
    }

    function createPeople() {

      people.length = 0;

      let numPeople = Number(document.getElementById("numppl").value);

      if (numPeople > 25) {
        numPeople = 25;
      }

      for (let i = 0; i < numPeople; i++) {
        const rfname = firstList[RandomIntMax(firstList.length - 1)];
        const rlname = lastList[RandomIntMax(lastList.length - 1)];
```

```
        const rcity = cityList[RandomIntMax(cityList.length - 1)];

        const xperson = new Person(rfname, rlname, rcity); // Use `new`
keyword to create a new Person object
        people[i] = xperson;
      }

      document.getElementById("numppl").value = '';
      displayPeople();
    }


    // Function to display the full names and cities of all Person objects in
the people array
    function displayPeople() {
      let strpeople = "";

      for (let i = 0; i < people.length; i++) {
        strpeople += people[i].fullname() + "\n<br>"; // Use the fullname
method of each Person object
      }
      console.log(strpeople);
      document.getElementById("ppl").innerHTML = strpeople;
    }
  </script>
</head>

<body>
  <div class="theContent">
    <h1>Objects By Function</h1>

    <br><hr><br>

    <label for="numppl">Enter the number of people:</label>
    <input type="number" name="numppl" id="numppl" min="0" max="25" value="0"
placeholder="1-25">
    <button onclick="createPeople()">Enter</button>

    <p id="ppl"></p>

    <p><a href="index.html">Objects By Class</a></p>
  </div>
</body>

</html>
```

25.2 Explaination

This code creates a webpage that allows users to generate a random list of people with full names and cities. Here's a breakdown of the different parts:

HTML (Structure and Appearance):

- The code starts with the standard HTML structure (`DOCTYPE`, `<html>`, `<head>`, `<body>`).
- Inside the `<head>` section, it defines the character encoding (`<meta charset="utf-8">`), sets the viewport for mobile devices (`<meta name="viewport"...>`), and sets the page title (`<title>`).
- There's also a `<style>` section that defines styles for a class named `.theContent` (likely for a content area). This style applies width, margins, background color, padding, and text color. Another style targets the element with the id `#numppl` and sets its width.
- The `<body>` section contains the main content displayed on the webpage wrapped in a `<div>` with the class `.theContent`.
- Inside the content area (

