Course: Web Programming

LAB 2: DISPLAY EMPLOYEE INFORMATION USING ARRAY METHODS

Author: Clim
Institution: CSIE
Date: October 5, 2025
Version: 1.03

TABLE OF CONTENTS

1. INTRODUCTION

This lab is a hands-on exercise designed for learners to practice using JavaScript array methods in a web development context. It assumes familiarity with basic web technologies and programming concepts. In this lab, you will build a simple **Employee Management System** interface and implement functionality to display and analyze employee data using array operations.

Target Audience: This lab is intended for students and junior developers with fundamental knowledge of HTML, CSS, and JavaScript, who are learning to manipulate data on the front end.

Prerequisites: Participants should be comfortable with basic HTML structure, JavaScript syntax (variables, functions, conditional statements), and have a basic understanding of arrays and objects in JavaScript. Familiarity with using a web browser's developer console and Git version control is also recommended.

Tools Used: You will use a text editor or IDE (e.g., Visual Studio Code), a modern web browser (for running and debugging the web page), and Git for version control. Ensure that Node.js or a simple local development server is available to serve the page (e.g., using VS Code Live Server extension or a similar tool).

Delivery Model: This lab can be conducted as a guided in-class activity or as a self-paced assignment. The instructions below assume a step-by-step approach, where each section builds on

the previous ones. The lab is delivered in a hands-on manner—readers are expected to follow the instructions, write code, and verify the outcomes at each step.

2. LEARNING OBJECTIVES

After completing this lab, you will be able to:

- **Apply JavaScript array methods** – Use `forEach`, `map`, `filter`, `reduce`, and `find` to manipulate and extract information from arrays of objects.
- **Implement event-driven functionality** – Trigger JavaScript functions via user interface elements (buttons) to update the Document Object Model (DOM) dynamically.
- **Perform data calculations and filtering** – Compute aggregate values (such as total salaries) and filter datasets based on criteria (such as department or other attributes).
- **Use version control in development** – Manage your code using Git and GitHub, following best practices (including meaningful commit messages following the Conventional Commits style).

These objectives are aligned with Bloom's taxonomy at the *Application* and *Analysis* levels, as you will be implementing code (applying knowledge) and examining outcomes to ensure the code works as intended.

3. Theory of JavaScript array methods

JavaScript provides several powerful **array methods** that enable developers to traverse and manipulate array data efficiently. In this section, we will review five essential array methods—`forEach`, `map`, `filter`, `reduce`, and `find`—including their syntax, typical use-cases, and how they differ from one another. We will illustrate each method with examples based on an **employee data model**, since our lab scenario involves an array of employee objects.

Assume we have an array of employee objects defined as follows, which will serve as the example dataset for discussing the array methods:

```
const employees = [
  { id: 1, name: 'John Doe', age: 30, department: 'IT', salary: 50000 },
  { id: 2, name: 'Alice Smith', age: 28, department: 'HR', salary: 45000 },
  { id: 3, name: 'Bob Johnson', age: 35, department: 'Finance', salary: 60000 }
];
```

Each object in this `employees` array represents an employee with properties such as `id`, `name`, `age`, `department`, and `salary`. We will use this array to demonstrate the different methods.

3.1. The *forEach* Method

Purpose: The `forEach` method executes a given function once for each element in an array. It is typically used to perform *side effects* (such as logging or modifying something outside the array) rather than to produce a new result. Unlike some other array methods, `forEach` **does not return a new array or any value** (it returns `undefined`). Its primary use is to simply iterate over elements.

Syntax:

```
array.forEach(function(element, index, array) {
    // ... perform action using element ...
});
```

The callback function is called with each element in turn (and optionally the index and the array itself). There is no return value from `forEach`—it simply runs the provided function for each array item.

Example (Logging each employee's name): Suppose we want to print the name of each employee to the console. We can use `forEach` on our `employees` array:

```
employees.forEach(employee => {
  console.log(employee.name);
});
```

In this example, the arrow function is executed for each `employee` in the array, and it prints the employee's name to the console. If our `employees` array contains **John Doe**, **Alice Smith**, and **Bob Johnson**, this code will output each name on a separate line in the console.

Use-Case: Use `forEach` when you need to simply *do something* with each element of an array (such as updating a counter, outputting values, or modifying an external variable) and you do not need to obtain a transformed array or result. For instance, in the context of our lab, we could use `forEach` to iterate through the employees and dynamically add their information to a webpage or accumulate values in a variable. However, if we want to produce new arrays or values from the original, other methods like `map`, `filter`, or `reduce` are more appropriate.

3.2. The *map* Method

Purpose: The `map` method creates a **new array** by applying a provided function to each element of the original array. In other words, `map` *transforms* each element and returns an array of the transformed values, without altering the original array. The returned array will have the **same number of elements** as the original, with each element being the result of the transformation of the corresponding original element.

