BVC COLLEGE OF ENGINEERING

Department of Computer Science and Engineering

LABORATORY MANUAL

MEAN STACK TECHNOLOGIES - MODULE I (HTML 5, JAVASCRIPT, EXPRESS.JS, NODE.JS AND TYPESCRIPT) (Skill Oriented Course)
III Year – II Semester - 2024-25

Course Outcomes (COs): At the end of the Course, Student will be able to:

- **CO1:** Develop professional web pages of an application using HTML elements like lists, navigations, tables, various form elements, embedded media which includes images, audio, video and CSS Styles.
- **CO2:** Utilize JavaScript for developing interactive HTML web pages and validate form data.
- **CO3:** Build a basic web server using Node.js and also working with Node Package Manager (NPM).
- **CO4:** Build a web server using Express.js
- **CO5:** Make use of Typescript to optimize JavaScript code by using the concept of strict type checking.

---

1. General Lab Instructions

1. Students are expected to be punctual for the lab sessions and should be in formal dress code.
2. Students should come prepared for the experiment by thoroughly reading the relevant theory and procedure in advance.
3. Maintain a neat and professional lab record book, documenting all experiments with observations and results.
4. Follow the instructions given by the faculty carefully and perform the experiments diligently.
5. Ensure proper shutdown of all equipment after the completion of each lab session.
6. In case of any difficulty or need for assistance, students should promptly consult the faculty.
7. Strictly adhere to safety precautions while working in the lab.
8. Any damage to equipment due to negligence or mishandling will be the responsibility of the student.
9. Regular attendance and active participation in lab sessions are mandatory.
10. Maintain discipline and a conducive learning environment in the laboratory.

Table of Contents

---

2. HTML 5 Exercises

Tools Required:

- Web Browser (Chrome, Firefox, etc.)
- Text Editor (VS Code, Sublime Text, Atom, Notepad++, etc.)

Experiment 1.a: Metadata Element in HTML

Course Name: HTML5 - The Language **Module Name:** Case-insensitivity, Platform-independency, DOCTYPE Declaration, Types of Elements, HTML Elements - Attributes, Metadata Element

Objective: To understand and implement metadata in HTML to provide information about the HTML document.

Procedure:

1. Create a new HTML file named `Homepage.html` for IEKart's Shopping Application.
2. Inside the `<head>` section of `Homepage.html`, add the `<meta>` element.
3. Use the `name` attribute set to "description" and the `content` attribute to provide the description as: "IEKart's is an online shopping website that sells goods in retail. This company deals with various categories like Electronics, Clothing, Accessories etc."

4. Open `Homepage.html` in a web browser and inspect the page source to verify the metadata.
5. Refer to the provided link for detailed information about Metadata elements: https://infyspringboard.onwingspan.com/web/en/viewer/web-module/lex_28320667711144660000_shared?collectionId=lex_17739732834840810000_shared&collectionType=Course
6.

Example Code Snippet (Homepage.html):

```
HTML
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <meta name="description" content="IEKart's is an online shopping website that sells goods in
retail. This company deals with various categories like Electronics, Clothing, Accessories etc.">
   <title>IEKart Homepage</title>
</head>
<body>
   <h1>Welcome to IEKart!</h1>
   </body>
</html>
```

Expected Output:

- The `Homepage.html` should contain the metadata description when viewed in the page source.

Observations:

Experiment 1.b: Sectioning Elements in HTML

Course Name: HTML5 - The Language **Module Name:** Sectioning Elements

Objective: To utilize HTML5 sectioning elements to structure the content of a webpage semantically.

Procedure:

1. Enhance the `Homepage.html` of IEKart's Shopping Application created in Experiment 1.a.
2. Add appropriate HTML5 sectioning elements like `<header>`, `<nav>`, `<main>`, `<article>`, `<section>`, `<footer>`, and `<aside>` to structure the content of the homepage (e.g., header for website title and navigation, main for the primary content, footer for copyright information).
3.
4. Populate these sectioning elements with sample content relevant to an online shopping application.
5.
6. View `Homepage.html` in a web browser to observe the structural changes.
7. Refer to the provided link for detailed information about Sectioning Elements: https://infyspringboard.onwingspan.com/web/en/viewer/web-module/lex_6372291347110857000_shared?collectionId=lex_17739732834840810000_shared&collectionType=Course

Example Code Snippet (Homepage.html):

```
HTML
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta name="description" content="IEKart's is an online shopping website...">
    <title>IEKart Homepage</title>
</head>
<body>
    <header>
        <h1>IEKart</h1>
        <nav>
            <ul>
                <li><a href="#">Home</a></li>
                <li><a href="#">Products</a></li>
                <li><a href="#">About Us</a></li>
                <li><a href="#">Contact</a></li>
            </ul>
        </nav>
    </header>

    <main>
        <section>
```

```
         <h2>Featured Products</h2>
         <article>Product 1...</article>
         <article>Product 2...</article>
      </section>
   </main>

   <aside>
      <h3>Customer Support</h3>
      <p>Need help? Contact us...</p>
   </aside>

   <footer>
      <p>© 2023 IEKart.com</p>
   </footer>
</body>
</html>
```

Expected Output: * The `Homepage.html` should be well-structured using HTML5 sectioning elements, improving the semantic organization of the content.

Observations:

Experiment 1.c: Grouping Elements - Paragraph, Division, Span, and List

Course Name: HTML5 - The Language **Module Name:** Paragraph Element, Division and Span Elements, List Element

Objective: To use HTML grouping elements to organize content within a webpage, specifically focusing on lists for the "About Us" page.

Procedure:

1. Create a new HTML file named `AboutUs.html` for IEKart's Shopping Application.
2.
3. Develop content for the "About Us" page.
4. Use `<p>` elements for paragraphs of text, `<div>` elements for logical divisions of content, and `<span>` elements for inline grouping of text.

5. For sections that represent lists of information within the "About Us" page (e.g., team members, company milestones, etc.), use appropriate list elements (`<ul>`, `<ol>`, `<li>`, `<dl>`, `<dt>`, `<dd>`).
6. View `AboutUs.html` in a web browser to ensure the content is grouped and displayed correctly using the chosen elements.
7. Refer to the provided link for detailed information about Grouping Elements: https://infyspringboard.onwingspan.com/web/en/viewer/web-module/lex_32785192040894940000_shared?collectionId=lex_17739732834840810000_shared&collectionType=Course

Example Code Snippet (AboutUs.html):

```
HTML
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>About Us - IEKart</title>
</head>
<body>
   <header>
      <h1>About IEKart</h1>
   </header>

   <main>
      <section>
         <h2>Our Story</h2>
         <p>IEKart started with a vision to...</p>

         <h2>Our Team</h2>
         <ul>
            <li><b>CEO:</b> <span>John Doe</span></li>
            <li><b>CTO:</b> <span>Jane Smith</span></li>
         </ul>

         <h2>Milestones</h2>
         <ol>
            <li><b>2010:</b> Company Founded</li>
            <li><b>2015:</b> Reached 1 Million Customers</li>
         </ol>

         <div class="contact-info">
```

```
                <h3>Contact Us</h3>
                <p>Email: <a href="mailto:info@iekart.com">info@iekart.com</a></p>
            </div>
        </section>
    </main>
</body>
</html>
```

Expected Output:

- The `AboutUs.html` page should effectively use grouping elements, particularly lists, to present information in a structured and readable manner.
-

Observations:

Experiment 1.d: Link Elements and Bookmarking

Course Name: HTML5 - The Language **Module Name:** Link Element

Objective: To implement navigation between pages and within a page using HTML link elements and bookmarking.

Procedure:

1. Create `Login.html`, `SignUp.html`, and `Track.html` files for IEKart's application. Add basic content to each page.
2. In `Homepage.html`, add link elements (`<a>`) to connect to these pages. Link "Login" text to `Login.html`, "SignUp" to `SignUp.html`, and "Track Order" to `Track.html`.
3. On the IEKart's application pages (e.g., `Homepage.html` or a details page), identify categories of products (e.g., Electronics, Clothing, Accessories).
4. Create bookmarks (using `id` attributes on headings or sections) for each category.
5. Create links that point to these bookmarks within the same page or from other pages, allowing users to directly navigate to specific category details.
6. Test the links and bookmarks in a web browser to ensure correct navigation.
7. Refer to the provided link for detailed information about Link Elements: https://infyspringboard.onwingspan.com/web/en/viewer/web-module/lex_15515105953273338000_shared?collectionId=lex_17739732834840810000_shared&collectionType=Course

Example Code Snippet (Homepage.html):

```
HTML
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>IEKart Homepage</title>
</head>
<body>
    <header>
        <h1>IEKart</h1>
        <nav>
            <ul>
                <li><a href="Homepage.html">Home</a></li>
                <li><a href="#electronics">Electronics</a></li>
                <li><a href="#clothing">Clothing</a></li>
                <li><a href="#accessories">Accessories</a></li>
                <li><a href="Login.html">Login</a></li>
                <li><a href="SignUp.html">Sign Up</a></li>
                <li><a href="Track.html">Track Order</a></li>
            </ul>
        </nav>
    </header>

    <main>
        <section id="electronics">
            <h2>Electronics</h2>
            <p>Explore our wide range of electronics...</p>
        </section>

        <section id="clothing">
           <h2>Clothing</h2>
            <p>Discover the latest fashion trends...</p>
        </section>

        <section id="accessories">
            <h2>Accessories</h2>
            <p>Complete your look with stylish accessories...</p>
        </section>
    </main>
```

```
  <footer>
    <p>© 2023 IEKart.com</p>
  </footer>
</body>
</html>
```

Example Code Snippet (Login.html, SignUp.html, Track.html - basic content):

HTML
```
<!DOCTYPE html><html><head><title>Login</title></head><body><h1>Login
Page</h1></body></html>

<!DOCTYPE html><html><head><title>Sign Up</title></head><body><h1>Sign Up
Page</h1></body></html>

<!DOCTYPE html><html><head><title>Track Order</title></head><body><h1>Track Your
Order</h1></body></html>
```

Expected Output:

- Navigation links ("Login", "SignUp", "Track Order") should correctly redirect to the respective HTML pages.
- Bookmark links should enable users to jump directly to specific category details within the application pages.

Observations:

Experiment 1.e: Character Entities

Course Name: HTML5 - The Language **Module Name:** Character Entities

Objective: To use HTML character entities to display special characters, like the copyright symbol, on a webpage.

Procedure:

1. Open `Homepage.html`.
2. Locate the footer section.

3. Add the copyright symbol © before the year in the footer text using the HTML character entity `&copy;`.
4. View `Homepage.html` in a web browser to verify that the copyright symbol is displayed correctly.
5. Refer to the provided link for detailed information about Character Entities: https://infyspringboard.onwingspan.com/web/en/viewer/web-module/lex_5476673769380962600_shared?collectionId=lex_17739732834840810000_shared&collectionType=Course

Example Code Snippet (Homepage.html footer section):

```
HTML
<footer>
   <p>&copy; 2023 IEKart.com</p>
</footer>
```

Expected Output:

- The footer of `Homepage.html` should display the copyright symbol ©.

Observations:

Experiment 1.f: HTML5 Global Attributes

Course Name: HTML5 - The Language **Module Name:** HTML5 Global Attributes

Objective: To use HTML5 global attributes like `contenteditable`, `spellcheck`, and `id` to enhance the functionality of web page elements, specifically on the Signup Page.

Procedure:

1. Open `SignUp.html`.
2. For input fields (e.g., for Name, Email, etc.), add the `contenteditable="true"` attribute to make them directly editable on the webpage (for testing or demonstration purposes - *note: in a real signup form, you would not typically make these editable like this after initial input*).
3. Add `spellcheck="true"` to input fields where text input is expected (like address or comments) to enable browser spell checking.

4. Assign unique `id` attributes to different elements on the Signup page (e.g., input fields, form element itself) for easy selection and manipulation using JavaScript or CSS, if needed later.
5. View `SignUp.html` in a web browser and test the functionalities provided by these global attributes.
6. Refer to the provided link for detailed information about HTML5 Global Attributes: https://infyspringboard.onwingspan.com/web/en/viewer/web-module/lex_28723566050321920000_shared?collectionId=lex_17739732834840810000_shared&collectionType=Course

Example Code Snippet (SignUp.html - form section):

```
HTML
<form id="signupForm">
   <label for="name">Name:</label>
   <input type="text" id="name" contenteditable="true" spellcheck="true"><br><br>

   <label for="email">Email:</label>
   <input type="email" id="email" contenteditable="true"><br><br>

   <label for="address">Address:</label>
   <textarea id="address" contenteditable="true" spellcheck="true"></textarea><br><br>

   <button type="submit">Sign Up</button>
</form>
```

Expected Output:

- Input fields on `SignUp.html` with `contenteditable="true"` should be directly editable in the browser.
- Input fields with `spellcheck="true"` should have browser spell checking enabled.
- Elements should have unique `id` attributes.

Observations:

Experiment 2.a: Table Elements

Course Name: HTML5 - The Language **Module Name:** Creating Table Elements, Table Elements : Colspan/Rowspan Attributes, border, cellspacing, cellpadding attributes

Objective: To enhance a webpage by adding tables to display structured data, focusing on attributes like `colspan`, `rowspan`, `border`, `cellspacing`, and `cellpadding`.

Procedure:

1. Create a new HTML file or enhance an existing details page for IEKart's application (e.g., `Details.html`).
2. Add a `<table>` element to display inventory data for mobiles or other products.
3. Create table rows (`<tr>`) and table data cells (`<td>`) to structure the data. Include headers using `<th>` within `<thead>` if needed.
4. Use `colspan` and `rowspan` attributes to merge cells horizontally or vertically as required for the data presentation.
5. Apply attributes like `border`, `cellspacing`, and `cellpadding` to control the table's appearance and cell spacing/padding.
6. View the HTML page in a browser to see the formatted table.
7. Refer to the provided link for detailed information about Table Elements: https://infyspringboard.onwingspan.com/web/en/viewer/web-module/lex_auth_013168035284033536113_shared?collectionId=lex_17739732834840810000_shared&collectionType=Course

Example Code Snippet (Details.html):

HTML
```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>Inventory Details</title>
   <style>
      table { border-collapse: collapse; width: 100%; }
      th, td { border: 1px solid black; padding: 8px; text-align: left; }
   </style>
</head>
<body>
   <h2>Mobile Inventory</h2>
   <table border="1" cellspacing="10" cellpadding="5">
      <thead>
         <tr>
            <th>Model</th>
            <th>Brand</th>
            <th>RAM</th>
```

```
                <th>Storage</th>
                <th>Price</th>
            </tr>
        </thead>
        <tbody>
            <tr>
                <td>Model XYZ</td>
                <td>Brand A</td>
                <td>8GB</td>
                <td>128GB</td>
                <td>$500</td>
            </tr>
            <tr>
                <td>Model ABC</td>
               <td>Brand B</td>
                <td>4GB</td>
                <td>64GB</td>
                <td>$300</td>
            </tr>
            <tr>
                <td colspan="5" style="text-align: center;">End of Inventory</td>
            </tr>
        </tbody>
    </table>
</body>
</html>
```

Expected Output:

- `Details.html` should display a well-formatted table with inventory data, utilizing `colspan`, `rowspan`, `border`, `cellspacing`, and `cellpadding` attributes to control table structure and appearance.
-

Observations:

Experiment 2.b: Form Elements, Pickers, Select and Datalist Elements

Course Name: HTML5 - The Language **Module Name:** Creating Form Elements, Color and Date Pickers, Select and Datalist Elements

Objective: To construct a Signup page using various HTML form elements, including color and date pickers, and `select` and `datalist` elements for user input.