```
<div>
```

):

- There's a heading (`<h1>`) for the page title ("Objects By Function").
- A horizontal line (`<hr>`) for separation.
- A label (`<label>`) for the number of people input field with the `for` attribute linking it to the input element's id.
- An input field (`<input>`) with type "number", id "numppl", minimum value (0), maximum value (25), initial value (0), and a placeholder ("1-25").
- A button (`<button>`) with text "Enter" that triggers the `createPeople()` function when clicked (using the `onclick` attribute).
- A paragraph element (`<p>`) with the id "ppl" where the generated list of people will be displayed.
- Another paragraph with a link (`<a>`) to "Objects By Class" (likely another webpage).

JavaScript (Functionality):

- Inside the `<head>` section, there's a `<script>` section containing the JavaScript code that makes the page interactive.
- The code defines a function called `Person(fname, lname, city)`. This function is a constructor for creating Person objects. It takes three arguments: first name (`fname`), last name (`lname`), and city (`city`). Inside the function, it uses the `this` keyword to assign these arguments as properties of the newly created object. It also defines another property called `fullname` using an arrow function. This `fullname` function returns a string by concatenating the first name, last name, comma, space, and city.
- Three lists of names and cities (`firstList`, `lastName`, `cityList`) are defined as arrays to be used for random generation. There's a typo in the code (`lastName` should be `lastIndex`).
- An empty array called `people` is created to store the Person objects.
- Another function called `RandomIntMax(max)` is defined. This function takes a maximum value (`max`) and uses `Math.random()` to generate a random floating-point number between 0 (inclusive) and the provided `max` (inclusive). It then uses `Math.floor()` to round down the number to the nearest integer, effectively making the maximum value inclusive.
- The

`createPeople()`

function is responsible for generating the random list of people. This function first clears the

`people`

array by setting its length to 0. Then, it retrieves the user input for the number of people using

`document.getElementById("numppl").value`

and converts it to a number using

`Number()`

. It checks if the user entered a value greater than 25 and limits it to 25 if so. A loop iterates for the desired number of people. Inside the loop:

- Three random indexes are generated using the `RandomIntMax()` function for each list (first name, last name, city). These indexes are used to access and retrieve random names and a city from the respective lists.
- A new Person object is created using the `new` keyword and the `Person` constructor, passing the retrieved random names and city.
- The newly created Person object is then stored in the `people` array at the current loop index.
- Finally, the function clears the input field for the number of people and calls the `displayPeople()` function.

- The `displayPeople()` function is responsible for displaying the generated list of people. It initializes an empty string variable (`strpeople`). It loops through the `people` array. Inside the loop, it calls the `fullname()` method of each Person object in the array to get the full name and city information. This information is then appended to the `strpeople` string along with a newline character (`\n`) and a line break tag (`<br>`).

26. Objects Inheritance

JavaScript inheritance allows you to create new classes (subclasses) that inherit properties and methods from existing classes (parent classes). This promotes code reusability and helps you organize your code in a hierarchical way.

Here's a breakdown of key concepts in JavaScript inheritance:

Classes and Inheritance:

- **Classes:** JavaScript uses the `class` keyword to define classes. A class acts as a blueprint for creating objects that share the same properties and methods.
- **Inheritance:** The `extends` keyword is used to create a subclass that inherits from a parent class. The subclass can access and use the properties and methods defined in the parent class.
- **Super Class (Parent Class):** The class from which properties and methods are inherited.
- **Sub Class (Child Class):** The class that inherits properties and methods from the super class.

Benefits of Inheritance:

- **Code Reusability:** By inheriting properties and methods from a parent class, you avoid duplicating code for common functionality.
- **Code Maintainability:** Changes made to the parent class are automatically reflected in all subclasses that inherit from it, making maintenance easier.
- **Polymorphism:** Inheritance allows for polymorphic behavior, where subclasses can override methods inherited from the parent class to provide their own implementation.

How Inheritance Works in JavaScript:

1. **Define the Parent Class:** You start by defining the parent class using the `class` keyword. This class will contain the properties and methods that you want subclasses to inherit.
2. **Create the Subclass:** To create a subclass that inherits from the parent class, you use the `extends` keyword after the class name. This establishes a connection between the subclass and the parent class.
3. **Accessing Inherited Properties and Methods:**

- A subclass automatically inherits all the properties and methods defined in the parent class.
- You can access these inherited properties and methods directly within the subclass using the `this` keyword.

1. **Overriding Inherited Methods:** Subclasses can override inherited methods from the parent class to provide their own implementation. This allows for customization of behavior specific to the subclass.

26.1 Lecture Code

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width">
  <title>Objects By Class</title>
  <style>
    .theContent {
      width: 40%;
      margin: 100px auto;
      background-color: bisque;
      padding: 40px;
      color: black;
```

```
    }
  </style>
  <script>
    class Person {
      constructor(fname, lname) {
        this.fname = fname;
        this.lname = lname;
      }

      getFullName() {
        return `${this.fname} ${this.lname}`;
      }
    }

    // ------- End Of Class --------------

    class Student extends Person {
      constructor(fname, lname, college, major) {
        super(fname, lname); // Call the parent class constructor
        this.college = college;
        this.major = major;
      }

      allinfo() {
        return `${this.fname} ${this.lname} - ${this.college}, ${this.major}`;
      }

      nameinfo() {
        return `${this.fname} ${this.lname}`;
      }
    }

    // ------- End Of Class --------------

    // Lists containing names and cities for random generation
    let firstList = ["John", "Mary", "Abe", "Jim", "Stanley", "Salley",
"Ann"];
    let lastList = ["Rodes", "Smith", "Miller", "Jones", "Stucci", "Kindreck",
"Patton"];
    let collegeList = ["MSU", "UM", "Montana Tech", "Ohio State", "Stanford",
"FVCC", "MIT"];
    let majorList = ["Comp Sci", "Engineering", "Theater", "Math", "Geology",
"AI", "Pre Med", "Pre Law"];
    let people = new Array(); // Array to store Person objects

    // Function to generate a random integer between 0 (inclusive) and max
(inclusive)
    function RandomIntMax(max) {
      // + 1 to make the max inclusive
      return Math.floor(Math.random() * (max + 1));
    }

    // Function to create a specified number of Person objects with random
names and cities
    function createPeople() {
      people.length = 0;
      let numPeople = Number(document.getElementById("numppl").value);
      if (numPeople > 25) {
        numPeople = 25;
      }
      for (let i = 0; i < numPeople; i++) {
        let rfname = firstList[RandomIntMax(firstList.length - 1)];
        let rlname = lastList[RandomIntMax(lastList.length - 1)];
        let rcollege = collegeList[RandomIntMax(collegeList.length - 1)];
        let rmajor = majorList[RandomIntMax(majorList.length - 1)];
        let xperson = new Student(rfname, rlname, rcollege, rmajor);
```

```
        people[i] = xperson;
      }
      document.getElementById("numppl").value = "";
      displayPeople();
    }

    // Function to display the full names and cities of all Person objects in
the people array
    function displayPeople() {
      let strpeople = "";
      strpeople += " <ol>";
      for (let i = 0; i < people.length; i++) {
        strpeople +=
          ` <li> ${people[i].getFullName()} <br> ${people[i].allinfo()} <br>
<br> </li>`;
      }
      strpeople += " </ol>";
      document.getElementById("ppl").innerHTML = strpeople;
    }
  </script>
</head>

<body>
  <div class="theContent">
    <h1>Inheritance</h1>
    <br><hr><br>
    <label for="numppl">Enter the number of people:</label>
    <input type="number" id="numppl" min="0" max="25" value="0"
placeholder="1-25">
    <button onclick="createPeople()">Enter</button>
    <p id="ppl"></p>
  </div>
</body>

</html>
```