Syntax:

```
const newArray = array.map(function(element, index, array) {
    return // ... transformed element ...
});
```

The callback function should `return` the new value for each element. The `map` method then builds and returns an array from these returned values.

Example (Extracting employee names): We can use `map` to create an array of all employee names from the `employees` data:

```
const employeeNames = employees.map(employee => employee.name);
console.log(employeeNames); // ["John Doe", "Alice Smith", "Bob Johnson"]
```

Here, the callback `employee => employee.name` takes each `employee` object and returns just the `name` property. The `map` method collects these names into a new array `employeeNames`. The original `employees` array remains unchanged. If we log `employeeNames`, we would see an array of strings containing each employee's name.

Use-Case: Use `map` when you want to derive a new array from an existing array by transforming each element. Common scenarios include converting an array of objects to an array of one of their properties (as we did with names), or applying a calculation to each number in an array (for example, increasing everyone's salary by 10% and returning a new array of updated salaries). Unlike `forEach`, which doesn't return anything, `map` is intended for producing new data sets from old ones.

Difference from forEach: It's important to note that unlike `forEach`, the `map` method **returns a new array** and does not modify the original array. If you find yourself writing a loop to transform data and push it into a new array, `map` is a cleaner and more declarative alternative.

3.3. The *filter* Method

Purpose: The `filter` method creates a **new array** containing all elements of the original array that satisfy a given condition (the "filtering" function). Elements for which the condition function returns `true` are included in the new array, while those for which it returns `false` are omitted. If no elements match, `filter` returns an empty array. Like `map`, `filter` does not modify the original array.

Syntax:

```
const filteredArray = array.filter(function(element, index, array) {
    return // true or false based on element
});
```

The callback should return a boolean (`true` to keep the element, `false` to exclude it). The result of `filter` is a new array of all elements that passed the test.

Example (Employees in the HR department): To get a list of all employees who work in the **HR** department, we can use `filter`:

```
const hrTeam = employees.filter(employee => employee.department === 'HR');
console.log(hrTeam);
```

Given our `employees` array, this code will check each employee's `department`. Only those with `employee.department === 'HR'` will be kept. In our example data, **Alice Smith** is the only employee in HR, so `hrTeam` would be an array containing just Alice's employee object. If no employee were in HR, `hrTeam` would be `[]` (an empty array). We can confirm the result by logging `hrTeam`, which should output something like:

```
[ { id: 2, name: "Alice Smith", age: 28, department: "HR", salary: 45000 } ]
```

Use-Case: Use `filter` when you need to derive a subset of elements from an array that meet certain criteria. This is common for tasks like finding all records that match a query (e.g., all employees older than 30, all items that are in stock, etc.). In the lab scenario, we use `filter` to find all employees in a particular department (HR) to display only those on the webpage.

Behavior: It's worth noting that `filter` will always iterate through the entire array to test each element (unlike the `find` method, which stops after finding one match, as we will see). The result array from `filter` can have anywhere from 0 up to the full number of elements of the original array, depending on how many elements satisfy the condition.

3.4. The *reduce* Method

Purpose: The `reduce` method iterates through an array and "reduces" it to a single value. It does this by applying a **reducer function** you provide, which accumulates a result as it processes each element. Common uses of `reduce` include summing up numbers, calculating products, or combining array elements into a single aggregated result (like an object or string).

Syntax:

```
const result = array.reduce(function(accumulator, element, index, array) {
    // ... update accumulator using element ...
```

```
    return accumulator;
}, initialAccumulatorValue);
```

The reducer function receives an *accumulator* (which carries over a running result) and the current element. It must return the updated accumulator after processing the element. The `reduce` call also takes an initial value for the accumulator (e.g., `0` for summing numbers, an empty string for concatenation, or an empty object for building an object).

Example (Total of all salaries): To calculate the **sum of salaries** of all employees, we can use `reduce` with an accumulator that adds up salaries:

```
const totalSalary = employees.reduce((sum, employee) => sum + employee.salary, 0);
console.log(`Total salary: $${totalSalary}`);
```

In this code, we pass a reducer function `(sum, employee) => sum + employee.salary` and an initial value of `0` for the accumulator `sum`. The reducer goes through each `employee` in the array, adding the `employee.salary` to the running `sum`. After iterating through all employees, `totalSalary` will hold the final accumulated sum. For our example data, this would yield:

```
Total salary: $155000
```

since 50000 + 45000 + 60000 = 155000.

Use-Case: Use `reduce` when you need to compute a single value (or a single object/structure) from an array. In practice, reduce is very powerful and can be used to do things like summing numbers, merging arrays of arrays, counting occurrences of values, etc. In our lab's context, we use `reduce` to compute the total of all salaries in the employees array. Another example could be using `reduce` to find the maximum or minimum value in an array by comparing and accumulating.