Procedure:

1. Open `SignUp.html`.
2. Within a `<form>` element, add different types of input fields:
   - Text fields (`<input type="text">`) for name, username.
   - Password field (`<input type="password">`) for password.
   - Email field (`<input type="email">`) for email.
   - Color picker (`<input type="color">`) for color selection.
   - Date picker (`<input type="date">`) for date of birth.
   - Dropdown select menu (`<select>`) with `<option>` elements for selecting options (e.g., country).
   - Datalist (`<datalist>`) paired with an input field (`<input list="datalistID">`) to provide suggestions as the user types (e.g., for city).
3. Add appropriate labels (`<label>`) for each form element.
4. Include a submit button (`<button type="submit">`).
5. View `SignUp.html` in a browser and interact with the form elements.
6. Refer to the provided link for detailed information about Form Elements: https://infyspringboard.onwingspan.com/web/en/viewer/web-module/lex_13975270903118459000_shared?collectionId=lex_17739732834840810000_shared&collectionType=Course

Example Code Snippet (SignUp.html - form section):

```
HTML
<form>
   <label for="name">Name:</label>
   <input type="text" id="name" name="name"><br><br>

   <label for="password">Password:</label>
   <input type="password" id="password" name="password"><br><br>

   <label for="email">Email:</label>
   <input type="email" id="email" name="email"><br><br>

   <label for="color">Favorite Color:</label>
```

```
  <input type="color" id="color" name="color"><br><br>

  <label for="dob">Date of Birth:</label>
  <input type="date" id="dob" name="dob"><br><br>

  <label for="country">Country:</label>
  <select id="country" name="country">
    <option value="">Select Country</option>
    <option value="usa">USA</option>
    <option value="canada">Canada</option>
    </select><br><br>

  <label for="city">City:</label>
  <input type="text" id="city" name="city" list="cities"><br>
  <datalist id="cities">
    <option value="New York"></option>
    <option value="Los Angeles"></option>
    <option value="Chicago"></option>
  </datalist><br><br>

  <button type="submit">Sign Up</button>
</form>
```

Expected Output:

- `SignUp.html` should render a signup form with various input types including text, password, email, color picker, date picker, select dropdown, and datalist with suggestions.
-

Observations:

Experiment 2.c: Input Element Attributes in Forms

Course Name: HTML5 - The Language **Module Name:** Input Elements - Attributes

Objective: To enhance the Signup page functionality by utilizing various attributes of HTML input elements to add constraints, placeholders, and improve user experience.

Procedure:

1. Open `SignUp.html` from Experiment 2.b.
2. For different input fields, add relevant attributes:
   - `placeholder="Enter your name"` for text fields to show example input.
   - `required` for fields that must be filled out before form submission.
   - `minlength="8"` and `maxlength="20"` for password fields to enforce password length constraints.
   - `pattern="[a-z0-9._%+-]+@[a-z0-9.-]+\.[a-z]{2,}$"` for email field for basic email format validation (HTML5 validation).
   - `autocomplete="on"` or `autocomplete="off"` as needed for form fields.
   - `disabled` or `readonly` if you want to temporarily disable or make an input field read-only.
3. Test the Signup page in a browser to observe the effects of these attributes (e.g., try submitting the form without filling required fields, check placeholder text, test password length constraints, etc.).
4. Refer to the provided link for detailed information about Input Element Attributes: https://infyspringboard.onwingspan.com/web/en/viewer/web-module/lex_14048414537062347000_shared?collectionId=lex_17739732834840810000_shared&collectionType=Course

Example Code Snippet (SignUp.html - form section, enhanced):

HTML
```
<form>
   <label for="name">Name:</label>
   <input type="text" id="name" name="name" placeholder="Enter your full name"
required><br><br>

   <label for="password">Password:</label>
   <input type="password" id="password" name="password" minlength="8" maxlength="20"
required><br><br>

   <label for="email">Email:</label>
   <input type="email" id="email" name="email" placeholder="user@example.com" pattern="[a-
z0-9._%+-]+@[a-z0-9.-]+\.[a-z]{2,}$" required><br><br>

   <label for="city">City:</label>
   <input type="text" id="city" name="city" list="cities" autocomplete="on"><br>
   <datalist id="cities">
      <option value="New York"></option>
      <option value="Los Angeles"></option>
      <option value="Chicago"></option>
   </datalist><br><br>
```

```
  <button type="submit">Sign Up</button>
</form>
```

Expected Output:

- The Signup form in `SignUp.html` should exhibit enhanced functionality with placeholders, required fields, password constraints, and basic email validation due to the added input attributes.

Observations:

Experiment 2.d: Media, Iframe

Course Name: HTML5 - The Language **Module Name:** Media, Iframe

Objective: To embed media content (audio and video) and external web pages into the Homepage of IEKart's Shopping Application using `<audio>`, `<video>`, and `<iframe>` elements.

Procedure:

1. Open `Homepage.html`.
2. Within the `<main>` section, or in an appropriate section, add:
   - An `<audio>` element to embed an audio file (e.g., background music, promotional audio). Use the `<source>` element inside `<audio>` to specify audio file path and type, and add `controls` attribute to display audio controls.
   - A `<video>` element to embed a video file (e.g., product demo video, advertisement). Use `<source>` inside `<video>` to specify video file path and type, and add `controls`, `width`, `height` attributes.
   - An `<iframe>` element to embed another webpage (e.g., link to a blog, promotional page, or external content related to IEKart). Set the `src` attribute to the URL of the webpage to be embedded, and adjust `width` and `height`.
3. Ensure you have audio and video files available (or use placeholder URLs for testing).
4. View `Homepage.html` in a web browser to verify that the audio, video, and iframe content are embedded and displayed correctly.
5. Refer to the provided link for detailed information about Media and Iframes: https://infyspringboard.onwingspan.com/web/en/viewer/web-

module/lex_3073840222579494500_shared?collectionId=lex_17739732834840810000_shared&collectionType=Course

Example Code Snippet (Homepage.html - media section):

```
HTML
<section>
   <h2>Welcome to IEKart - Explore Exciting Products!</h2>

   <audio controls>
      <source src="audio/background_music.mp3" type="audio/mpeg">
      Your browser does not support the audio element.
   </audio>

   <video controls width="320" height="240">
      <source src="videos/product_demo.mp4" type="video/mp4">
      Your browser does not support the video element.
   </video>

   <iframe src="https://iekart-blog-example.com" width="600" height="400" title="IEKart Blog">
      </iframe>
</section>
```

Note: You will need to replace `"audio/background_music.mp3"` and `"videos/product_demo.mp4"` with actual paths to your audio and video files or use publicly accessible URLs for testing purposes. Similarly, replace `"https://iekart-blog-example.com"` with a real URL.

Expected Output:

- `Homepage.html` should embed and display audio and video controls for the respective media files and should embed the webpage from the specified URL within the iframe.

Observations:

3. JavaScript Exercises

Tools Required:

- Web Browser with Developer Tools (Chrome, Firefox, etc.)
-
- Text Editor (VS Code, Sublime Text, Atom, Notepad++, etc.)
-

Experiment 3.a: Identifiers and Scope (var, let, const)

Course Name: Javascript **Module Name:** Type of Identifiers

Objective: To understand the difference between `var`, `let`, and `const` in JavaScript concerning variable declaration and scope, particularly reassignment.

Procedure:

1. Create a new HTML file (e.g., `area_circle.html`) and include `<script>` tags within the `<body>` or `<head>` section.
2. Write JavaScript code to calculate the area of a circle.
3. Declare the radius variable using `var` and calculate the area. Reassign a new value to the radius variable and observe the output.
4. Repeat step 3, but declare the radius variable using `let`. Observe if reassignment is allowed and the output.
5. Declare the PI constant using `const` and assign the value 3.14159. Attempt to reassign a new value to PI and observe the result (error).
6. Display the calculated area (using both `var` and `let` for radius) on the webpage or console.
7. Refer to the provided link for detailed information about Identifiers: https://infyspringboard.onwingspan.com/web/en/viewer/web-module/lex_auth_0130532644148183047 32_shared?collectionId=lex_18109698366332810000_shared&collectionType=Course

Example Code Snippet (area_circle.html):

HTML
```
<!DOCTYPE html>
<html>
<head><title>Area of Circle</title></head>
<body>
  <h1>Calculate Area of Circle</h1>
  <div id="output"></div>

  <script>
    const PI = 3.14159;
    var radiusVar = 5;
    let radiusLet = 5;

    // Area with var
    var areaVar = PI * radiusVar * radiusVar;
    radiusVar = 10; // Reassign var
    areaVar = PI * radiusVar * radiusVar; // Recalculate
    console.log("Area with var (after reassignment):", areaVar);

    // Area with let
    let areaLet = PI * radiusLet * radiusLet;
    radiusLet = 10; // Reassign let
    areaLet = PI * radiusLet * radiusLet; // Recalculate
    console.log("Area with let (after reassignment):", areaLet);

    // Attempt to reassign const - will cause error
    // PI = 3.14; // Uncommenting this will cause an error

    document.getElementById('output').innerHTML =
      "<p>Area with var (after reassignment): " + areaVar + "</p>" +
      "<p>Area with let (after reassignment): " + areaLet + "</p>";
  </script>
</body>
</html>
```

Expected Output:

- Code should demonstrate that `var` allows reassignment and function scope.
- Code should demonstrate that `let` allows reassignment and block scope.
- Code should demonstrate that `const` does not allow reassignment and has block scope, resulting in an error upon reassignment attempt.
- The calculated area of the circle should be displayed for both `var` and `let` scenarios.

Observations:

Experiment 3.b: Primitive and Non Primitive Data Types

Course Name: Javascript **Module Name:** Primitive and Non Primitive Data Types

Objective: To understand and use primitive and non-primitive data types in JavaScript to represent movie details, using template literals for output.

Procedure:

1. Create a new HTML file (e.g., `movie_details.html`) and include `<script>` tags.
2. Declare variables to store movie details: `movieName` (string), `starring` (string), `language` (string), and `ratings` (number). Initialize them with movie information.
3. Use template literals to construct a string that displays all movie details in a formatted way.
4. Output the movie details string to the webpage or console.
5. Refer to the provided link for detailed information about Data Types: https://infyspringboard.onwingspan.com/web/en/viewer/web-module/lex_21528322245232402000_shared?collectionId=lex_18109698366332810000_shared&collectionType=Course

Example Code Snippet (movie_details.html):

```
HTML
<!DOCTYPE html>
<html>
<head><title>Movie Details</title></head>
<body>
  <div id="movieInfo"></div>

  <script>
    let movieName = "Avengers: Endgame";
    let starring = "Robert Downey Jr., Chris Evans";
    let language = "English";
    let ratings = 4.9;

    let movieDetails = `
      <h2>Movie Details</h2>
      <p><b>Movie Name:</b> ${movieName}</p>
      <p><b>Starring:</b> ${starring}</p>
      <p><b>Language:</b> ${language}</p>
      <p><b>Ratings:</b> ${ratings} stars</p>
    `;

    document.getElementById('movieInfo').innerHTML = movieDetails;
    console.log(movieDetails);
  </script>
</body>
```

```
</html>
```

Expected Output:

- The webpage (or console) should display the movie details formatted using template literals and showing values of different primitive data types.

Observations:

Experiment 3.c: Operators and Calculation

Course Name: Javascript **Module Name:** Operators and Types of Operators

Objective: To write JavaScript code to perform calculations for online movie ticket booking, including discount application using arithmetic and assignment operators.

Procedure:

1. Create a new HTML file (e.g., `ticket_booking.html`) with `<script>` tags.
2. Declare variables for `numberOfTickets` and `pricePerTicket` (initialize `pricePerTicket` to 150).
3. Calculate `totalPrice` before discount using multiplication operator.
4. Declare a `discountPercentage` (e.g., 10 for 10%).
5. Calculate `discountAmount` and `discountedPrice` using arithmetic operators.
6. Display the original price, discount, and discounted price on the webpage or console using template literals.
7. Refer to the provided link for detailed information about Operators: https://infyspringboard.onwingspan.com/web/en/viewer/web-module/lex_13808338384481720000_shared?collectionId=lex_18109698366332810000_shared&collectionType=Course

Example Code Snippet (ticket_booking.html):

HTML
```
<!DOCTYPE html>
<html>
<head><title>Ticket Booking</title></head>
<body>
  <div id="bookingSummary"></div>
  <script>
    let numberOfTickets = 4;
    const pricePerTicket = 150;
    let discountPercentage = 10;

    let totalPrice = numberOfTickets * pricePerTicket;
    let discountAmount = totalPrice * (discountPercentage / 100);
    let discountedPrice = totalPrice - discountAmount;

    let bookingDetails = `
      <h2>Ticket Booking Summary</h2>
      <p>Number of Tickets: ${numberOfTickets}</p>
      <p>Price per Ticket: Rs. ${pricePerTicket}</p>
      <p>Total Price: Rs. ${totalPrice}</p>
      <p>Discount (${discountPercentage}%): Rs. ${discountAmount}</p>
      <p>Discounted Price: Rs. ${discountedPrice}</p>
    `;

    document.getElementById('bookingSummary').innerHTML = bookingDetails;
    console.log(bookingDetails);
  </script>
</body>
</html>
```

Expected Output:

- The webpage (or console) should display the ticket booking summary, including total price, discount amount, and discounted price, calculated using JavaScript operators.