26.2 Explanation

This code creates a simple web page that allows you to generate a list of random student objects. Here's a breakdown of the different sections:

HTML (Structure and Styling):

- The HTML portion of the code defines the structure of the webpage using elements like `<html>`, `<head>`, `<body>`, etc.
- It sets the character encoding (`<meta charset="utf-8">`) and viewport size (`<meta name="viewport" content="width=device-width">`) for proper display.
- The `<title>` tag sets the title of the page to "Objects By Class".
- The `<style>` section defines styles for the content area using the class `.theContent`. This class sets the width, margin, background color, padding, and text color for the content.

JavaScript (Creating Classes and Functionality):

- The `<script>` section contains the JavaScript code responsible for creating the functionality of the page.

1. Defining Classes:

- `Person` Class:

This class represents a generic person with two properties:

`fname`

(first name) and

`lname`

(last name).

- The constructor (`constructor(fname, lname)`) takes the first and last names as arguments and assigns them to the corresponding properties.
- The `getFullName()` method returns the full name of the person by concatenating the first and last names.
- `Student` Class:

This class extends the

`Person`

class, inheriting its properties and methods. It adds an additional property,

`college`

and

`major`

, to represent a student.

- The constructor (`constructor(fname, lname, college, major)`) takes the first name, last name, college, and major as arguments and assigns them to the corresponding properties. It also calls the `super(fname, lname)` constructor to initialize the inherited properties from the `Person` class.
- The `allinfo()` method returns a formatted string containing the student's full name, college, and major.
- The `nameinfo()` method is the same as the `getFullName()` method from the `Person` class, just with a different name.