Important Differences: Unlike `map` and `filter`, which return arrays, `reduce` returns a single value. The shape of that value depends on what you initialize and how you update the accumulator. Also note that because `reduce` can do many things, it's important to use it when appropriate – if you specifically need an array or subset, `map` or `filter` might be clearer. But for summing or accumulating results, `reduce` is ideal.

3.5. The *find* Method

Purpose: The `find` method returns **the first element** in an array that satisfies a given condition. If an element meeting the condition is found, `find` immediately returns that element (and does not check the remaining elements). If no matching element is found, `find` returns `undefined`. This method is useful when you want to locate a single object or value in an array based on a criterion, rather than all of them.

Syntax:

```
const foundElement = array.find(function(element, index, array) {
    return // true if element matches criteria
});
```

The callback should return `true` for the element that you want to find. `find` will return the first element for which the callback returns `true` and stop processing further.

Example (Find an employee by a condition): Suppose we want to find the first employee who is older than 30 years:

```
const seniorEmployee = employees.find(employee => employee.age > 30);
console.log(seniorEmployee);
```

This code will scan through the `employees` array and return the first employee object whose `age` is greater than 30. In our dataset, the employees' ages are 30, 28, and 35. The first employee who satisfies `age > 30` is **Bob Johnson** (age 35), who is the third element in the array. The variable `seniorEmployee` will be set to Bob's object:

```
{ id: 3, name: 'Bob Johnson', age: 35, department: 'Finance', salary: 60000 }
```

If we `console.log(seniorEmployee)`, we'll see Bob's details. If no employee older than 30 existed, the result would be `undefined`.

Another common use in our context is to find an employee by a unique identifier. For example, to find the employee with `id === 2`:

```
const emp = employees.find(employee => employee.id === 2);
// This would find Alice Smith's object (id 2) if it exists.
```

Use-Case: Use `find` when you need to locate a **single** item in an array. For instance, finding a user by ID, finding an order with a specific number, or any scenario where exactly one match (or just the first match) is needed. In the lab, we will use `find` to search for an employee by a given ID and display that employee's details.

Difference from filter: It's important to highlight that `find` and `filter` are similar in that they both test elements against a condition, but they differ in their output and iteration: - `filter` returns **all** elements that match the condition (in an array, possibly empty), and will always iterate through the entire array (since it must check every element to collect all matches). - `find` returns **the first** element that matches (as a single value, not an array) and stops as soon as it finds one, potentially not iterating over the rest once a match is found. If you only need one match (for example, an ID lookup which should correspond to a single entry), `find` is more efficient and semantically appropriate than `filter`.

3.6. What We've Learned (Summary)

In summary, JavaScript's array methods provide a declarative way to handle common data operations on arrays: - **forEach** – Iterates through an array executing a function on each element. Use for side effects; *does not return* a new array or value. - **map** – Transforms each array element and returns a new array of equal length containing the transformed values. Use when you want an output array. - **filter** – Tests each element with a condition and returns a new array of all elements that pass the test. Use to get a *subset* of elements. - **reduce** – Combines all elements into a single accumulated result (e.g., sum, product, aggregated object). Use for computing a single value from an array. - **find** – Searches for the *first* element that meets a condition and returns it (or `undefined` if not found). Use when looking for one specific item.

These methods help make code more concise and readable compared to traditional `for` loops. In the context of this lab, you will leverage these methods to implement functionality such as listing all employees, calculating total salaries, filtering employees by department, and finding an employee by ID in an interactive web page.

4. PRACTICAL LAB

In this practical portion, we will build a simple web interface for an Employee Management System and implement the functionality using the array methods discussed above. The lab instructions are organized into steps. Each step has a specific objective and a set of tasks (instructions). We will provide code snippets (in **CodeBlue** style for clarity) and command-line instructions (in **CommandBlack** style) where appropriate. After following each step, you should test the outcome to

verify that the functionality works as expected. Extensions and additional explorations are suggested along the way.

4.1. Step 1: Setting Up the Environment

Objective: Prepare the project workspace by creating the necessary files and basic structure for the employee management web page.

Instructions:

1. **Clone the repository.** Open a terminal in your development environment and navigate to the directory where you keep your projects. Clone the lab repository (or your course repository) by running the following Git command (replace `<repository-url>` with the URL provided by your instructor or the course setup):

   ```
   git clone <repository-url>
   ```

   This command creates a local copy of the repository on your machine. If you already have a main repository for the course, ensure you are working in the appropriate folder.

1. **Navigate to the project folder.** After cloning, change your working directory to the repository's folder. For example, if the repository folder is named `WebProgrammingLabs`:

   ```
   cd WebProgrammingLabs
   ```

   If you have a specific folder for this lab, navigate there. Otherwise, you can create a new folder for this lab within the repository.

1. **Create project files.** Inside your project (or lab) directory, create a new folder for this lab's files (for example, `employeeDetails`). Within that folder, create two files: an HTML file and a JavaScript file. Name them `employee_details.html` and `employee_details.js` respectively. The HTML file will contain the webpage structure, and the JS file will contain the script to manipulate the page.