Observations:

Experiment 3.d: Conditional Statements (if, switch)

Course Name: Javascript **Module Name:** Types of Statements, Non - Conditional Statements, Types of Conditional Statements, if Statements, switch Statements

Objective: To implement conditional logic in JavaScript for movie ticket booking based on the number of seats, using `if` and `switch` statements.

Procedure:

1. Create a new HTML file (e.g., `conditional_booking.html`) with `<script>` tags.
2. Declare a variable `seatsToBook`. Get user input or set a default value for `seatsToBook`.
3. Use `if-else if-else` statements to implement the following conditions:
    - If `seatsToBook` is not more than 2, set `costPerTicket` to 150.
    - If `seatsToBook` is 6 or more, display "Booking not allowed".
    - Otherwise (if seats are between 3 and 5), set `costPerTicket` to a different value (e.g., 180).
4. Optionally, demonstrate a `switch` statement for a different condition (e.g., day of the week to apply different discounts, though if-else is more appropriate for seat conditions).
5. Display the booking confirmation or rejection message and total price (if booking is allowed) on the webpage or console.
6. Refer to the provided link for detailed information about Conditional Statements: https://infyspringboard.onwingspan.com/web/en/viewer/web-module/lex_16257498471333610000_shared?collectionId=lex_18109698366332810000_shared&collectionType=Course

Example Code Snippet (conditional_booking.html):

HTML
```
<!DOCTYPE html>
<html>
<head><title>Conditional Ticket Booking</title></head>
<body>
  <div id="bookingResult"></div>

  <script>
    let seatsToBook = 4; // Example: try changing this value
    let costPerTicket;
    let bookingMessage = "";

    if (seatsToBook <= 2) {
      costPerTicket = 150;
      bookingMessage = "Booking confirmed. Cost per ticket: Rs. 150.";
    } else if (seatsToBook >= 6) {
      bookingMessage = "Sorry, booking not allowed for more than 5 seats.";
      costPerTicket = 0; // No cost as booking is not allowed
    } else {
      costPerTicket = 180;
      bookingMessage = "Booking confirmed. Cost per ticket: Rs. 180.";
    }
```

```
        let totalPrice = seatsToBook * costPerTicket;

        let output = `<h2>Booking Result</h2><p>${bookingMessage}</p>`;
        if (costPerTicket > 0) {
            output += `<p>Total Price for ${seatsToBook} seats: Rs. ${totalPrice}</p>`;
        }

        document.getElementById('bookingResult').innerHTML = output;
        console.log(output);
    </script>
</body>
</html>
```

Expected Output:

- Based on the `seatsToBook` value, the webpage (or console) should display the appropriate booking message and, if applicable, the total price, according to the conditional logic implemented.

Observations:

Experiment 3.e: Loops

Course Name: Javascript **Module Name:** Types of Loops

Objective: To use JavaScript loops to iterate and perform actions, demonstrated in the context of movie ticket booking with conditional pricing based on the number of seats.

Procedure:

1. Create a new HTML file (e.g., `loop_booking.html`) with `<script>` tags.
2. For demonstration, assume you want to allow booking for multiple seat quantities and display the price for each valid quantity (e.g., for 1 to 5 seats, excluding 6 and above as per conditions).
3. Use a `for` loop to iterate through seat quantities from 1 to 5.
4. Inside the loop, apply the same conditional logic as in Experiment 3.d to determine `costPerTicket` based on the current seat quantity in the loop.
5. Calculate `totalPrice` for each seat quantity.

6. Display the seat quantity and corresponding total price for all valid bookings (1 to 5 seats) on the webpage or console.
7. Refer to the provided link for detailed information about Loops: https://infyspringboard.onwingspan.com/web/en/viewer/web-module/lex_6238536888292970000_shared?collectionId=lex_18109698366332810000_shared&collectionType=Course

Example Code Snippet (loop_booking.html):

```
HTML
<!DOCTYPE html>
<html>
<head><title>Loop Based Ticket Booking</title></head>
<body>
  <div id="bookingPrices"></div>

  <script>
    let bookingOutput = "<h2>Ticket Prices for Different Seat Quantities</h2><ul>";

    for (let seatsToBook = 1; seatsToBook <= 5; seatsToBook++) {
      let costPerTicket;
      let message = "";

      if (seatsToBook <= 2) {
        costPerTicket = 150;
        message = `For ${seatsToBook} seats, cost per ticket is Rs. 150.`;
      } else {
        costPerTicket = 180;
        message = `For ${seatsToBook} seats, cost per ticket is Rs. 180.`;
      }

      let totalPrice = seatsToBook * costPerTicket;
      bookingOutput += `<li>${message} Total Price: Rs. ${totalPrice}</li>`;
    }
    bookingOutput += "</ul>";

    document.getElementById('bookingPrices').innerHTML = bookingOutput;
    console.log(bookingOutput);
  </script>
</body>
</html>
```

Expected Output:

- The webpage (or console) should display a list of ticket prices for seat quantities from 1 to 5, calculated using a `for` loop and conditional pricing logic.

Observations:

Experiment 4.a: Functions and Scope

Course Name: Javascript **Module Name:** Types of Functions, Declaring and Invoking Function, Arrow Function, Function Parameters, Nested Function, Built-in Functions, Variable Scope in Functions

Objective: To implement a JavaScript function to calculate movie ticket prices with conditional logic based on seat count, demonstrating different function types and scope.

Procedure:

1. Create a new HTML file (e.g., `function_booking.html`) with `<script>` tags.
2. Define a function (using function declaration or arrow function) named `calculateTicketPrice` that accepts `seats` as a parameter.
3. Inside the function, implement the same conditional logic for ticket pricing as in Experiment 3.d based on the `seats` parameter. The function should return the `totalPrice` or a message if booking is not allowed.
4. Invoke the `calculateTicketPrice` function for different seat counts (e.g., 2, 4, 6) and display the returned results on the webpage or console.
5. Demonstrate variable scope by declaring variables inside and outside the function and observing their accessibility.
6. Refer to the provided link for detailed information about Functions: https://infyspringboard.onwingspan.com/web/en/viewer/web-module/lex_1545519957061332600 0_shared?collectionId=lex_18109698366332810000_shared&collectionType=Course
7.

Example Code Snippet (function_booking.html):

```
HTML
<!DOCTYPE html>
<html>
<head><title>Function Based Ticket Booking</title></head>
<body>
```

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

    <script>
        function calculateTicketPrice(seats) {
            let costPerTicket;
            if (seats <= 2) {
                costPerTicket = 150;
            } else if (seats >= 6) {
                return "Booking not allowed for more than 5 seats.";
            } else {
                costPerTicket = 180;
            }
            return seats * costPerTicket;
        }

        let bookingOutput = "<h2>Ticket Prices Using Function</h2><ul>";
        let seatsOptions = [2, 4, 6];

        seatsOptions.forEach(seats => {
            let price = calculateTicketPrice(seats);
            let message = typeof price === 'string' ? price : `Total Price for ${seats} seats: Rs.
${price}`;
            bookingOutput += `<li>${message}</li>`;
        });
        bookingOutput += "</ul>";

        document.getElementById('bookingResults').innerHTML = bookingOutput;
        console.log(bookingOutput);
    </script>
</body>
</html>
```

Expected Output:

- The webpage (or console) should display the ticket prices calculated by the `calculateTicketPrice` function for different seat inputs, demonstrating function usage and conditional logic.

Observations:

Experiment 4.b: Classes and Inheritance

Course Name: Javascript **Module Name:** Working With Classes, Creating and Inheriting Classes

Objective: To implement class inheritance in JavaScript by creating an `Employee` class that inherits from a `Person` base class, demonstrating class attributes, constructors, and inheritance.

Procedure:

1. Create a new HTML file (e.g., `classes_inheritance.html`) with `<script>` tags.
2. Define a base class `Person` with attributes `name` and `age`. Include a constructor to initialize these attributes.
3. Define a class `Employee` that extends `Person`. Add an additional attribute `role` and a constructor that calls the `super()` constructor to initialize inherited attributes from `Person` and then initializes the `role` attribute.
4. Create instances (objects) of both `Person` and `Employee` classes.
5. Display the properties of these objects on the webpage or console to demonstrate class creation and inheritance.
6. Refer to the provided link for detailed information about Classes and Inheritance: https://infyspringboard.onwingspan.com/web/en/viewer/web-module/lex_auth_0125998111177605124 58_shared?collectionId=lex_18109698366332810000_shared&collectionType=Course

Example Code Snippet (classes_inheritance.html):

```
HTML
<!DOCTYPE html>
<html>
<head><title>Classes and Inheritance</title></head>
<body>
  <div id="classOutput"></div>

  <script>
    class Person {
      constructor(name, age) {
        this.name = name;
        this.age = age;
      }
      toString() {
        return `Name: ${this.name}, Age: ${this.age}`;
      }
```

```
        }

        class Employee extends Person {
            constructor(name, age, role) {
                super(name, age); // Call Person constructor
                this.role = role;
            }
            toString() {
                return `${super.toString()}, Role: ${this.role}`;
            }
        }

        let person1 = new Person("John Doe", 30);
        let employee1 = new Employee("Jane Smith", 25, "Developer");

        let output = `<h2>Class Examples</h2>
                      <p>Person Details: ${person1.toString()}</p>
                      <p>Employee Details: ${employee1.toString()}</p>`;

        document.getElementById('classOutput').innerHTML = output;
        console.log(output);
    </script>
</body>
</html>
```

Expected Output:

- The webpage (or console) should display details of `Person` and `Employee` objects, demonstrating inheritance by showing attributes from both classes.

Observations:

Experiment 4.c: In-built Events and Handlers

Course Name: Javascript **Module Name:** In-built Events and Handlers

Objective: To handle in-built JavaScript events to control actions on a webpage, using the movie ticket booking example to update prices dynamically based on seat selection.

Procedure:

1. Create a new HTML file (e.g., `event_booking.html`). Include an input field for the number of seats and a button to calculate the price.
2. Attach an event listener to the button (e.g., using `onclick` in HTML or `addEventListener` in JavaScript).
3. In the event handler function, get the number of seats from the input field.
4. Call the `calculateTicketPrice` function (from Experiment 4.a) with the number of seats.
5. Display the booking result (price or error message) in a designated area on the webpage.
6. Refer to the provided link for detailed information about Events and Handlers: https://infyspringboard.onwingspan.com/web/en/viewer/web-module/lex_4192188372573027000_shared?collectionId=lex_18109698366332810000_shared&collectionType=Course

Example Code Snippet (event_booking.html):

```
HTML
<!DOCTYPE html>
<html>
<head><title>Event Handling in Booking</title></head>
<body>
   <h2>Movie Ticket Booking</h2>
   <label for="seatsInput">Number of Seats:</label>
   <input type="number" id="seatsInput" value="1" min="1" max="5">
   <button onclick="showBookingPrice()">Calculate Price</button>

   <div id="bookingPriceResult"></div>

   <script>
      function calculateTicketPrice(seats) { // Function from Experiment 4.a
         let costPerTicket;
         if (seats <= 2) {
            costPerTicket = 150;
         } else if (seats >= 6) {
            return "Booking not allowed for more than 5 seats.";
         } else {
            costPerTicket = 180;
         }
         return seats * costPerTicket;
      }

      function showBookingPrice() {
            let seats = document.getElementById('seatsInput').value;
            let priceResult = calculateTicketPrice(seats);
            let message = typeof priceResult === 'string' ? priceResult : `Total Price: Rs.
${priceResult}`;
            document.getElementById('bookingPriceResult').textContent = message;
        }
    </script>
</body>
</html>
```

Expected Output:

- When the button is clicked, the webpage should dynamically update to display the calculated ticket price or booking error message based on the number of seats entered.

Observations:

Experiment 4.d: Objects and DOM Manipulation

Course Name: Javascript **Module Name:** Working with Objects, Types of Objects, Creating Objects, Combining and cloning Objects using Spread operator, Destructuring Objects, Browser Object Model, Document Object Model

Objective: To manipulate the Document Object Model (DOM) in JavaScript based on user interaction, specifically changing page elements' content and styles on a button click, simulating a cone filling/refilling scenario.

Procedure:

1. Create a new HTML file (e.g., `dom_manipulation.html`). Include an empty `<div>` to represent a cone, a heading, a message area, and a button.
2. Write JavaScript code to:
    - Get references to the cone `<div>`, heading, message area, and button using `document.getElementById` or similar methods.
    - Add an event listener to the button for the 'click' event.
    - In the event handler function:
        - Toggle the state of the cone (empty/refilled). You can achieve this by changing the background color or class of the cone `<div>`.
        - Change the text content of the heading and message area to reflect the current state (e.g., "Cone Empty" vs "Cone Refilled").

-
- Change the background color of the page (or another element) to further indicate the state change.

3. Initialize the page in the "empty cone" state on page load.
4. Refer to the provided link for detailed information about Objects and DOM: https://infyspringboard.onwingspan.com/web/en/viewer/web-module/lex_13197025862804100000_shared?collectionId=lex_18109698366332810000_shared&collectionType=Course

Example Code Snippet (dom_manipulation.html):

```
HTML
<!DOCTYPE html>
<html>
<head><title>DOM Manipulation</title>
  <style>
    #cone {
      width: 100px;
      height: 200px;
      border-bottom: 150px solid brown;
      border-left: 50px solid transparent;
      border-right: 50px solid transparent;
      margin: 20px auto;
      background-color: white; /* Initially empty */
    }
    .refilled {
      background-color: yellow; /* Refilled color */
    }
  </style>
</head>
<body>
  <h2 id="heading">Cone Empty</h2>
  <div id="cone"></div>
  <p id="message">Click the button to fill the cone.</p>
  <button id="coneButton">Fill Cone</button>

  <script>
    let coneElement = document.getElementById('cone');
    let headingElement = document.getElementById('heading');
    let messageElement = document.getElementById('message');
    let buttonElement = document.getElementById('coneButton');
    let isFilled = false; // Track cone state
```