2. Data and Functions:

- Several lists (`firstList`, `lastName`, etc.) contain pre-defined names and colleges for random generation.
- The `people` array is used to store the created `Student` objects.
- The `RandomIntMax(max)` function generates a random integer between 0 (inclusive) and the specified `max` value (inclusive).
- The

`createPeople()`

function:

- Clears the `people` array.
- Retrieves the number of people entered in the input field (`numppl`).
- Limits the number of people to 25 if the entered value is greater.
- Loops for the specified number of people:
  - Randomly selects a first name, last name, college, and major from the respective lists.
  - Creates a new `Student` object with the selected information.
  - Adds the created `Student` object to the `people` array.
- Clears the input field and calls the `displayPeople()` function.
- The

`displayPeople()`

function:

- Initializes an empty string `strpeople`.
- Starts an ordered list (`<ol>` tag).
- Loops through the

`people`

array:

- For each student object, it uses template literals (```) to create a formatted list item (
-

) containing the student's full name (from getFullName()), all information (from allinfo()`), and line breaks.

- Closes the ordered list and sets the inner HTML content of the element with the id "ppl" to the formatted string `strpeople`. This displays the list of students on the webpage.

Overall Functionality:

When you load the page, you'll see a heading "Inheritance", a label for entering the number of people (with a limit of 25), an input field, and a button labeled "Enter".

- Entering a number (between 0 and 25) and clicking the "Enter" button will trigger the `createPeople()` function.
- This function will generate the specified number of random student objects and store them in the `people` array.
- The `displayPeople()` function will then format and display these student objects on the page as a list.

27. Storing Web Data

All three, cookies, local storage, and session storage, are mechanisms for storing data related to web browsing. They differ in where the data is stored, persistence, accessibility, and use cases.

Cookies:

- **Storage:** On the user's device (client-side)
- **Persistence:** Varies. Cookies can have an expiration date set by the server or persist until manually cleared by the user.
- **Accessibility:** Accessible by both the client (browser) and the server (website). When a browser requests a web page from a server that previously sent a cookie, the browser includes the cookie in the request.
- **Use cases:** - Remembering user preferences (e.g., language) - Session management (e.g., shopping cart contents) - Tracking user behavior (e.g., analytics)

Local Storage:

- **Storage:** On the user's device (client-side)
- **Persistence:** Persistent until manually cleared by the user or until storage quota is reached.
- **Accessibility:** Accessible only by the client (browser) for the specific website that created it. Isolated from other websites and server-side scripts.
- **Use cases:** - Storing user preferences (e.g., location) - Application data (e.g., to-do list items)

Session Storage:

- **Storage:** On the user's device (client-side)
- **Persistence:** Temporary. Data is cleared when the browser tab or window is closed.
- **Accessibility:** Accessible only by the client (browser) for the specific website and tab/window that created it.
- **Use cases:** - Caching temporary data (e.g., form input during multi-step process)

Here's a table summarizing the key differences:

Feature	Cookies	Local Storage	Session Storage
Storage	Client-side	Client-side	Client-side
Persistence	Varies	Persistent	Session only
Accessibility	Client & Server	Client only	Client only
Use cases	Preferences, Session	Preferences, App data	Temporary data

27.1 Lecture Code

27.1.1 cookiefunctions.js Explanation

Here's a breakdown of the functions:

1. **checkCookie (demo only):** (commented out for other projects)
   - This function retrieves a cookie named "firstName" using the `getCookie` function.
   - It then constructs a welcome message using the retrieved name. (The name itself might be set elsewhere in the main code).
   - It uses console logs (for testing) and DOM manipulation to show/hide welcome messages and an input field based on whether a name is found in the cookie.
2. **setCookie(cookieName, cookieValue, expireDays):**
   - This function creates a cookie with a specified name, value, and expiry time in days.

- It creates a date object set to the current time plus the provided number of days in milliseconds.
- It formats the expiry date in the required format for cookies.
- Finally, it sets the document's cookie with the name, value, and expiry information.