2. **Set up the HTML structure.** Open `employee_details.html` in your text editor and add a basic HTML5 template. Include a title and a script reference to the JS file. For this lab, we need a simple interface with a heading and a set of buttons to trigger our JavaScript functions, as well as a placeholder `<div>` to display results. Use the following code as a starting point for your HTML file:

   ```
   <!DOCTYPE html>
   <html>
   <head>
     <meta charset="UTF-8" />
     <title>Employee Management System</title>
   </head>
   <body>
     <h1>Employee Management System</h1>
     <div>
       <button onclick="displayEmployees()">Display Employees</button>
       <button onclick="calculateTotalSalaries()">Calculate Total Salaries</button>
       <button onclick="displayHREmployees()">Display HR Employees</button>
       <button onclick="findEmployeeById(2)">Find Employee by ID 2</button>
     </div>
     <div id="employeesDetails"></div>
     <script src="./employee_details.js"></script>
   </body>
   </html>
   ```

This HTML code sets up: - A heading (<h1>) for the page title. - A series of four buttons inside a that, when clicked, will call JavaScript functions (which we will implement in the next steps) to: - Display all employees. - Calculate the total of all salaries. - Display only the employees in the HR department. - Find and display the employee with ID 2. - An empty which will serve as a container to inject the output (employee information) dynamically. - A script tag at the end to include the employee_details.js file so that our functions are loaded.

ⓘ **Note:** After typing or pasting the HTML code, save the file. It's important to save whenever you make changes, so that when you load the page in a browser, you see the updated content.

At this stage, if you open employee_details.html in a web browser, you should see a simple page with the title "Employee Management System" and four buttons. However, clicking the buttons *right now* will do nothing because we haven't implemented the JavaScript functions yet. In the next step, we will define the data and those functions in employee_details.js. Keep the browser open or ready to refresh as we add functionality.

4.2. Step 2: Defining Variables and Functions

Objective: Implement the data model and core functions in JavaScript to handle displaying and processing employee information when the buttons are clicked.

Instructions: Open the employee_details.js file in your editor. We will add the following in sequence: an array to store employee data, and four functions corresponding to the buttons (plus one helper function for finding by ID). Add each piece of code below to the JavaScript file:

1. **Initialize the employees array.** Define a constant array employees that holds a list of employee objects. Each employee should have properties such as id, name, age, department, and salary. For example:

```
const employees = [
  { id: 1, name: 'John Doe', age: 30, department: 'IT', salary: 50000 },
  { id: 2, name: 'Alice Smith', age: 28, department: 'HR', salary: 45000 },
  { id: 3, name: 'Bob Johnson', age: 35, department: 'Finance', salary: 60000 }
  // ... you can add more employee objects as needed
];
```

   This array serves as our data model. Feel free to add more sample employees to make the output more interesting. For instance, you could add another IT or HR employee. Make sure each object has a unique id.

2. **Create the displayEmployees() function.** This function will generate a list of all employees and display it on the webpage. We can utilize the map method to create an array of HTML strings for each employee and then join them. Add the following function definition:

```
// Function to display all employees on the page
function displayEmployees() {
  const allEmployeesHTML = employees
    .map(employee => `<p>${employee.id}: ${employee.name} - ${employee.department} -
$${employee.salary}</p>`)
    .join('');
  document.getElementById('employeesDetails').innerHTML = allEmployeesHTML;
}
```

   Explanation: The .map() call goes through each employee in the employees array and returns a string of HTML for that employee's info. For example, an element might look like:

   " 1: John Doe - IT - $50000 ".

We include the employee's id, name, department, and salary in a readable format. The `join('')` at the end concatenates all these `<p>` strings into one long string without any separator, so that when inserted into the page it will display as separate paragraphs. Finally, we find the `<div>` with id `"employeesDetails"` in the HTML and set its `innerHTML` to this combined string, causing all employee details to appear in that container. Each time this function is called (by clicking the "Display Employees" button), it will refresh the contents of the `employeesDetails` `<div>` with the full list of employees.

3. **Create the `calculateTotalSalaries()` function.** This function will compute the sum of all employee salaries and show the result. We'll use the `reduce` method to sum up salaries, and then display the total in an alert dialog:

```
// Function to calculate and alert the total of all salaries
function calculateTotalSalaries() {
  const totalSalaries = employees.reduce((acc, employee) => acc + employee.salary, 0);
  alert(`Total Salaries: $${totalSalaries}`);
}
```

Explanation: Here we use `employees.reduce(...)` with an initial accumulator value of `0`. The reducer adds each `employee.salary` to the running total `acc`. The final result is stored in `totalSalaries`. We then use `alert()` to show a pop-up message displaying the total. For example, given our current three employees, the alert would say "Total Salaries: $155000". Using an alert is a simple way to show output; it pauses execution until the user dismisses it. (Alternatively, we could display this result on the page by inserting into the DOM, but for demonstration purposes an alert is fine.)