```
        buttonElement.addEventListener('click', function() {
            isFilled = !isFilled; // Toggle state
            if (isFilled) {
                coneElement.classList.add('refilled');
                coneElement.classList.remove('empty');
                headingElement.textContent = "Cone Refilled!";
                messageElement.textContent = "Cone is now full.";
                document.body.style.backgroundColor = "#lightgreen";
                buttonElement.textContent = "Empty Cone";
            } else {
                coneElement.classList.remove('refilled');
                coneElement.classList.add('empty');
                coneElement.style.backgroundColor = "white"; // Explicitly set background to white
                headingElement.textContent = "Cone Empty";
                messageElement.textContent = "Click the button to fill the cone.";
                document.body.style.backgroundColor = "white"; // Reset body background
                buttonElement.textContent = "Fill Cone";
            }
        });
    </script>
</body>
</html>
```

Expected Output:

- Initially, the page should show an "empty cone" state.
- Clicking the button should toggle the cone to a "refilled" state (visual change in cone, heading, message, and background color). Clicking again should revert it back to the "empty cone" state.

Observations:

Experiment 5.a: Arrays and Array Methods

Course Name: Javascript **Module Name:** Creating Arrays, Destructuring Arrays, Accessing Arrays, Array Methods

Objective: To work with JavaScript arrays and array methods to manage and display a list of movie details, demonstrating array creation, object storage, and rendering array data to a webpage.

Procedure:

1. Create a new HTML file (e.g., `movie_array.html`) with `<script>` tags.
2. Create an array named `movies`. Each element in the array should be an object representing movie details with properties: `name`, `starring`, `language`, `ratings`. Add details for at least three movies.
3. Use array methods (e.g., `forEach`, `map`) to iterate over the `movies` array.
4. For each movie object, create HTML elements (e.g., `<p>`, `<div>`, `<ul>`) to display the movie details.
5. Append these HTML elements to a designated container element in the HTML file to render the movie list on the webpage.
6. Optionally, demonstrate array destructuring to access movie object properties.
7. Refer to the provided link for detailed information about Arrays: https://infyspringboard.onwingspan.com/web/viewer/web-module/lex_auth_013053270191734784711_shared?collectionId=lex_18109698366332810000_shared&collectionType=Course

Example Code Snippet (movie_array.html):

```
HTML
<!DOCTYPE html>
<html>
<head><title>Movie Array</title></head>
<body>
  <div id="movieList"></div>

  <script>
    const movies = [
      { name: "Avengers: Endgame", starring: "Robert Downey Jr., Chris Evans", language:
"English", ratings: 4.9 },
      { name: "Avatar", starring: "Sam Worthington, Zoe Saldana", language: "English",
ratings: 4.7 },
      { name: "Spirited Away", starring: "Rumi Hiiragi, Miyu Irino", language: "Japanese",
ratings: 4.8 }
    ];

    let movieListHTML = "<h2>Movie List</h2><ul>";
    movies.forEach(movie => {
      const { name, starring, language, ratings } = movie; // Destructuring
      movieListHTML += `<li>
                <b>${name}</b> - Starring: ${starring}, Language: ${language}, Ratings:
${ratings} stars
```

```
                      </li>`;
        });
        movieListHTML += "</ul>";

        document.getElementById('movieList').innerHTML = movieListHTML;
        console.log(movieListHTML);
    </script>
</body>
</html>
```

Expected Output:

- The webpage (or console) should display a list of movie details, rendered from the `movies` array, with each movie's information clearly presented.

Observations:

Experiment 5.b: Asynchronous Programming and Fetch API

Course Name: Javascript **Module Name:** Introduction to Asynchronous Programming, Callbacks, Promises, Async and Await, Executing Network Requests using Fetch API

Objective: To simulate periodic stock price changes using JavaScript's asynchronous features and display these changes, mimicking fetching data from an API, using `setInterval` and basic random number generation.

Procedure:

1. Create a new HTML file (e.g., `stock_price.html`) with `<script>` tags.
2. Write a function `getRandomPriceChange()` that returns a random number (positive or negative) representing a stock price change. Use `Math.random()` and `Math.floor()` (or similar methods) to generate a rounded random value.
3. Use `setInterval()` to invoke a function every 3 seconds.
4. Inside the `setInterval` callback function:
   - Call `getRandomPriceChange()` to get a new price change.
   - Update a variable representing the current stock price by adding the price change.
   - Display the updated stock price on the webpage or console.

   - Add a condition to stop the `setInterval` after a certain number of updates (e.g., after 10 updates) using `clearInterval()`.
5. Use `Fetch API` conceptually - you are simulating a price change, no actual API call is needed for this exercise, but understand how `fetch` is used for real network requests in asynchronous JavaScript.
6. Refer to the provided link for detailed information about Asynchronous Programming and Fetch API: https://infyspringboard.onwingspan.com/web/viewer/web-module/lex_auth_0125998116339056644 60_shared?collectionId=lex_18109698366332810000_shared&collectionType=Course

Example Code Snippet (stock_price.html):

HTML
```
<!DOCTYPE html>
<html>
<head><title>Simulated Stock Price</title></head>
<body>
  <h2>Stock Price Simulator</h2>
  <div id="stockPriceDisplay">Current Price: $100</div>

  <script>
    let currentPrice = 100;
    let priceDisplay = document.getElementById('stockPriceDisplay');
    let updateCount = 0;
    const maxUpdates = 10;

    function getRandomPriceChange() {
      let change = Math.floor(Math.random() * 10) - 5; // Random change between -5 and +4
      return change;
    }

    let intervalId = setInterval(function() {
      let priceChange = getRandomPriceChange();
      currentPrice += priceChange;
      priceDisplay.textContent = `Current Price: $${currentPrice}`;
      console.log(`Price updated to: $${currentPrice} (Change: ${priceChange})`);
      updateCount++;

      if (updateCount >= maxUpdates) {
        clearInterval(intervalId);
        console.log("Stock price simulation stopped.");
      }
      }, 3000); // Update every 3 seconds (3000 milliseconds)
   </script>
</body>
</html>
```

Expected Output:

- The webpage (and console) should display the stock price updating every 3 seconds, with random price changes, for a limited number of updates, simulating asynchronous data updates.

Observations:

Experiment 5.c: Modules

Course Name: Javascript **Module Name:** Creating Modules, Consuming Modules

Objective: To create and consume JavaScript modules for user validation, demonstrating modular code organization and reusability by creating a login module.

Procedure:

1. Create two JavaScript files:
   - `login.js`: This will be your module. Define a class `User` inside this file. Implement a static method `validate(username, password)` within the `User` class. This method should check if `username` and `password` are equal (for simplicity, you can compare against hardcoded values like "testuser" and "password123"). Return "Login Successful" or "Login Failed" accordingly.
   - `main.js`: This will be your main script to consume the `login.js` module.
2. In `main.js`, import the `User` class from `login.js` using `import`.
3. In `main.js`, write code to get username and password (you can hardcode these for simplicity or get from user input).
4. Call `User.validate(username, password)` to validate the user.
5. Display the validation result ("Login Successful" or "Login Failed") on the webpage or console.
6. Create an HTML file (e.g., `login_module.html`) and include `main.js` as a module script (`<script type="module" src="main.js"></script>`).
7. Refer to the provided link for detailed information about Modules: https://infyspringboard.onwingspan.com/web/viewer/web-

module/lex_auth_013052857053585408667_shared?collectionId=lex_18109698366332810000_shared&collectionType=Course

Example Code Snippet (login.js - module file):

```
JavaScript
// login.js
export class User {
  static validate(username, password) {
    if (username === "testuser" && password === "password123") {
      return "Login Successful";
    } else {
      return "Login Failed";
    }
  }
}
```

Example Code Snippet (main.js - main script file):

```
JavaScript
// main.js
import { User } from './login.js'; // Assuming login.js is in the same directory

const username = "testuser"; // Example username
const password = "password123"; // Example password

const loginResult = User.validate(username, password);
console.log(loginResult); // Output to console
document.getElementById('loginResultDisplay').textContent = loginResult; // Display on
webpage
```

Example Code Snippet (login_module.html - HTML file):

```
HTML
<!DOCTYPE html>
<html>
<head><title>Login Module</title></head>
<body>
  <h2>User Login Validation</h2>
  <div id="loginResultDisplay"></div>

  <script type="module" src="main.js"></script>
```

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

Expected Output:

- When `login_module.html` is opened, the webpage (and console) should display "Login Successful" if the username and password in `main.js` match the validation criteria in `login.js`, otherwise "Login Failed". This demonstrates module creation and consumption.

Observations:

---

4. Node.js Exercises

Tools Required:

- Web Browser (Chrome, Firefox, etc.)
- Text Editor (VS Code, Sublime Text, Atom, Notepad++, etc.)
- Node.js and NPM (Node Package Manager)
- Command Prompt/Terminal

Experiment 6.a: Executing Functions in Node.js

Course Name: Node.js **Module Name:** How to use Node.js

Objective: To verify the execution of JavaScript functions within the Node.js environment, understanding Node.js as a JavaScript runtime environment.

Procedure:

1. Create a new JavaScript file (e.g., `functions.js`).
2. Define several JavaScript functions within `functions.js`. These functions can perform simple tasks like:
    - A function to add two numbers.
    - A function to reverse a string.
    - A function to check if a number is even or odd.
3. Use `console.log()` within each function to output results or messages to the console.

4. Open your command prompt or terminal, navigate to the directory where you saved `functions.js`.
5. Execute the script using Node.js by running the command: `node functions.js`.
6. Observe the output in the console, verifying that your functions have executed correctly in the Node.js environment.
7. Refer to the provided link for detailed information about using Node.js: https://infyspringboard.onwingspan.com/web/viewer/web-module/lex_19002830632103186000_shared?collectionId=lex_32407835671946760000_shared&collectionType=Course

Example Code Snippet (functions.js):

```
JavaScript
// functions.js

function add(a, b) {
  let sum = a + b;
  console.log(`Sum of ${a} and ${b} is: ${sum}`);
}

function reverseString(str) {
  let reversed = str.split("").reverse().join("");
  console.log(`Reversed string of "${str}" is: "${reversed}"`);
}

function isEven(num) {
  if (num % 2 === 0) {
    console.log(`${num} is even.`);
  } else {
    console.log(`${num} is odd.`);
  }
}

add(5, 3);
reverseString("hello");
isEven(7);
isEven(10);
```

Expected Output:

- When you run `node functions.js` in the terminal, you should see the output from `console.log` statements within your functions, confirming their execution in Node.js.

Observations:

Experiment 6.b: Creating a Web Server in Node.js

Course Name: Node.js **Module Name:** Create a web server in Node.js

Objective: To write a basic Node.js program to create a web server, demonstrating the workflow of JavaScript code execution in a server-side environment.

Procedure:

1. Create a new JavaScript file (e.g., `server.js`).
2. Import the `http` module in `server.js`.
3. Use `http.createServer()` to create a web server.
4. Provide a request handler function to `createServer()`. This function should take `req` (request) and `res` (response) objects as parameters.
5. Inside the request handler:
   - Set the response header using `res.writeHead(200, {'Content-Type': 'text/html'})`.
   - Send an HTML response body using `res.end('<h1>Hello from Node.js Server!</h1>')`.
6. Make the server listen on a specific port (e.g., 3000) using `server.listen(3000)`.
7. Add a `console.log()` message to indicate that the server is running (e.g., `console.log('Server running at http://localhost:3000/')`).
8. Run the server by executing `node server.js` in your terminal.
9. Open a web browser and navigate to `http://localhost:3000/`. You should see the "Hello from Node.js Server!" message.
10. Refer to the provided link for detailed information about creating a web server in Node.js: https://infyspringboard.onwingspan.com/web/viewer/web-module/lex_28177338996267815000_shared?collectionId=lex_32407835671946760000_shared&collectionType=Course

Example Code Snippet (server.js):

```
JavaScript
// server.js
const http = require('http');

const server = http.createServer((req, res) => {
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.end('<h1>Hello from Node.js Server!</h1>');
});

const port = 3000;
server.listen(port, () => {
  console.log(`Server running at http://localhost:${port}/`);
});
```

Expected Output:

- Running `node server.js` should start a Node.js server.
- Accessing `http://localhost:3000/` in a browser should display the "Hello from Node.js Server!" message.
- The console should log "Server running at http://localhost:3000/".

Observations:

Experiment 6.c: Modular Programming in Node.js

Course Name: Node.js **Module Name:** Modular programming in Node.js

Objective: To demonstrate modular programming in Node.js by creating and using modules to organize application workflow.

Procedure:

1. Create two JavaScript files:
   - `calculator.js` (module file): Define functions for basic calculations (add, subtract, multiply, divide) and export these functions as a module using `module.exports`.
   -
   - `app.js` (main application file): Import the `calculator` module using `require()`. Use the functions from the `calculator` module in `app.js` to perform calculations and display results using `console.log()`.
2. Open your terminal, navigate to the directory containing `calculator.js` and `app.js`.
3. Run the main application using `node app.js`.
4. Observe the output in the console, which should show the results of calculations performed using the modularized functions.

5. Refer to the provided link for detailed information about modular programming in Node.js: https://infyspringboard.onwingspan.com/web/viewer/web-module/lex_28865394191004004000_shared?collectionId=lex_32407835671946760000_shared&collectionType=Course

Example Code Snippet (calculator.js - module):

```
JavaScript
// calculator.js
exports.add = (a, b) => a + b;
exports.subtract = (a, b) => a - b;
exports.multiply = (a, b) => a * b;
exports.divide = (a, b) => a / b;
```

Example Code Snippet (app.js - main application):

```
JavaScript
// app.js
const calculator = require('./calculator'); // Import the calculator module

let num1 = 10;
let num2 = 5;

console.log(`Addition: ${calculator.add(num1, num2)}`);
console.log(`Subtraction: ${calculator.subtract(num1, num2)}`);
console.log(`Multiplication: ${calculator.multiply(num1, num2)}`);
console.log(`Division: ${calculator.divide(num1, num2)}`);
```

Expected Output:

- Running `node app.js` should execute the `app.js` script, which imports and uses the `calculator.js` module to perform calculations.
- The console output should display the results of addition, subtraction, multiplication, and division as performed by the functions from the `calculator` module.

Observations:

Experiment 6.d: Restarting Node Application

Course Name: Node.js **Module Name:** Restarting Node Application

Objective: To observe the workflow of restarting a Node.js application, understanding how changes are applied and the server is refreshed.

Procedure:

1. Use the `server.js` file from Experiment 6.b (or create a new simple server file).
2. Start the server by running `node server.js` in the terminal. Keep the server running.
3. Open `http://localhost:3000/` in a web browser and verify it's working.
4.
5. Edit `server.js`: Change the response message in `res.end()` to something different (e.g., '<h1>Updated message from Node.js Server!</h1>'). Save the file.
6. Refresh the page in your web browser at `http://localhost:3000/`. You will notice that the change is **not** immediately reflected. This is because the Node.js server needs to be restarted to pick up code changes.
7. Stop the currently running Node.js server in the terminal (usually by pressing `Ctrl+C`).
8. Restart the server by running `node server.js` again.
9. Refresh the page in your web browser at `http://localhost:3000/` again. Now you should see the updated message.
10. Optionally, explore tools like `nodemon` which can automatically restart the Node.js server on file changes, streamlining development workflow. Install nodemon using `npm install -g nodemon` and run your server using `nodemon server.js`. Repeat steps 4 and 5 with `nodemon` running and observe the difference.
11. Refer to the provided link for detailed information about restarting Node.js applications: https://infyspringboard.onwingspan.com/web/viewer/web-module/lex_9174073856000159000_shared?collectionId=lex_32407835671946760000_shared&collectionType=Course

Example Code Snippet (server.js - initial version, from 6.b):

JavaScript
```
// server.js (Initial version)
const http = require('http');

const server = http.createServer((req, res) => {
   res.writeHead(200, {'Content-Type': 'text/html'});
   res.end('<h1>Hello from Node.js Server!</h1>');
});

const port = 3000;
server.listen(port, () => {
    console.log(`Server running at http://localhost:${port}/`);
});
```

Example Code Snippet (server.js - updated version, for step 4):