3. **getCookie(cookieName):**
   - This function retrieves a cookie with a specified name.
   - It splits the entire cookie string (which can contain multiple cookies) into an array by semicolons (';').
   - It iterates through the cookie array, searching for each cookie that starts with the provided name.
   - If a matching cookie is found, it extracts the value after the name and returns it.
   - Otherwise, it returns an empty string.
4. **deleteCookie(cookieName):**
   - This function deletes a cookie with a specified name.
   - It creates a date object set to one day in the past (effectively expiring the cookie).
   - It formats the expiry date in the required format for cookies.
   - Finally, it sets the document's cookie with the name, an empty value, and the past expiry date, essentially deleting it.

```
// this is for the demo
// remove if used in other projects
window: onload = checkCookie;

function setCookie(cookieName, cookieValue, expireDays) {

    let d = new Date();
    // days to milliseconds
    d.setTime(d.getTime() + (expireDays * 24 * 60 * 60 * 1000));

    let expires = "expires=" + d.toGMTString();

    document.cookie = cookieName + "=" + cookieValue + ";" + expires;

    // dogname=fido;1232023203
}

// retreive a cookie
function getCookie(cookieName) {
    let name = cookieName + "=";

    let cookieArray = document.cookie.split(';');

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

        let c = cookieArray[i].trim();

        if (c.indexOf(name) == 0) {
            // testing only
            console.log("Cookie name is " + c.substring(name.length,
c.length));

            // actual code
            return c.substring(name.length, c.length);
        }
    }

    return "";

}

function deleteCookie(cookieName) {
    let d = new Date();
```

```
    let cookieValue = "";

    d.setTime(d.getTime() + (-1000));

    let expires = "expires=" + d.toGMTString();

    document.cookie = cookieName + "=" + cookieValue + ";" + expires;

}

// for demo only
function checkCookie() {

    // person is written in main file
    let user = getCookie("firstName");
    let message = "Welcome Back " + user + " we've missed you.";

    // for testing only
    console.log(message);

    if (user != "") {
        document.getElementById("EnterName").style.display = "none";
        document.getElementById("Welcome").style.display = "block";
        document.getElementById("p1").innerHTML = message;

    }
    else {
        document.getElementById("EnterName").style.display = "block";
        document.getElementById("Welcome").style.display = "none";
    }

}
```

27.1.2 index.html Explanation

This code snippet appears to be an HTML form that demonstrates the usage of cookies, local storage, and session storage. Here's a breakdown of the code:

HTML Structure:

- The code defines a basic HTML page with a form and buttons.
- The form has fields for first name, last name, and city.
- There are buttons to delete cookies, local storage, and session storage.
- There's also a link to a second page ("Page 2").

JavaScript Functions:

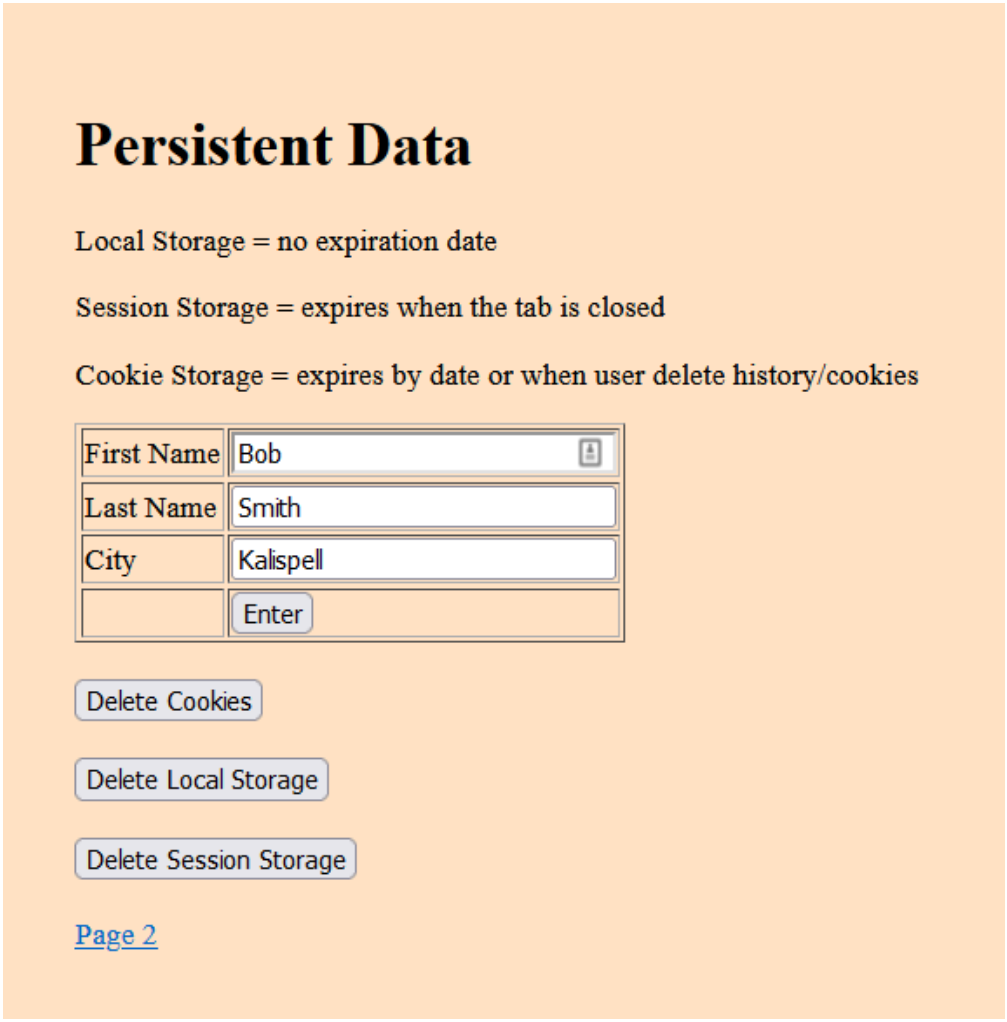
- validateForm():
  - This function is called when the form is submitted.
  - It retrieves the values from the form fields for first name, last name, and city.
  - It calls the `setCookie` function to store these values in cookies with an expiry time of 20 days.
  - It uses `localStorage.setItem` to store the values in local storage (no expiry).
  - It uses `sessionStorage.setItem` to store the values in session storage (expires when the tab closes).
  - The function returns true, allowing the form submission to proceed.
- deleteCookies():
  - This function calls the `deleteCookie` function (cookiefunctions.js`) to delete the cookies named "firstName", "lastName", and "city".
- deleteLocalStorage():
  - This function uses `localStorage.clear()` to remove all items from local storage. (Alternatively, it could use `localStorage.removeItem` to target specific items).
- deleteSessionStorage():