4. **Create the `displayHREmployees()` function.** This will filter the employees to those in the HR department and display them on the page (similar to `displayEmployees()` but with a subset). Add the following:

```
// Function to display only employees from the HR department
function displayHREmployees() {
  const hrEmployees = employees.filter(employee => employee.department === 'HR');
  const hrEmployeesHTML = hrEmployees
    .map(employee => `<p>${employee.id}: ${employee.name} - ${employee.department} -
$${employee.salary}</p>`)
    .join('');
  document.getElementById('employeesDetails').innerHTML = hrEmployeesHTML;
}
```

Explanation: We first use `filter` to get a new array `hrEmployees` containing only those employees whose `department` is `'HR'`. Then, similar to `displayEmployees()`, we use `map` on that filtered list to create HTML paragraphs for each remaining employee, and join them into one string. Finally, we update the `innerHTML` of the same `employeesDetails` `<div>` to show only the HR employees. If there are no HR employees in the list, `hrEmployees` would be an empty array and nothing (blank) will be displayed in the container.

5. **Create the `findEmployeeById(employeeId)` function.** This function will search for a single employee by a given ID and display that employee's information. We will use the `find` method to locate the employee. Add the following code:

```
// Function to find an employee by ID and display their details
function findEmployeeById(employeeId) {
  const foundEmployee = employees.find(employee => employee.id === employeeId);
  if (foundEmployee) {
    // If an employee with the given ID is found, display their info
    document.getElementById('employeesDetails').innerHTML =
      `<p>${foundEmployee.id}: ${foundEmployee.name} - ${foundEmployee.department} -
$${foundEmployee.salary}</p>`;
```

```
  } else {
    // If no employee is found with that ID, display a message
    document.getElementById('employeesDetails').innerHTML =
      `<p><em>No employee found with ID ${employeeId}.</em></p>`;
  }
}
```

Explanation: The function `findEmployeeById` accepts a parameter `employeeId`. It uses `employees.find(...)` to search for an employee object whose `id` matches the given `employeeId`. If such an employee exists, `foundEmployee` will hold that object; otherwise it will be `undefined`. We then use an `if` check: - If `foundEmployee` is truthy (found), we update the `employeesDetails` `<div>` to display that single employee's information (similarly formatted in a `<p>` tag). - If `foundEmployee` is falsy (`undefined`, meaning no match), we update the container with a message indicating that no employee was found for that ID. The message is italicized using `<em>` tags for clarity.

This function allows the button "Find Employee by ID 2" (in our HTML) to display the details of the employee with ID 2 when clicked. In our dataset, ID 2 corresponds to Alice Smith, so it should show Alice's info. If you change the argument (say, to an ID that doesn't exist in the array), it will show the "No employee found" message.

With these definitions in `employee_details.js`, save the file. You have now implemented the core logic for all four buttons: - `displayEmployees()` – lists all employees. - `calculateTotalSalaries()` – calculates total payroll. - `displayHREmployees()` – filters and shows HR department members. - `findEmployeeById(id)` – finds an employee by ID and shows them.

Make sure that your HTML file has the script reference `<script src="./employee_details.js"></script>` included (as provided in Step 1). The functions need to be loaded for the buttons to call them.

4.3. Step 3: Running and Testing the Application

Objective: Launch the HTML page in a web browser and verify that each button triggers the intended functionality, producing the expected output.

Instructions:

1. **Open the HTML file in a browser.** If you are using Visual Studio Code with the Live Server extension, you can right-click on `employee_details.html` and choose "Open with Live Server". This will start a local development server (often at port 5500) and open the page in your default browser. If not using Live Server, you can open a terminal and use a simple server (for example, via Node.js you could run `npx serve` or `python -m http.server 3000` in the directory) or just double-click the HTML file to open it directly (though some features work best with a server). The goal is to have the page served at an address like `http://127.0.0.1:5500/employeeDetails/employee_details.html` (if using Live Server) or `http://localhost:3000/employee_details.html`, etc., depending on your setup.

2. **Verify the interface loads.** You should see the **Employee Management System** page with the four buttons visible. At this point, the `employeesDetails` area below the buttons should be empty (since we haven't clicked anything yet).

3. **Test the "Display Employees" button.** Click the **Display Employees** button. This should trigger the `displayEmployees()` function, which in turn uses `map` to list all employees in the `employeesDetails` `<div>`. Verify that you now see a list of all employees displayed, for example:
   `1: John Doe – IT – $50000`

```
2: Alice Smith – HR – $45000
3: Bob Johnson – Finance – $60000
```
Each employee should appear on a separate line (because we wrapped each in a <p> tag). If you added more employee objects to the array, they should all be listed here as well.