```
JavaScript
// server.js (Updated version - changed response message)
const http = require('http');

const server = http.createServer((req, res) => {
    res.writeHead(200, {'Content-Type': 'text/html'});
    res.end('<h1>Updated message from Node.js Server!</h1>'); // Message changed here
});

const port = 3000;
server.listen(port, () => {
    console.log(`Server running at http://localhost:${port}/`);
});
```

Expected Output:

- Initially, accessing `http://localhost:3000/` will show the original message.
- After editing `server.js` and restarting the server, refreshing the browser should display the updated message.
- Using `nodemon server.js` should automatically restart the server and reflect changes upon saving the `server.js` file.
-

Observations:

Experiment 6.e: File Operations

Course Name: Node.js **Module Name:** File Operations

Objective: To perform basic file operations in Node.js, specifically creating a text file and writing data into it using Node.js file system module.

Procedure:

1. Create a new JavaScript file (e.g., `file_operation.js`).
2.
3. Import the `fs` (file system) module in `file_operation.js`.
4. Define the file path and name for the text file to be created (e.g., `src.txt`).
5. Prepare the data to be written to the file (e.g., "Mongo, Express, Angular, Node.").
6. Use `fs.writeFile()` method to create and write data to `src.txt`. Provide the file path, data, and a callback function to handle errors or success.
7. In the callback function, check for errors. If there's an error, log the error to the console. If successful, log a success message (e.g., "Data written to file successfully.").
8. Execute the script using `node file_operation.js` in your terminal.
9. After execution, check if `src.txt` file is created in the same directory and if it contains the specified data.
10. Refer to the provided link for detailed information about file operations in Node.js: https://infyspringboard.onwingspan.com/web/viewer/web-module/lex_3337644018024610000_shared?collectionId=lex_32407835671946760000_shared&collectionType=Course

Example Code Snippet (file_operation.js):

```
JavaScript
// file_operation.js
const fs = require('fs');

const filePath = 'src.txt';
const dataToWrite = "Mongo, Express, Angular, Node.";

fs.writeFile(filePath, dataToWrite, (err) => {
  if (err) {
    console.error("Error writing to file:", err);
  } else {
    console.log("Data written to file successfully.");
  }
});

console.log("Writing data to file..."); // This will likely execute before the callback
```

Expected Output:

- Running `node file_operation.js` should execute the script.
- A file named `src.txt` should be created in the same directory (or path specified).
- `src.txt` should contain the text "Mongo, Express, Angular, Node.".

- The console should display "Data written to file successfully." (if successful) or an error message if writing fails.

Observations:

5. Express.js Exercises

Tools Required:

- Web Browser (Chrome, Firefox, etc.)
- Text Editor (VS Code, Sublime Text, Atom, Notepad++, etc.)
- Node.js and NPM (Node Package Manager)
- Command Prompt/Terminal
- MongoDB (Optional, for some exercises)

Experiment 7.a: Routing in Express.js

Course Name: Express.js **Module Name:** Defining a route, Handling Routes, Route Parameters, Query Parameters

Objective: To implement routing in an Express.js application for AdventureTrails, focusing on defining routes and handling route and query parameters.

Procedure:

1. Create a new directory for your Express.js application (e.g., `adventure-trails`).
2. Inside this directory, initialize a Node.js project using `npm init -y`.
3. Install Express.js: `npm install express`.
4. Create a file `route.js` inside a new directory named `routes` (create the `routes` directory first if it doesn't exist).
5. In `routes/route.js`, implement routing for the AdventureTrails application as described in the provided link. This typically involves:
   - Importing Express Router: `const express = require('express'); const router = express.Router();`.
   - Defining routes using `router.get()`, `router.post()`, etc., for different paths (e.g., `/trails`, `/trails/:trailId`, `/search?location=...`).
   - Handling route parameters (e.g., `:trailId`) and query parameters (e.g., `?location=`) within the route handlers (callback functions).
   - Sending responses using `res.send()`, `res.json()`, `res.render()`, etc.

- Exporting the router: `module.exports = router;`.

6. In your main application file (e.g., `app.js` in the project root), import Express: `const express = require('express'); const app = express();`.
7. Mount the routes defined in `routes/route.js` in your main app: `const trailRoutes = require('./routes/route'); app.use('/', trailRoutes);`.
8. Start the Express server and listen on a port (e.g., 3000).
9. Test your routes using a web browser or tools like `curl` or Postman, accessing different paths and providing route/query parameters as defined in `routes/route.js`.
10. Refer to the provided link for detailed information about routing in Express.js: https://infyspringboard.onwingspan.com/web/viewer/web-module/lex_29394215542149950000_shared?collectionId=lex_32407835671946760000_shared&collectionType=Course

Example Code Snippet (routes/route.js - example routes):

JavaScript
```
// routes/route.js
const express = require('express');
const router = express.Router();

// Route for homepage
router.get('/', (req, res) => {
  res.send('Welcome to AdventureTrails!');
});

// Route for listing all trails
router.get('/trails', (req, res) => {
  res.json({ message: 'List of all adventure trails will be here' });
});

// Route with route parameter for a specific trail
router.get('/trails/:trailId', (req, res) => {
  const trailId = req.params.trailId;
  res.send(`Details for trail ID: ${trailId}`);
});

// Route with query parameters for searching trails
router.get('/search', (req, res) => {
  const location = req.query.location;
  const activity = req.query.activity;
  res.send(`Search results for location: ${location}, activity: ${activity}`);
});

module.exports = router;
```

Example Code Snippet (app.js - main application):

```
JavaScript
// app.js
const express = require('express');
const app = express();
const trailRoutes = require('./routes/route'); // Import route.js

app.use('/', trailRoutes); // Mount routes

const port = 3000;
app.listen(port, () => {
   console.log(`Server running at http://localhost:${port}`);
});
```

Expected Output:

- Running `node app.js` should start the Express server.
- Accessing different URLs like `http://localhost:3000/`, `http://localhost:3000/trails`, `http://localhost:3000/trails/123`, `http://localhost:3000/search?location=Mountains&activity=Hiking` in a browser or using `curl` should trigger the corresponding routes defined in `routes/route.js` and display the appropriate responses.

Observations:

Experiment 7.b: Middleware in Express.js

Course Name: Express.js **Module Name:** How Middleware works, Chaining of Middlewares, Types of Middlewares

Objective: To implement middleware functions in an Express.js application (myNotes) to handle POST submissions, customize error messages, and perform request logging, demonstrating middleware functionality and chaining.

Procedure:

1. Create a new Express.js application (or use an existing one).
2. Implement the following middleware functions in your `app.js` (or a separate middleware file):
   - **Parsing Middleware (for POST requests):** Use `express.urlencoded({ extended: true })` middleware to parse URL-encoded request bodies from POST requests.
   - **Error Handling Middleware:** Create a custom error handling middleware function that takes `err, req, res, next` parameters. In this middleware, customize error messages based on error type or status, and send an appropriate error response (e.g., res.status(500).send('Custom error message')). Place this middleware after your route handlers.
   - **Logging Middleware:** Create a custom middleware function to log incoming requests. This middleware should log the request method, URL, and timestamp to the console. This should be placed before your route handlers so it logs every incoming request.
3. Chain these middleware functions in your Express app using `app.use()`. Ensure the order is logical (logging first, then parsing, then route handling, then error handling last).
4. Define a route that handles POST requests (e.g., a route for creating a note in "myNotes" application).
5. Simulate scenarios to test your middleware:
   - Send a POST request to your defined route with URL-encoded data to test the parsing middleware.
   - In your route handler, intentionally throw an error (e.g., `next(new Error('Something went wrong'))`) to trigger your error handling middleware.
   - Observe the console logs to verify that the logging middleware is working for every request.
6. Refer to the provided link for detailed information about middleware in Express.js: https://infyspringboard.onwingspan.com/web/viewer/web-module/lex_13930661312009580000_shared?collectionId=lex_32407835671946760000_shared&collectionType=Course

Example Code Snippet (app.js - middleware implementation):

```
JavaScript
// app.js
const express = require('express');
const app = express();

// 1. Logging Middleware
```

```
const requestLogger = (req, res, next) => {
   console.log(`[${new Date().toISOString()}] ${req.method} ${req.url}`);
   next(); // Pass control to the next middleware or route handler
};

// 2. Parsing Middleware for POST requests
app.use(express.urlencoded({ extended: true })); // Parse URL-encoded bodies

// Use logging middleware - should be before route handlers
app.use(requestLogger);

// Route to handle POST requests (example for myNotes app - create note)
app.post('/notes', (req, res, next) => {
   // Assume you're supposed to create a note from req.body
   if (!req.body.content) {
      return next(new Error('Note content is required')); // Trigger error middleware
   }
   // Process note creation here (e.g., save to database)
   res.send('Note created successfully!');
});

// Route for GET request (just for testing logging middleware)
app.get('/', (req, res) => {
   res.send('GET request to homepage');
});


// 3. Error Handling Middleware - placed after route handlers
app.use((err, req, res, next) => {
   console.error(err.stack); // Log error stack trace for debugging
   res.status(500).send(`Custom Error: ${err.message}`); // Send custom error response
});

const port = 3000;
app.listen(port, () => {
   console.log(`Server running at http://localhost:${port}`);
});
```

Expected Output:

- When the Express app runs:

- Every incoming request (GET or POST) should be logged in the console by the logging middleware, showing timestamp, method, and URL.
- POST requests to `/notes` with URL-encoded data should be processed (though in this example, it just sends a success message if content is provided).
- If you send a POST request to `/notes` without 'content' in the body, or if the `/notes` route intentionally calls `next(error)`, the error handling middleware should be triggered, and a custom error message ("Custom Error: Note content is required" or "Custom Error: Something went wrong" if you change the error message) should be sent as a 500 status response.

Observations:

Experiment 7.c: Connecting to MongoDB with Mongoose Schema

Course Name: Express.js **Module Name:** Connecting to MongoDB with Mongoose, Validation Types and Defaults

Objective: To write a Mongoose schema in an Express.js application to define the structure and validation rules for data that will be stored in MongoDB.

Procedure:

1. Ensure you have MongoDB installed and running.
2. In your Express.js application directory (e.g., `myNotes-app`), install Mongoose: `npm install mongoose`.
3. Create a new directory (e.g., `models`) and inside it create a file for your schema (e.g., `note.model.js`).
4. In `note.model.js`, import Mongoose: `const mongoose = require('mongoose');`.
5. Define a Mongoose schema using `new mongoose.Schema({...})`. For "myNotes" application, consider fields like `title` (String, required), `content` (String, required), `createdAt` (Date, default to current date), `updatedAt` (Date).
6. Apply validation rules to your schema fields using Mongoose schema types and validators (e.g., `required: true`, `type: String`, `minlength`, `maxlength`, `enum`, custom validators).
7. Define default values if needed (e.g., for `createdAt` and `updatedAt`, you can use `default: Date.now`).
8. Export the schema: `module.exports = noteSchema;`.

In your main application file (`app.js`), connect to MongoDB using Mongoose:

```
 JavaScript
mongoose.connect('mongodb://localhost:27017/mynotesdb', { // Replace 'mynotesdb' with your
DB name
    useNewUrlParser: true,
    useUnifiedTopology: true
}).then(() => console.log('Connected to MongoDB'))
  .catch(err => console.error('MongoDB connection error:', err));
```

9.
10. In `app.js`, import the schema you just created: `const noteSchema = require('./models/note.model');`. (Though for the next step, you'll wrap this schema in a model).
11. Run your `app.js` using `node app.js`. Check the console output to verify successful connection to MongoDB. You are not yet creating or saving data in this step, just setting up the schema and connection.
12. Refer to the provided link for detailed information about Mongoose Schemas and MongoDB connection: https://infyspringboard.onwingspan.com/web/viewer/web-module/lex_auth_0130355887754854406 91_shared?collectionId=lex_32407835671946760000_shared&collectionType=Course

Example Code Snippet (models/note.model.js - Mongoose schema):

```
JavaScript
// models/note.model.js
const mongoose = require('mongoose');

const noteSchema = new mongoose.Schema({
    title: {
        type: String,
        required: true,
        trim: true, // Removes whitespace from both ends of a string
        maxlength: 100
    },
    content: {
        type: String,
        required: true
    },
    createdAt: {
        type: Date,
        default: Date.now
    },
    updatedAt: Date // No default, will be updated on modifications
});

module.exports = noteSchema;
```

Example Code Snippet (app.js - connecting to MongoDB and using schema):

```javascript
JavaScript
// app.js
const express = require('express');
const mongoose = require('mongoose');
const noteSchema = require('./models/note.model'); // Import the schema

const app = express();

mongoose.connect('mongodb://localhost:27017/mynotesdb', {
    useNewUrlParser: true,
    useUnifiedTopology: true
}).then(() => console.log('Connected to MongoDB'))
  .catch(err => console.error('MongoDB connection error:', err));


// ... rest of your Express app (middleware, routes will be added later) ...