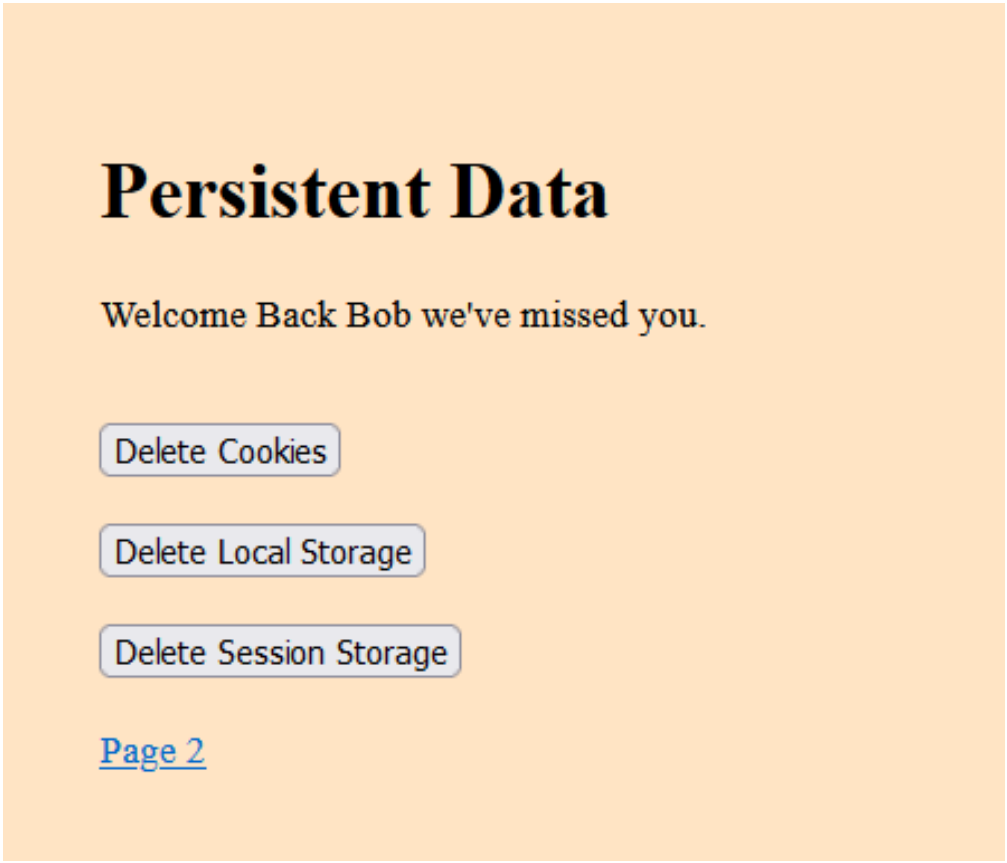
- This function uses `sessionStorage.clear()` to remove all items from session storage. (Alternatively, it could use `sessionStorage.removeItem` to target specific items).

Overall Functionality:

- When the user enters their information and submits the form, the data is stored in cookies (with expiry), local storage (no expiry), and session storage (expires with the tab).
- The buttons allow the user to delete the stored data from cookies, local storage, or session storage independently.
- There's a link to a second page ("Page 2") but it's unclear how this page might interact with the stored data.

This code provides a basic example of using these data storage mechanisms in a web page. In a real application, you'd likely want to consider the appropriate storage method based on the type of data and how long you need it to persist.






```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Page 1</title>
    <script src="cookiefunctions.js" type="text/javascript"></script>

    <style>
        .theContent {
            width: 40%;
            margin: 100px auto;
            background-color: bisque;
            padding: 40px;
            color: black;
        }
    </style>



    <script>

        function validateForm() {

            let xfname, xlname, xcity;

            xfname = document.getElementById("fname").value;
            xlname = document.getElementById("lname").value;
            xcity = document.getElementById("city").value;

            // set cookies - expires by time
            setCookie("firstName", xfname, 20);
            setCookie("lastName", xlname, 20);
            setCookie("city", xcity, 20);
```

```
            // set localstorage - no expiration date
            localStorage.setItem("firstName", xfname);
            localStorage.setItem("lastName", xlname);
            localStorage.setItem("city", xcity);

            // set set session storage
            sessionStorage.setItem("firstName", xfname);
            sessionStorage.setItem("lastName", xlname);
            sessionStorage.setItem("city", xcity);

            return true;

        }

        function deleteCookies() {
            deleteCookie("firstName");
            deleteCookie("lastName");
            deleteCookie("city");

        }

        function deleteLocalStorage() {
            // clear them all
            localStorage.clear();

            // delete one item;
            //localStorage.removeItem("firstName");
        }

        function deleteSessionStorage() {
            sessionStorage.clear();

            // delete one
            // sessionStorage.removeItem("firstName");
        }

    </script>


</head>

<body>
    <div class="theContent">
        <h1>Persistent Data</h1>
        <div id="Welcome">
            <p id="p1"></p>
        </div>

        <div id="EnterName">

            <p>Local Storage = no expiration date</p>
            <p>Session Storage = expires when the tab is closed</p>
            <p>Cookie Storage = expires by date or when user delete history/
cookies</p>


            <form onsubmit="return validateForm()" action="">

                <table border="1">
                    <tr>
                        <td><label for="fname">First Name</label></td>
                        <td><input type="text" name="fname" id="fname"
                            value="Bob" size="25"></td>
                    </tr>
                    <tr>
                        <td><label for="lname">Last Name</label></td>
```

```
                        <td><input type="text" name="lname" id="lname"
                            value="Smith" size="25"></td>
                    </tr>
                    <tr>
                        <td><label for="city">City</label></td>
                        <td><input type="text" name="city" id="city"
                            value="Kalispell" size="25"></td>
                    </tr>
                    <tr>
                        <td></td>
                        <td><input type="submit" value="Enter" size="25"></td>
                    </tr>

                </table>

            </form>


        </div>
        <br>
        <button onclick="deleteCookies()">Delete Cookies</button><br><br>
        <button onclick="deleteLocalStorage()">Delete Local Storage</
button><br><br>
        <button onclick="deleteSessionStorage()">Delete Session Storage</
button><br><br>
        <a href="page2.html">Page 2</a>
    </div>
</body>

</html>
```