4. **Test the "Calculate Total Salaries" button.** Click the **Calculate Total Salaries** button. This triggers the `calculateTotalSalaries()` function, which uses `reduce`. An alert pop-up should appear on the page showing the total salary figure. For the given sample data, it should say **"Total Salaries: $155000"**. Click "OK" or close the alert to continue. If you added or changed salaries, verify that the sum reflects those changes.

5. **Test the "Display HR Employees" button.** Click the **Display HR Employees** button. This triggers `displayHREmployees()`. The `employeesDetails` section should now update to show only the employees whose department is HR. Based on our data, you should see only Alice Smith's information (since she's the only one in HR). The format will be the same as the full list, but filtered. If you had multiple HR employees, they all would appear here. If no employees are in HR, the section would simply become blank (empty string) as no paragraphs would be generated.

6. **Test the "Find Employee by ID 2" button.** Click the **Find Employee by ID 2** button. This calls `findEmployeeById(2)`. The `employeesDetails` section should now show only the employee with ID 2. In our array, that is Alice Smith, so it should display Alice's details (just as one line). To further test this function, you might temporarily modify the HTML to pass a different ID, or call `findEmployeeById(1)` from the browser console, for example, to see it display John Doe. If an ID is not found, the message "No employee found with ID X." will be shown in the page.

All four functions should now be working. Try clicking the buttons in different orders as well – for example, display all, then display HR (the HR output should replace the full list in the same), then display all again, etc., to ensure the content updates dynamically.

ⓘ **Note:** After making any changes to your JavaScript code, save the file and refresh the page in the browser to test the changes. If you're using Live Server, it might auto-reload when you save. The page is static, so there's no need to restart the server for changes to take effect—simply refreshing the browser will load the new script.

4.4. Step 4: Using Git for Version Control

Objective: Commit your changes and push the completed lab project to your remote GitHub repository. This ensures your work is backed up and can be reviewed or graded.

By now, you have a functioning mini-application. It's a good practice (and often a requirement in courses) to use Git to manage your code changes. Let's go through the basic Git commands to save (commit) our work and upload (push) it to GitHub.

Instructions:

1. **Stage your changes:** In the terminal, make sure you are inside your project repository directory. Stage all the files you created or modified (HTML, JS, etc.) using:
```
git add -A
```
This command tells Git to add *all changes* (new files, modified files, deleted files) to the staging area, preparing them to be committed. Alternatively, you could use `git add .` which does the same for the current directory.

2. **Commit your changes:** Now commit the staged changes to your local repository with a descriptive commit message:

```
git commit -m "feat: implement lab 1 array methods functionality"
```

The text after the `-m` in quotes is the commit message. Here we used a prefix "`feat:`" following the Conventional Commits style to indicate a new feature, and a brief description. You might write a different message, but make sure it reflects what you did (e.g., "Add employee array and functions to display and filter employee data").

ⓘ **Note:** If this is your first time committing in this repository, Git may prompt you to configure your identity. You might see a message about setting your user.email and user.name. In that case, run the following commands with your details:

```
git config --global user.email "you@example.com"
git config --global user.name "Your Name"
```

Replace the email and name with your own. This sets up your Git identity for commits. After setting this, run the commit command again.

3. **Push to GitHub:** Now that your changes are committed locally, send them to the remote repository on GitHub:
   ```
   git push origin main
   ```

   This assumes the main branch is named "main" (which is the default on most new repositories; in older repositories it might be "master", so adjust if necessary). The `origin` is the default name for the remote repository. This command will transfer your commit(s) to GitHub. You may be prompted for your GitHub username and password (or a token) if not already authenticated.

Once the push is complete, your code is now on GitHub. You can visit the repository page on GitHub to verify that the new files (`employee_details.html` and `employee_details.js`) are present and the commit message appears in the history.

By following these steps, you've saved a snapshot of your lab work. This is important for both collaboration and grading. In a classroom setting, your instructor can now see your code on GitHub. In a professional setting, you or others can later review the commit history to understand changes.

4.5. Extension (Optional)

For additional practice and to deepen your understanding, you can extend the functionality of the Employee Management System with the following challenge tasks:

1. **Add a new property and filter:** Introduce a new key–value pair for each employee in the `employees` array, for example a `specialization` field (e.g., programming language or domain expertise). Populate this field for each employee (e.g., `'JavaScript'`, `'Python'`, `'HR Management'`, etc.). Then, add a new button on the HTML page (e.g., "Display JS Specialists") that, when clicked, will filter employees based on that specialization.

2. **Button and function for specialization:** After adding the above property, implement a function (similar to `displayHREmployees()`) that filters and displays employees who have a certain specialization, for instance all employees whose specialization is **"JavaScript"**. You can decide to filter by a hardcoded value or make the function more flexible (perhaps by passing a parameter).

3. **Test the new feature:** Ensure that clicking your new button properly shows the filtered list of employees on the page. For example, if you added a "Display JS Specialists" button, clicking it should show only those employees from the array whose specialization is "JavaScript."