const port = 3000;
app.listen(port, () => {
    console.log(`Server running at http://localhost:${port}`);
});
```

Expected Output:

- Running `node app.js` should attempt to connect to MongoDB.
- If the connection is successful, you should see "Connected to MongoDB" logged in the console.
- If there's a connection error, an error message will be logged.
- The `noteSchema` should be defined in `note.model.js` with the specified fields, types, validation rules, and defaults.

Observations:

Experiment 7.d: Mongoose Models

Course Name: Express.js **Module Name:** Models

Objective: To wrap the Mongoose schema created in Experiment 7.c into a Mongoose Model object, making it possible to interact with the MongoDB collection defined by the schema.

Procedure:

1. Open `models/note.model.js`.
2. After defining the `noteSchema` (from Experiment 7.c), create a Mongoose Model using `mongoose.model('Note', noteSchema);`. The first argument 'Note' is the *singular* name of the collection your model is for (Mongoose will look for the plural, lowercase version in MongoDB, i.e., 'notes' collection).
3. Replace `module.exports = noteSchema;` with `module.exports = Note;` to export the Model instead of just the Schema.
4. In your `app.js`, import the `Note` model: `const Note = require('./models/note.model');`.
5. To verify model creation (without actually performing database operations yet), you can log the `Note` model to the console in `app.js`: `console.log("Note Model:", Note);`.
6. Run `node app.js`. Check the console output. You should see information about the `Note` model logged, indicating that the model has been successfully created from your schema.
7. Refer to the provided link for detailed information about Mongoose Models: https://infyspringboard.onwingspan.com/web/viewer/web-module/lex_auth_01303559389686988862_shared?collectionId=lex_32407835671946760000_shared&collectionType=Course

Example Code Snippet (models/note.model.js - creating and exporting Model):

```
JavaScript
// models/note.model.js
const mongoose = require('mongoose');

const noteSchema = new mongoose.Schema({ /* ... schema definition from 7.c ... */ });

const Note = mongoose.model('Note', noteSchema); // Create Model from schema

module.exports = Note; // Export the Model
```

Example Code Snippet (app.js - importing and logging the Model):

```
JavaScript
// app.js
const express = require('express');
const mongoose = require('mongoose');
const Note = require('./models/note.model'); // Import the Note Model

const app = express();

mongoose.connect('mongodb://localhost:27017/mynotesdb', { /* ... connection code from 7.c ...
*/ });

console.log("Note Model:", Note); // Log the Model to console

// ... rest of your Express app ...

const port = 3000;
app.listen(port, () => {
   console.log(`Server running at http://localhost:${port}`);
});
```

Expected Output:

- Running `node app.js` should still connect to MongoDB (if connection code is retained from 7.c).
- The console output should now include "Note Model: function model..." followed by details of the Mongoose Model object, confirming successful model creation.

Observations:

Experiment 8.a: CRUD Operations using Mongoose

Course Name: Express.js **Module Name:** CRUD Operations

Objective: To perform Create, Read, Update, and Delete (CRUD) operations on MongoDB using Mongoose library functions within an Express.js application.

Procedure:

1. Ensure you have MongoDB connected and the `Note` model from Experiment 7.d is set up.
2. In your `app.js` or a separate test file, use the `Note` model to perform CRUD operations:
   - **Create (Create):** Use `Note.create({ title: 'Test Note', content: 'This is a test note' })` to create a new note document. Handle the promise returned by `create()` to log success or error.
   - **Read (Read - Find all):** Use `Note.find({})` to retrieve all notes from the database. Use `.then()` to access the results and log them to the console.
   - **Read (Read - Find one by ID):** Use `Note.findById('someNoteId')` (replace `'someNoteId'` with an actual ID from your database) to find a specific note. Handle the promise to log the found note or error.
   - **Update (Update):** Use `Note.findByIdAndUpdate('someNoteId', { title: 'Updated Title', content: 'Updated content' }, { new: true })` to update a note. `{ new: true }` option returns the modified document. Handle the promise to log the updated note or error.
   - **Delete (Delete):** Use `Note.findByIdAndDelete('someNoteId')` to delete a note by ID. Handle the promise to log success or error.
3.
4. Run your script (e.g., `node app.js` or `node test_crud.js`).
5. Check the console output to verify the results of each CRUD operation (success messages, retrieved data, updated data, etc.). You can also use a MongoDB client (like MongoDB Compass or command-line `mongo shell`) to directly inspect your database and confirm the changes.
6. Refer to the provided link for detailed information about CRUD operations with Mongoose: https://infyspringboard.onwingspan.com/web/viewer/web-module/lex_auth_013035684270129152696_shared?collectionId=lex_32407835671946760000_shared&collectionType=Course

Example Code Snippet (crud_operations.js - separate file for CRUD testing):

JavaScript
```
// crud_operations.js
const mongoose = require('mongoose');
const Note = require('./models/note.model'); // Assuming note.model.js is in the same directory

mongoose.connect('mongodb://localhost:27017/mynotesdb', {
  useNewUrlParser: true,
  useUnifiedTopology: true
}).then(() => {
  console.log('Connected to MongoDB for CRUD operations testing');
  runCRUDOperations(); // Call function to perform CRUD after successful connection
}).catch(err => console.error('MongoDB connection error:', err));

async function runCRUDOperations() {
  try {
    // Create Operation
    const createdNote = await Note.create({ title: 'Test Note', content: 'This is a test note
created via Mongoose' });
    console.log('Note Created:', createdNote);
    const noteIdForUpdateAndDelete = createdNote._id; // Save ID for further operations

    // Read Operation - Find all notes
    const allNotes = await Note.find({});
    console.log('All Notes:', allNotes);

    // Read Operation - Find one note by ID
    const foundNote = await Note.findById(noteIdForUpdateAndDelete);
    console.log('Found Note by ID:', foundNote);

    // Update Operation
    const updatedNote = await Note.findByIdAndUpdate(noteIdForUpdateAndDelete, { title:
'Updated Title', content: 'Content updated!' }, { new: true });
    console.log('Updated Note:', updatedNote);

    // Delete Operation
    await Note.findByIdAndDelete(noteIdForUpdateAndDelete);
    console.log('Note Deleted successfully.');

    // Read Operation - Verify deletion (should be empty or different)
    const notesAfterDeletion = await Note.find({});
    console.log('Notes after deletion:', notesAfterDeletion);


  } catch (error) {
    console.error('CRUD Operation Error:', error);
  } finally {
    mongoose.disconnect(); // Disconnect after operations are done
    console.log('Disconnected from MongoDB.');
  }
}
```

Note: You need to replace `'someNoteId'` placeholders in the example code with actual valid `_id` values of documents in your 'notes' collection if you intend to test update and delete operations on existing data. Otherwise, the create operation in this example sets up an ID (`noteIdForUpdateAndDelete`) for immediate update and delete operations.

Expected Output:

- Running `node crud_operations.js` should connect to MongoDB, perform all specified CRUD operations, and log the results of each operation to the console (created note details, list of notes, found note, updated note, deletion success, etc.).
- Inspecting the MongoDB 'notes' collection (using a MongoDB client) should reflect the changes made by these CRUD operations (newly created note, updated note, and deleted note).

Observations:

Experiment 8.b: API Development

Course Name: Express.js **Module Name:** API Development

Objective: To develop APIs in the myNotes application using Express.js to fetch and update note details based on a `notesID` provided in the URL, and to test these APIs.

Procedure:

1. Continue with your "myNotes" Express.js application setup (with Mongoose and `Note` model from previous experiments).
2. Define two API endpoints in your `routes/route.js` file:
   - **GET `/notes/:notesID`:** This API should fetch and return details of a single note based on the `notesID` path parameter.
     - Use `Note.findById(req.params.notesID)` to fetch the note from MongoDB.
     - Handle cases where the note is found and not found. If found, send the note data as JSON response (`res.json(note)`). If not found, send a 404 status with a message (e.g., `res.status(404).json({ message: 'Note not found' })`).
   -
   - **PUT `/notes/:notesID`:** This API should update the details of an existing note based on the `notesID` and data provided in the request body.

- Use `Note.findByIdAndUpdate(req.params.notesID, req.body, { new: true, runValidators: true })` to update the note. `{ new: true }` returns the updated document, `runValidators: true` ensures validation rules from your schema are applied during update.
- Handle cases where the note is found and updated, or not found. If updated successfully, send the updated note as JSON (`res.json(updatedNote)`). If not found, or update fails validation, send appropriate error responses (e.g., 404 if not found, 400 for validation errors).

○

3.
4. In `app.js`, make sure you are using `express.json()` middleware to parse JSON request bodies for PUT requests.
5. Test these APIs using tools like Postman, `curl`, or a simple HTML form:
   - **Test GET API:** Send a GET request to `http://localhost:3000/notes/7555` (replace `7555` with an actual note ID in your database, or create a test note first and use its ID). Verify that you receive the note details in JSON format if the note exists, or a 404 error if it doesn't.
   - **Test PUT API:** Send a PUT request to `http://localhost:3000/notes/7555` (again, replace `7555` with a valid note ID). In the request body, send JSON data with fields to update (e.g., `{ "title": "Updated Title from API", "content": "Updated content via PUT API" }`). Verify that the note is updated in MongoDB and the API responds with the updated note in JSON format. Test with invalid data to check validation errors.
6.
7. Refer to the provided link for detailed information about API development in Express.js: https://infyspringboard.onwingspan.com/web/viewer/web-module/lex_auth_01303574525097574475_shared?collectionId=lex_32407835671946760000_shared&collectionType=Course

Example Code Snippet (routes/route.js - API routes):

JavaScript
```
// routes/route.js
const express = require('express');
const router = express.Router();
const Note = require('../models/note.model'); // Assuming note.model.js is in the models dir,
relative to routes
```

```
// GET API to fetch note by ID
router.get('/notes/:notesID', async (req, res) => {
  try {
    const note = await Note.findById(req.params.notesID);
    if (!note) {
      return res.status(404).json({ message: 'Note not found' });
    }
    res.json(note);
  } catch (error) {
    res.status(500).json({ message: 'Error fetching note', error: error.message });
  }
});

// PUT API to update note by ID
router.put('/notes/:notesID', async (req, res) => {
  try {
    const updatedNote = await Note.findByIdAndUpdate(req.params.notesID, req.body, { new:
true, runValidators: true });
    if (!updatedNote) {
      return res.status(404).json({ message: 'Note not found for update' });
    }
    res.json(updatedNote);
  } catch (error) {
    if (error.name === 'ValidationError') {
      return res.status(400).json({ message: 'Validation error', errors: error.errors });
    }
    res.status(500).json({ message: 'Error updating note', error: error.message });
  }
});


module.exports = router;
```

Example Code Snippet (app.js - including JSON parsing middleware):

JavaScript
```
// app.js
const express = require('express');
const mongoose = require('mongoose');
const noteRoutes = require('./routes/route'); // Import route.js
const app = express();

app.use(express.json()); // Middleware to parse JSON request bodies

app.use('/', noteRoutes); // Mount API routes

mongoose.connect('mongodb://localhost:27017/mynotesdb', { /* ... connection code from 7.c ...
*/ });


const port = 3000;
app.listen(port, () => {
  console.log(`Server running at http://localhost:${port}`);
});
```

Expected Output:

- Running the Express app and testing the GET API (e.g., `http://localhost:3000/notes/someValidNoteId`) should return note details in JSON if a note with `someValidNoteId` exists. For an invalid ID, it should return a 404 JSON response.
-
- Testing the PUT API (e.g., sending a PUT request to `http://localhost:3000/notes/someValidNoteId` with JSON body `{ "title": "Updated Title" }`) should update the corresponding note in MongoDB and return the updated note in JSON, or a 404 if note not found, or 400 for validation errors.

Observations:

Experiment 8.c: Session Management with Cookies

Course Name: Express.js **Module Name:** Why Session management, Cookies

Objective: To implement session management in an Express.js application using cookies to track user sessions.

Procedure:

1. In your Express.js application, install `cookie-parser` middleware: `npm install cookie-parser`.
2. In `app.js`, import and use `cookie-parser` middleware: `const cookieParser = require('cookie-parser'); app.use(cookieParser());`.
3. Define a route (e.g., `/setcookie`) to set a cookie. In the route handler:
   - Use `res.cookie('sessionID', 'uniqueSessionValue', { httpOnly: true, maxAge: 3600000 });` to set a cookie named 'sessionID' with a value, `httpOnly` flag for security, and `maxAge` (e.g., 1 hour in milliseconds).
   - Send a response to indicate that the cookie has been set (e.g., `res.send('Cookie set!')`).
4.
5. Define another route (e.g., `/getcookie`) to retrieve and display the cookie. In the route handler:
   - Access cookies using `req.cookies.sessionID`.
   - Send a response displaying the cookie value or a message if the cookie is not found (e.g., `res.send('Cookie value: ' + req.cookies.sessionID || 'No cookie found!');`).
6.
7. Test these routes in a browser or using `curl`:
   - Access `/setcookie`. It should set the cookie in your browser and display "Cookie set!".
   - Access `/getcookie`. It should retrieve and display the value of the 'sessionID' cookie that was set. If you access `/getcookie` before setting the cookie, it should display "No cookie found!".
   - Inspect browser's developer tools (Application -> Cookies) to see the cookie details after accessing `/setcookie`.
8.
9. Refer to the provided link for detailed information about session management and cookies in Express.js: https://infyspringboard.onwingspan.com/web/viewer/web-module/lex_24299316914857090000_shared?collectionId=lex_32407835671946760000_shared&collectionType=Course

Example Code Snippet (app.js - cookie-based session management):

```
JavaScript
// app.js
const express = require('express');
const cookieParser = require('cookie-parser');
const app = express();
app.use(cookieParser()); // Use cookie-parser middleware

// Route to set a cookie
app.get('/setcookie', (req, res) => {
  res.cookie('sessionID', 'uniqueSessionValue123', { httpOnly: true, maxAge: 3600000 }); //
Cookie for 1 hour
  res.send('Cookie set!');
});

// Route to get and display cookie
app.get('/getcookie', (req, res) => {
  res.send('Cookie value: ' + (req.cookies.sessionID || 'No cookie found!'));
});

const port = 3000;
app.listen(port, () => {
  console.log(`Server running at http://localhost:${port}`);
});
```

Expected Output:

- Accessing `/setcookie` should set a cookie named `sessionID` in your browser.
- Accessing `/getcookie` immediately after `/setcookie` should display the value of the `sessionID` cookie.
- If you haven't visited `/setcookie`, accessing `/getcookie` should display "No cookie found!".
- Browser's developer tools should show the set cookie under Application -> Cookies.

Observations:

Experiment 8.d: Session Management with Sessions

Course Name: Express.js **Module Name:** Sessions

Objective: To implement session management in an Express.js application using sessions to manage user-specific data on the server-side.

Procedure:

1. In your Express.js application, install `express-session` middleware: `npm install express-session`.

In `app.js`, import and configure `express-session` middleware:

```
 JavaScript
const session = require('express-session');
app.use(session({
   secret: 'your-secret-key', // Replace with a strong, random secret key
   resave: false,
   saveUninitialized: true,
   cookie: { httpOnly: true, maxAge: 3600000 } // Session cookie settings (optional, but good to
set httpOnly and maxAge)
}));
```

2. **Important:** Replace `'your-secret-key'` with a real, randomly generated secret key for security in a production application.
3. Define routes to interact with session data:
   - **`/setsession`:** To set session data. In the route handler:
     - Set a session variable (e.g., `req.session.username = 'testUser';`).
     - Send a response indicating session data is set (e.g., `res.send('Session data set!');`).
   -
   - **`/getsession`:** To retrieve and display session data. In the route handler:
     - Access session data using `req.session.username`.
     - Send a response displaying the session data or a message if not set (e.g., `res.send('Username from session: ' + req.session.username || 'No session username found!');`).
   -
   - **`/destroysession`:** To destroy the session. In the route handler:
     - Use `req.session.destroy((err) => { ... });` to destroy the session. In the callback, handle errors or send a success response (e.g., `res.send('Session destroyed!');`).
   -
4.
5. Test these routes in a browser:
   - Access `/setsession`. It should set session data and display "Session data set!".

- Access `/getsession`. It should display the username from the session, e.g., "Username from session: testUser".
- Access `/destroysession`. It should destroy the session and display "Session destroyed!".
- Access `/getsession` again after destroying the session. It should now display "No session username found!".

6.
7. Refer to the provided link for detailed information about session management using sessions in Express.js: https://infyspringboard.onwingspan.com/web/viewer/web-module/lex_905413034723449100_shared?collectionId=lex_32407835671946760000_shared&collectionType=Course

Example Code Snippet (app.js - session-based session management):

```
JavaScript
// app.js
const express = require('express');
const session = require('express-session');

const app = express();

app.use(session({
  secret: 'your-secret-key-should-be-long-and-random', // Replace with a strong secret
  resave: false,
  saveUninitialized: true,
  cookie: { httpOnly: true, maxAge: 3600000 } // 1 hour cookie lifespan
}));

// Route to set session data
app.get('/setsession', (req, res) => {
  req.session.username = 'testUser';
  res.send('Session data set!');
});

// Route to get session data
app.get('/getsession', (req, res) => {
  res.send('Username from session: ' + (req.session.username || 'No session username
found!'));
});

// Route to destroy session
app.get('/destroysession', (req, res) => {
  req.session.destroy((err) => {
        if (err) {
            console.error('Error destroying session:', err);
            res.send('Error destroying session.');
        } else {
            res.send('Session destroyed!');
        }
    });
});


const port = 3000;
app.listen(port, () => {
    console.log(`Server running at http://localhost:${port}`);
});
```

Expected Output:

- Accessing `/setsession` sets the username in the session.
- Accessing `/getsession` retrieves and displays the username from the session.
- Accessing `/destroysession` destroys the session.
- After session destruction, accessing `/getsession` will indicate that no session username is found.
- Session data is maintained server-side (not visible directly in browser cookies like in Experiment 8.c, although a session cookie ID is usually used behind the scenes).

Observations:

Experiment 8.e: Security with Helmet Middleware

Course Name: Express.js **Module Name:** Why and What Security, Helmet Middleware

Objective: To implement basic security features in the myNotes application using Helmet middleware to secure HTTP headers.

Procedure:

1. In your Express.js application, install `helmet` middleware: `npm install helmet`.

2. In `app.js`, import and use `helmet` middleware: `const helmet = require('helmet'); app.use(helmet());`. Adding `app.use(helmet())` will enable a collection of security-enhancing middleware functions by default.
3. Optionally, explore specific Helmet middleware functions you can enable or disable individually for finer control, as described in the Helmet documentation (e.g., `helmet.contentSecurityPolicy()`, `helmet.frameguard()`, `helmet.xssFilter()`, etc.). For this basic experiment, using `app.use(helmet())` is sufficient to demonstrate the general principle.
4. Start your Express.js application.
5. Use browser developer tools (Network tab) or tools like `curl -I http://localhost:3000/` to inspect the HTTP headers of your application's responses, both *before* and *after* adding `app.use(helmet())`.
6. Observe the changes in headers like `X-Frame-Options`, `X-Content-Type-Options`, `X-XSS-Protection`, `Strict-Transport-Security`, `Content-Security-Policy`, etc., after enabling Helmet. Helmet typically sets or modifies these headers to enhance security against common web vulnerabilities.
7. Refer to the provided link for detailed information about security and Helmet middleware in Express.js: https://infyspringboard.onwingspan.com/web/viewer/web-module/lex_3167745306117794000_shared?collectionId=lex_32407835671946760000_shared&collectionType=Course

Example Code Snippet (app.js - implementing Helmet middleware):

JavaScript
```
// app.js
const express = require('express');
const helmet = require('helmet');

const app = express();

app.use(helmet()); // Enable Helmet! It will set various security headers

app.get('/', (req, res) => {
  res.send('MyNotes application with Helmet security');
});

const port = 3000;
app.listen(port, () => {
  console.log(`Server running at http://localhost:${port}`);
});
```

To test with `curl` (in terminal):

- **Before Helmet:** `curl -I http://localhost:3000/` (and examine the headers in the output)
- **After Helmet (with `app.use(helmet())` added):** `curl -I http://localhost:3000/` (compare the headers - you should see added or modified security-related headers).

Expected Output:

- Before adding `app.use(helmet())`, inspecting HTTP headers will show default headers set by Express or your server environment.
- After adding `app.use(helmet())`, inspecting HTTP headers will reveal the addition of several security-related headers (like `X-Frame-Options`, `X-Content-Type-Options`, `Strict-Transport-Security`, etc.) set by Helmet middleware, enhancing the application's security posture.

Observations:

6. TypeScript Exercises

Tools Required:

- Web Browser (Chrome, Firefox, etc.)
- Text Editor (VS Code, Sublime Text, Atom, Notepad++, etc.)
- Node.js and NPM (Node Package Manager)
- TypeScript Compiler (`tsc`)
- Command Prompt/Terminal

Experiment 9.a: Basics of TypeScript and String Enums

Course Name: Typescript **Module Name:** Basics of TypeScript

Objective: To demonstrate basic TypeScript concepts and string enums by displaying mobile prices in different colors, using string enum values to represent color names.

Procedure:

1. Create a new directory for your TypeScript exercise (e.g., `typescript-exercises`).

2. Inside this directory, create a TypeScript file (e.g., `price_colors.ts`).
3. Define a string enum named `Color` with members like `GoldPlatinum`, `PinkGold`, `SilverTitanium`, assigning string values like `"gold"`, `"pinkgold"`, `"silver"` to them respectively.
4. Declare a variable `mobilePrice` of type `number` and assign a price value.
5. Write a function `displayPriceInColor(price: number, colorName: Color)` that:
   - Takes a `price` (number) and `colorName` (of type `Color`) as arguments.
   - Uses `console.log()` or DOM manipulation (if you are setting up an HTML file) to display the `price` along with the color name (using the string value from the `Color` enum). You can style the price in the specified color if working with HTML.
6.
7. In your main code, call `displayPriceInColor()` function three times, passing `mobilePrice` and different `Color` enum values (e.g., `Color.GoldPlatinum`, `Color.PinkGold`, `Color.SilverTitanium`).
8. Compile your TypeScript file to JavaScript using `tsc price_colors.ts` in the terminal (make sure you have TypeScript compiler installed: `npm install -g typescript`).
9. Run the generated JavaScript file (e.g., using Node.js: `node price_colors.js` or by including it in an HTML file and opening in browser).
10. Observe the output, which should display the mobile price in different colors (or color names if just logging to console), using the string enum values.
11. Refer to the provided link for detailed information about TypeScript basics and enums: https://infyspringboard.onwingspan.com/web/viewer/web-module/lex_28910354929502245000_shared?collectionId=lex_9436233116512678000_shared&collectionType=Course

Example Code Snippet (price_colors.ts):

```
TypeScript
// price_colors.ts

enum Color {
   GoldPlatinum = "gold",
   PinkGold = "pinkgold",
   SilverTitanium = "silver"
}

let mobilePrice: number = 799.99;
```

```
function displayPriceInColor(price: number, colorName: Color): void {
   console.log(`Mobile Price in ${colorName} color: $${price}`);
   // If using HTML:
   // document.body.innerHTML += `<p style="color: ${colorName};">Price in ${colorName}:
$${price}</p>`;
}

displayPriceInColor(mobilePrice, Color.GoldPlatinum);
displayPriceInColor(mobilePrice, Color.PinkGold);
displayPriceInColor(mobilePrice, Color.SilverTitanium);
```

Expected Output:

- Compiling and running `price_colors.ts` should output to the console (or webpage, if you use HTML output) the mobile price displayed with each of the color names defined in the `Color` enum.

Example console output:

```
 Mobile Price in gold color: $799.99
Mobile Price in pinkgold color: $799.99
Mobile Price in silver color: $799.99
```

-

Observations:

Experiment 9.b: Functions and Arrow Functions

Course Name: Typescript **Module Name:** Function

Objective: To define and use arrow functions in TypeScript within event handlers to filter product arrays based on a selected product ID and pass product data to the next screen (simulated here with console output).

Procedure:

1. Create a new TypeScript file (e.g., `product_filter.ts`).
2. Define an interface `Product` with properties `productId: number`, `productName: string`, and any other relevant product properties.
3. Create an array of `Product` objects named `products` with sample product data.

4. Simulate an event handler scenario. For example, assume you have a function `handleProductSelection(selectedProductId: number)` that is called when a product is selected (in a real application, this would be triggered by a user interface event).
5. Inside `handleProductSelection()`:
   - Define an arrow function `filterProduct` that takes a `product` object (of type `Product`) as a parameter and returns a boolean value indicating if `product.productId` matches the `selectedProductId`.
   - Use the `products.filter(filterProduct)` method to filter the `products` array using the `filterProduct` arrow function. This will return a new array containing only the selected product (or products if multiple match, though productId should ideally be unique).
   - For this exercise, instead of passing to "next screen," just log the filtered product object (or the first element of the filtered array if you expect only one match) to the console.
6.
7. Call `handleProductSelection()` with a sample `productId` to test the filtering logic.
8. Compile and run your TypeScript file.
9. Observe the console output, which should display the details of the product object that matches the `selectedProductId`.
10. Refer to the provided link for detailed information about functions and arrow functions in TypeScript: https://infyspringboard.onwingspan.com/web/viewer/web-module/lex_10783156469383723000_shared?collectionId=lex_9436233116512678000_shared&collectionType=Course

Example Code Snippet (product_filter.ts):

TypeScript

```
// product_filter.ts

interface Product {
  productId: number;
  productName: string;
  price: number;
  category: string;
}

const products: Product[] = [
  { productId: 101, productName: "Laptop", price: 1200, category: "Electronics" },
  { productId: 102, productName: "T-Shirt", price: 25, category: "Clothing" },
  { productId: 103, productName: "Headphones", price: 100, category: "Electronics" }
];
```

```
function handleProductSelection(selectedProductId: number): void {
   const filterProduct = (product: Product) => product.productId === selectedProductId; // Arrow
function

   const selectedProducts = products.filter(filterProduct);

   if (selectedProducts.length > 0) {
      console.log("Selected Product Details:", selectedProducts[0]); // Assuming only one
product will match by ID
   } else {
      console.log(`Product with ID ${selectedProductId} not found.`);
   }
}

handleProductSelection(102); // Test with product ID 102
handleProductSelection(104); // Test with a non-existent product ID
```

Expected Output:

- Compiling and running `product_filter.ts` should output to the console:
    - For `handleProductSelection(102)`, it should display the details of the "T-Shirt" product object.
    - For `handleProductSelection(104)`, it should display "Product with ID 104 not found."
-

Observations:

Experiment 9.c: Parameter Types and Return Types

Course Name: Typescript **Module Name:** Parameter Types and Return Types

Objective: To declare a TypeScript function `getMobileByVendor` that accepts a string (vendor name) as an input parameter and returns a list (array) of mobiles (you can represent mobile as a simple type or interface). Demonstrate type annotations for parameters and return types.

Procedure:

1. Create a new TypeScript file (e.g., `mobile_vendor.ts`).
2. Define an interface `Mobile` with properties like `model: string`, `vendor: string`, `price: number`.
3. Create an array of `Mobile` objects named `mobiles` with sample mobile data from different vendors.
4. Declare a function `getMobileByVendor` in TypeScript.
    - It should accept one parameter named `vendorName` of type `string`.
    - It should be annotated to return an array of `Mobile` objects (`Mobile[]`).
    - Inside the function, use the `mobiles.filter()` method to filter the `mobiles` array and return only the mobiles where the `vendor` property matches the input `vendorName`.
5.
6. Call the `getMobileByVendor` function with different vendor names (e.g., "Samsung", "Apple", "OnePlus").
7. Log the returned list of mobiles for each vendor to the console.
8. Compile and run your TypeScript file.
9. Observe the console output. For each vendor name, it should display the list of `Mobile` objects from that vendor.
10. Refer to the provided link for detailed information about parameter types and return types in TypeScript functions: https://infyspringboard.onwingspan.com/web/viewer/hands-on/lex_auth_0127129124270571529O1_shared?collectionId=lex_9436233116512678000_shared&collectionType=Course

Example Code Snippet (mobile_vendor.ts):

TypeScript
```
// mobile_vendor.ts

interface Mobile {
  model: string;
  vendor: string;
  price: number;
}

const mobiles: Mobile[] = [
  { model: "Galaxy S21", vendor: "Samsung", price: 999 },
  { model: "iPhone 13", vendor: "Apple", price: 1099 },
  { model: "OnePlus 9", vendor: "OnePlus", price: 799 },
  { model: "Galaxy Note 20", vendor: "Samsung", price: 899 },
  { model: "iPhone SE", vendor: "Apple", price: 399 }
];

function getMobileByVendor(vendorName: string): Mobile[] { // Parameter type: string, Return
type: Mobile[]
  return mobiles.filter(mobile => mobile.vendor === vendorName);
}

console.log("Samsung Mobiles:", getMobileByVendor("Samsung"));
console.log("Apple Mobiles:", getMobileByVendor("Apple"));
console.log("OnePlus Mobiles:", getMobileByVendor("OnePlus"));
console.log("Xiaomi Mobiles:", getMobileByVendor("Xiaomi")); // Vendor not in list, should return
empty array
```

Expected Output:

- Compiling and running `mobile_vendor.ts` should output to the console, displaying lists of mobiles for "Samsung", "Apple", "OnePlus" vendors, and an empty array (or empty list representation) for "Xiaomi" as there are no Xiaomi mobiles in the `mobiles` array.

Observations:

Experiment 9.d: Arrow Function with Objects

Course Name: Typescript **Module Name:** Arrow Function

Objective: To declare an arrow function `myfunction` in TypeScript that populates the `id` parameter of a manufacturers array (of objects) whose `price` is greater than or equal to a given price threshold.

Procedure:

1. Create a new TypeScript file (e.g., `manufacturer_price_filter.ts`).
2. Define an interface `Manufacturer` with properties `id: number` and `price: number`.
3. Create an array of `Manufacturer` objects named `manufacturers` with sample data. Initially, you can set the `id` property to `null` or `undefined` in these objects as the goal is to populate them.

4. Declare an arrow function `myfunction` in TypeScript.
    - It should accept two parameters: `manufacturersArray` (array of `Manufacturer` objects) and `priceThreshold` (number).
    - It should iterate through the `manufacturersArray`.
    - For each manufacturer object, it should check if its `price` is greater than or equal to `priceThreshold`.
    - If the price condition is met, it should populate the `id` property of that manufacturer object (e.g., set it to a value based on index or a counter, or any logical ID you want to assign).
    - The function doesn't need to explicitly return anything (void return type is appropriate), as it's modifying the objects in the input array directly.
5.
6. Call `myfunction` with your `manufacturers` array and a `priceThreshold` value (e.g., 50).
7. After calling `myfunction`, log the `manufacturers` array to the console to see the updated `id` properties for manufacturers meeting the price criteria.
8. Compile and run your TypeScript file.
9. Observe the console output. It should show the `manufacturers` array with `id` properties populated for manufacturers whose price is greater than or equal to the `priceThreshold`.
10. Refer to the provided link for detailed information about arrow functions in TypeScript: https://infyspringboard.onwingspan.com/web/viewer/hands-on/lex_auth_01271291087550054490 4_shared?collectionId=lex_9436233116512678000_shared&collectionType=Course

Example Code Snippet (manufacturer_price_filter.ts):