27.1.3 page2.html Explanation

The second page (“Page 2”) of the same website. This page focuses on displaying the information stored in cookies, local storage, and session storage.

Here’s a breakdown of the code:

HTML Structure:

- The structure is similar to the previous page with a “theContent” class for styling.
- It has a link back to the homepage (“index.html”).
- There are sections for “Cookie Information,” “Local Storage Information,” and “Session Storage Information”.
- Each section has empty spans with IDs for displaying retrieved data.

JavaScript:

- The code retrieves the values from cookies, local storage, and session storage using the same functions (`getCookie`, `localStorage.getItem`, and `sessionStorage.getItem`) as in the previous page.
- It then finds the corresponding span elements using their IDs and sets their innerHTML content to the retrieved values.

Overall Functionality:

- This page retrieves the data previously stored on the first page and displays it in separate sections for cookies, local storage, and session storage.
- This allows the user to see what information is stored using each mechanism.

In essence, this code demonstrates how to access and display data stored on the previous page (“Page 1”) using cookies, local storage, and session storage.

Persistent Data

Home Page

Cookie Information

Bob
Smith
Kalispell

Local Storage Information

Bob
Smith
Kalispell

Session Storage Information

Bob
Smith
Kalispell

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Page 2</title>
    <script src="cookiefunctions.js" type="text/javascript"></script>

    <style>
        .theContent {
            width: 40%;
            margin: 100px auto;
            background-color: bisque;
            padding: 40px;
```

```
            color: black;
        }
    </style>
</head>

<body>
    <div class="theContent">
        <h1>Persistent Data</h1>
        <a href="index.html">Home Page</a>

        <h2>Cookie Information</h2>
        <span id="cspanfn"></span><br>
        <span id="cspanln"></span><br>
        <span id="cspancity"></span><br><br>

        <h2>Local Storage Information</h2>
        <span id="lspanfn"></span><br>
        <span id="lspanln"></span><br>
        <span id="lspancity"></span><br><br>

        <h2>Session Storage Information</h2>
        <span id="sspanfn"></span><br>
        <span id="sspanln"></span><br>
        <span id="sspancity"></span><br><br>
    </div>
    <script>
        document.getElementById("cspanfn").innerHTML = getCookie("firstName");
        document.getElementById("cspanln").innerHTML = getCookie("lastName");
        document.getElementById("cspancity").innerHTML = getCookie("city");

        document.getElementById("lspanfn").innerHTML =
            localStorage.getItem("firstName");
        document.getElementById("lspanln").innerHTML =
            localStorage.getItem("lastName");
        document.getElementById("lspancity").innerHTML =
            localStorage.getItem("city");

        document.getElementById("sspanfn").innerHTML =
            sessionStorage.getItem("firstName");
        document.getElementById("sspanln").innerHTML =
            sessionStorage.getItem("lastName");
        document.getElementById("sspancity").innerHTML =
            sessionStorage.getItem("city");
    </script>
</body>

</html>
```

28. END