4. **UX improvement - clear output:** You might also add a button to clear the displayed results or a message indicating when no employees match a filter (similar to how the find function shows a message when not found). This can make the interface more user-friendly.

5. **Expected output example:** If you added a specialization field and a button to display, say, "JavaScript Developers," clicking that button should list only the employees with specialization "JavaScript". If none have that specialization, the displayed area would be empty or show a "no employees found" message.

These extensions are optional and meant for practice. They encourage you to use array methods (`filter`, `find`) in slightly different ways and to manipulate the DOM dynamically. Completing them will further solidify your understanding of handling array data and events in JavaScript.

4.6. What We've Learned (Summary)

In this lab, we accomplished the following key tasks:

1. **Environment setup:** Created an HTML and JavaScript file and structured a simple web interface with buttons. The HTML includes a container `<div>` to display dynamic content without reloading the page.
2. **Data and function implementation:** Defined a JavaScript array of employee objects and wrote functions leveraging array methods (`map`, `reduce`, `filter`, `find`) to manipulate and display that data based on user input (button clicks).
3. **Dynamic output and testing:** Injected content into the DOM to show results (all employees, filtered lists, or single finds) and used alerts for aggregate feedback. We tested each feature manually in the browser to ensure the expected output appears upon interacting with the interface.
4. **Version control usage:** Practiced using Git commands (`add`, `commit`, `push`) to save our work and upload it to a remote repository, reinforcing good development practices like frequent commits with meaningful messages (following Conventional Commits style).

By completing this lab, you not only wrote a functional mini-app but also exercised reading and writing operations on arrays and connected those operations to a user interface. This is a fundamental skill in web development: responding to user actions by processing data and updating the view.

5. FINAL QUIZ

Test your understanding of the concepts covered in this lab by answering the following multiple-choice questions. Each question is followed by its answer and an explanation.

Q1. Which JavaScript array method should you use to create a new array by transforming each element of an existing array?

a. `forEach`
b. `map`
c. `filter`
d. `reduce`

Answer: b. `map`.
Explanation: The `map` method is designed to apply a transformation to each element and produce a new array of the same length with the transformed values. `forEach` executes a function for each element but does not return a new array. `filter` returns only certain

elements (not necessarily transforming them, just selecting them), and `reduce` collapses the array into a single value.

Q2. What is the return value of the `forEach` method in JavaScript?

a. The original array
b. A new array of modified elements
c. `undefined`
d. The number of elements iterated

Answer: c. **`undefined`**.
Explanation: The `forEach` method does not return any value (technically it returns `undefined`). It simply iterates over the array and allows you to execute code for each element. Options (a) and (b) are behaviors of other methods (`map` returns a new array; many methods return something, but `forEach` specifically returns nothing useful). Option (d) is incorrect because `forEach` doesn't count or return the number of elements (though you could manually count in a closure if needed, that's not the return value of `forEach` itself).

Q3. Which array method would you use to compute a single value (such as a sum or product) from all elements of an array?

a. `filter`
b. `map`
c. `find`
d. `reduce`

Answer: d. **`reduce`**.
Explanation: The `reduce` method is explicitly intended to take an array and *reduce* it to a single accumulated value using a reducer function. `filter` returns a subset array, `map` returns an array of equal length after transformation, and `find` returns a single element (but specifically the first matching element, not an accumulated result of all elements). Only `reduce` combines all elements into one value.

Q4. If you need to find the first employee in an array who meets a certain condition (for example, the first employee in the IT department), which method should you use?

a. `find`
b. `filter`
c. `forEach`
d. `map`

Answer: a. **`find`**.
Explanation: The `find` method will return the first element that satisfies the condition. This is perfect for retrieving a single object (e.g., the first IT employee). While `filter` could retrieve *all* IT employees, that would be unnecessary if you only want the first one. `forEach` and `map` are not meant for searching; `forEach` just iterates, and `map` transforms elements. So `find` is the most efficient choice here.

Q5. Which of the following statements about the `filter` method is true?

a. It stops iterating over the array once the first matching element is found.
b. It returns `undefined` if no array elements match the condition.

c. It returns a new array containing all elements that satisfy the given condition.
d. It modifies the original array by removing elements that do not match.

Answer: c. **It returns a new array containing all elements that satisfy the given condition.**
Explanation: The `filter` method will check every element of the array and include it in the result if the condition returns true. Option (a) describes the behavior of `find`, not `filter` – `filter` will not stop early; it goes through every element. Option (b) is incorrect because if no elements match, `filter` returns an empty array ([ ]), not `undefined`. Option (d) is also incorrect – `filter` does not alter the original array at all; it produces a new array and leaves the source intact.

6. HOMEWORK

For homework, you will expand the functionality of the Employee Management System developed in this lab and practice deploying it. This assignment is due by **next weekend (October 19, 2025)**. Please complete the following tasks:

- **Extend the data model:** Add two new properties to each employee object in the `employees` array: an **email** address and a **hiringDate**. The hiring date can be represented as a string or a date object (e.g., `"2021-07-15"` or a JavaScript `Date` object). Ensure each entry in the array is updated with realistic values for these new fields.

- **Display new fields:** Update the display functions (or create new ones) to include the email and hiring date when showing employee details. For instance, the full display of an employee could now show **Name – Department – Salary – Email – Hiring Date**.

- **Compound filtering:** Implement a feature that allows filtering employees based on *multiple criteria simultaneously*. For example, create a function (and corresponding UI, like a form or additional buttons) to filter employees who belong to a certain department **and** were hired after a certain date, or perhaps who match two conditions like being in IT department and having a salary above 50,000, etc. You can choose the specific criteria for compound filtering based on what data you have (for instance, department and hiringDate make a good combination to practice date comparison).

- **Run at localhost:3000:** Ensure your application can run on port 3000 on your local machine. If you are using a simple static server, configure it to serve on port 3000. If you haven't done so, you might create a basic Node.js/Express setup or use a library to serve the page at `http://localhost:3000`. This is to practice setting up a custom local server environment.

- **Deploy to GitHub repository:** Create a new folder in your GitHub repository for this homework (for example, under your main repo, create a directory `sem2/HWclim` if `sem2` is a semester folder and `HWclim` denotes this homework). Place your HTML, JS, and any other needed files in that folder. Ensure that your project has a **README** file explaining what the application does and how to run it, and include a LICENSE file if one is required by your course or preferred (you can use an open-source license template or a simple statement of rights).

- **Follow Git/GitHub best practices:** As you work on the homework, commit your changes in stages with clear messages (e.g., `"feat: add email and hiringDate to employee data"`, `"feat: implement filter by department and hiring date"`, `"docs: add usage instructions to README"`). Push your commits to the GitHub repository so that the instructor can see your progress. Ultimately, the `HWclim` folder in your repo should contain the final application code, and your commit history should reflect the development process.

Submission: By the due date, ensure that your GitHub repository is up to date with the homework changes. You may need to inform your instructor or submit the repository link according to course guidelines.

This homework will reinforce your skills with arrays (particularly adding new data and filtering on multiple conditions) and give you practice in preparing a small project for deployment (with documentation and proper structure on GitHub).

7. Evaluation and version control guidelines

Your lab and homework will be evaluated based on the following criteria. Make sure to review this rubric to understand how your work will be assessed:

Evaluation Rubric:

Criteria	Description	Points
Correct Implementation	All required functionalities are implemented correctly (array methods work as expected for display, filter, reduce, find, etc.). The web page responds to button clicks with the correct output.	50%
Extended Features	Homework extensions (email, hiringDate, compound filtering) are implemented and functioning. The application runs on localhost:3000 as required.	30%
Code Quality & Style	Code is well-organized and readable. Uses meaningful variable/function names. Follows best practices covered in class. No obvious errors or sloppy code segments. Comments or docstrings are present where necessary to explain complex logic.	20%
Documentation	README file is present and clearly explains how to run the application and describes its features. The lab document (this one) is properly formatted. The project includes a LICENSE if required.	10%
Git Usage	Changes are committed to Git with appropriate frequency. Commit messages are descriptive and follow a logical format (using Conventional Commits style – e.g., prefixing with *feat*, *fix*, *docs*, etc.). The repository is structured correctly (homework in the designated folder).	20%
Timely Submission	The project was submitted (pushed to GitHub) by the deadline. Late submissions may be penalized according to course policy.	*(deduction)*

Total Points: 100% (points correspond to an approximate weighting; your instructor may adjust scoring or convert to a percentage/grade accordingly).

Git/GitHub Submission Guidelines:

- Ensure that your final code resides in the correct location in the GitHub repository (as specified, e.g., under `/sem2/HWclim` for the homework).
- Use **Conventional Commits** for your commit messages during development. For example:
- `feat: add filter by department and hiring date`
- `fix: correct logic for calculating total salary`
- `docs: update README with setup instructions`

These prefixes (*feat*, *fix*, *docs*, etc.) make it clear what each commit represents. They help instructors and collaborators quickly understand the nature of changes. - Commit often. It's better to have multiple small commits than one monolithic commit at the end. This demonstrates your development

process and makes it easier to track where issues might have arisen. - Push your commits to GitHub regularly. Don’t wait until the last minute to push all your work; incremental pushes serve as a backup and also show progressive work (which can be helpful if any issues need to be discussed). - Finally, double-check the GitHub repository in a web browser. Make sure all intended files (HTML, JS, README, etc.) are visible and that the formatting of your documentation is correct. A well-structured repository is easier to navigate and grade.

By adhering to these guidelines and meeting the rubric criteria, you will set yourself up for success on this lab and the subsequent homework. Good luck, and happy coding!