```
TypeScript
// manufacturer_price_filter.ts

interface Manufacturer {
   id: number | null; // Initially null, to be populated
   price: number;
}

const manufacturers: Manufacturer[] = [
   { id: null, price: 40 },
   { id: null, price: 60 },
   { id: null, price: 30 },
   { id: null, price: 70 }
];
```

```
const myfunction = (manufacturersArray: Manufacturer[], priceThreshold: number): void => { // Arrow function
    let currentId = 1; // Counter for assigning IDs
    manufacturersArray.forEach(manufacturer => {
        if (manufacturer.price >= priceThreshold) {
            manufacturer.id = currentId++; // Populate id and increment counter
        }
    });
};

myfunction(manufacturers, 50); // Call the arrow function with manufacturers array and price threshold of 50

console.log("Updated Manufacturers Array:", manufacturers);
```

Expected Output:

- Compiling and running `manufacturer_price_filter.ts` should output to the console.
- The console output will show the `manufacturers` array. In this array, manufacturer objects with `price` >= 50 should have their `id` property populated with a number (starting from 1 and incrementing), while those with `price` < 50 will still have `id: null`.

Observations:

Experiment 9.e: Optional and Default Parameters

Course Name: Typescript **Module Name:** Optional and Default Parameters

Objective: To declare a TypeScript function `getMobileByManufacturer` with optional and default parameters to demonstrate function parameter flexibility. The function should retrieve mobiles by manufacturer, with an optional ID parameter.

Procedure:

1. Create a new TypeScript file (e.g., `mobile_manufacturer_optional.ts`).
2. Reuse the `Mobile` interface and `mobiles` array from Experiment 9.c.

3. Declare a function `getMobileByManufacturer` in TypeScript.
   - It should accept two parameters: `manufacturer` (string, required) and `id` (number, optional). Make `id` parameter optional using `?`.
   - Annotate the function to return an array of `Mobile` objects (`Mobile[]`).
   - Inside the function:
     - If `id` parameter is provided, filter the `mobiles` array to return mobiles matching *both* the `manufacturer` *and* the `id`.
     - If `id` parameter is *not* provided (it's undefined), filter the array to return mobiles matching only the `manufacturer`.
   -
4.
5. Call `getMobileByManufacturer` function in different ways:
   - Invoke it with just the `manufacturer` name (e.g., `getMobileByManufacturer("Samsung")`) - testing the case where the optional parameter is omitted.
   - Invoke it with both `manufacturer` and `id` (e.g., `getMobileByManufacturer("Samsung", 101)`) - testing with the optional parameter provided.
   - Invoke it with a manufacturer that might not exist or an ID that doesn't match for a given manufacturer to test different scenarios.
6.
7. Log the results of each function call to the console.
8. Optionally, demonstrate default parameters. Modify `getMobileByManufacturer` to have a default value for the `id` parameter (e.g., `id: number = null`). Adjust the function logic to handle the default value appropriately (e.g., if `id` is `null`, return all mobiles for the manufacturer, otherwise filter by ID).
9. Compile and run your TypeScript file.
10. Observe the console output for each function call, verifying that it behaves correctly based on whether the optional `id` parameter is provided or not.
11. Refer to the provided link for detailed information about optional and default parameters in TypeScript functions: https://infyspringboard.onwingspan.com/web/viewer/hands-on/lex_auth_012712914940641280906_shared?collectionId=lex_9436233116512678000_shared&collectionType=Course

Example Code Snippet (mobile_manufacturer_optional.ts):

```
TypeScript
// mobile_manufacturer_optional.ts

interface Mobile {
  model: string;
```

```
    vendor: string;
    price: number;
    productId: number; // Added productId for filtering by ID
}

const mobiles: Mobile[] = [
    { productId: 101, model: "Galaxy S21", vendor: "Samsung", price: 999 },
    { productId: 102, model: "iPhone 13", vendor: "Apple", price: 1099 },
    { productId: 103, model: "OnePlus 9", vendor: "OnePlus", price: 799 },
    { productId: 104, model: "Galaxy Note 20", vendor: "Samsung", price: 899 },
    { productId: 105, model: "iPhone SE", vendor: "Apple", price: 399 }
];

function getMobileByManufacturer(manufacturer: string, id?: number): Mobile[] { // 'id' is optional
    if (id !== undefined) {
        return mobiles.filter(mobile => mobile.vendor === manufacturer && mobile.productId ===
id);
    } else {
        return mobiles.filter(mobile => mobile.vendor === manufacturer);
    }
}

console.log("Samsung Mobiles (all):", getMobileByManufacturer("Samsung"));
console.log("Samsung Mobile with ID 104:", getMobileByManufacturer("Samsung", 104));
console.log("Apple Mobiles (all):", getMobileByManufacturer("Apple"));
console.log("OnePlus Mobiles (all):", getMobileByManufacturer("OnePlus"));
console.log("Xiaomi Mobiles (all):", getMobileByManufacturer("Xiaomi")); // No Xiaomi mobiles
console.log("Samsung Mobile with ID 999 (not exist):", getMobileByManufacturer("Samsung",
999)); // ID not existing
```

Expected Output:

- Compiling and running `mobile_manufacturer_optional.ts` should output to the console, showing different lists of mobiles based on the function calls:
    - `getMobileByManufacturer("Samsung")` will list all Samsung mobiles.
    - `getMobileByManufacturer("Samsung", 104)` will list only the Samsung mobile with `productId: 104` (Galaxy Note 20).
    - `getMobileByManufacturer("Apple")` will list all Apple mobiles, and so on.
    - Calls with non-existent manufacturers or IDs will return empty arrays.
-

Observations:

Experiment 10.a: Rest Parameter

Course Name: Typescript **Module Name:** Rest Parameter

Objective: To implement business logic for adding multiple product values into a cart variable (string array) using TypeScript's rest parameters in a function.

Procedure:

1. Create a new TypeScript file (e.g., `product_cart.ts`).
2. Declare a variable `cart` as a string array `string[]` and initialize it as an empty array `[]`. This will represent the shopping cart.
3. Declare a function `addProductToCart` in TypeScript that uses a rest parameter `...products: string[]`.
   - This function should accept a variable number of product names as string arguments using the rest parameter syntax.
   - Inside the function, use a loop (e.g., `for...of` or `forEach`) to iterate through the `products` rest parameter array.
   - For each product name in `products`, add it to the `cart` array using `cart.push(productName)`.
   - Optionally, you can log a message to the console for each product added to the cart.
4.
5. Call the `addProductToCart` function multiple times, passing different numbers of product names as arguments in each call (e.g., `addProductToCart("Laptop", "Mouse");`, `addProductToCart("Keyboard", "Monitor", "Webcam");`, `addProductToCart("Tablet");`).
6. After all calls to `addProductToCart`, log the entire `cart` array to the console to see all added products.
7. Compile and run your TypeScript file.
8. Observe the console output. It should show the messages for each product being added to the cart and then the final content of the `cart` array, containing all products added through different function calls using rest parameters.
9. Refer to the provided link for detailed information about rest parameters in TypeScript: https://infyspringboard.onwingspan.com/web/viewer/hands-on/lex_auth_012712921860915200909_shared?collectionId=lex_9436233116512678000_shared&collectionType=Course

Example Code Snippet (product_cart.ts):

```
TypeScript
// product_cart.ts

let cart: string[] = []; // Shopping cart array

function addProductToCart(...products: string[]): void { // Rest parameter
  for (const product of products) {
    cart.push(product);
    console.log(`Added to cart: ${product}`);
  }
}

addProductToCart("Laptop", "Mouse");
addProductToCart("Keyboard", "Monitor", "Webcam");
addProductToCart("Tablet");

console.log("Final Cart Contents:", cart);
```

Expected Output:

- Compiling and running `product_cart.ts` should output to the console.
- The console output should show messages indicating each product being added to the cart and finally display the `cart` array containing all added products: `["Laptop", "Mouse", "Keyboard", "Monitor", "Webcam", "Tablet"]`.

Observations:

Experiment 10.b: Creating an Interface

Course Name: Typescript **Module Name:** Creating an Interface

Objective: To declare a TypeScript interface named `Product` with properties `productId` (number) and `productName` (string) and implement logic to populate product details conforming to this interface.

Procedure:

1. Create a new TypeScript file (e.g., `product_interface.ts`).

2. Declare an interface named `Product` in TypeScript.
   - Define two properties within the interface: `productId` of type `number` and `productName` of type `string`.
3.
4. Create a function `createProduct` in TypeScript.
   - This function should take two parameters: `id` of type `number` and `name` of type `string`.
   - The function should be annotated to return an object of type `Product`.
   - Inside the function, create an object literal that conforms to the `Product` interface using the provided `id` and `name` parameters for `productId` and `productName` properties respectively. Return this object.
5.
6. Call the `createProduct` function multiple times with different product IDs and names (e.g., `createProduct(1, "Laptop")`, `createProduct(2, "Keyboard")`).
7. Log the returned product objects to the console.
8. Optionally, create an array of type `Product[]` and store the created product objects in this array, then log the array.
9. Compile and run your TypeScript file.
10. Observe the console output. It should display the product objects created by `createProduct` function, and these objects should conform to the `Product` interface structure.
11. Refer to the provided link for detailed information about creating interfaces in TypeScript: https://infyspringboard.onwingspan.com/web/viewer/hands-on/lex_auth_0127129252442767369 10_shared?collectionId=lex_9436233116512678000_shared&collectionType=Course

Example Code Snippet (product_interface.ts):

TypeScript
```
// product_interface.ts

interface Product {
  productId: number;
  productName: string;
}

function createProduct(id: number, name: string): Product { // Return type is Product interface
  return {
    productId: id,
    productName: name
  };
}

const laptopProduct = createProduct(1, "Laptop");
const keyboardProduct = createProduct(2, "Keyboard");

console.log("Laptop Product:", laptopProduct);
console.log("Keyboard Product:", keyboardProduct);

const productList: Product[] = [laptopProduct, keyboardProduct]; // Array of Product type
console.log("Product List:", productList);
```

Expected Output:

- Compiling and running `product_interface.ts` should output to the console.
- The console output should display the `laptopProduct` and `keyboardProduct` objects, and the `productList` array, all conforming to the `Product` interface structure (having `productId` as number and `productName` as string properties).

Observations:

Experiment 10.c: Duck Typing

Course Name: Typescript **Module Name:** Duck Typing

Objective: To demonstrate duck typing in TypeScript using the `Product` interface (from Experiment 10.b), showing how objects are considered type-compatible if they structurally match an interface, regardless of explicit class implementation.

Procedure:

1. Create a new TypeScript file (e.g., `duck_typing.ts`).
2. Reuse the `Product` interface from Experiment 10.b.
3. Create two object literals `product1` and `product2`. Ensure both objects have properties `productId` (number) and `productName` (string), fulfilling the structure of the `Product` interface. *Crucially, do not explicitly say these objects "implement" or are "instances of" the* `Product` *interface (to demonstrate duck typing).*

4. Create a function `printProductDetails(product: Product)` that takes a parameter `product` annotated with the `Product` interface. This function should access and log `product.productId` and `product.productName`.
5. Call `printProductDetails` function, passing both `product1` and `product2` objects as arguments.
6. TypeScript compiler should allow these function calls without error because `product1` and `product2` structurally match the `Product` interface (they have the required properties with correct types), even though they are just plain JavaScript objects and not explicitly declared as implementing `Product`.
7. Compile and run your TypeScript file.
8. Observe the console output. It should display the product details for both `product1` and `product2`, demonstrating that TypeScript's type system is structural (duck typing).
9. Refer to the provided link for detailed information about duck typing in TypeScript: https://infyspringboard.onwingspan.com/web/viewer/hands-on/lex_auth_0127129259954585609l2_shared?collectionId=lex_9436233116512678000_shared&collectionType=Course

Example Code Snippet (duck_typing.ts):

TypeScript

```
// duck_typing.ts

interface Product { // Interface from Experiment 10.b
   productId: number;
   productName: string;
}

// Object literals - not explicitly implementing Product interface
const product1 = {
   productId: 1001,
   productName: "Gaming Mouse"
};

const product2 = {
   productId: 1002,
   productName: "Ergonomic Keyboard"
};

function printProductDetails(product: Product): void { // Parameter type is Product interface
   console.log(`Product ID: ${product.productId}, Name: ${product.productName}`);
}
```

printProductDetails(product1); // Pass product1
printProductDetails(product2); // Pass product2

Expected Output:

- Compiling and running `duck_typing.ts` should successfully compile without TypeScript errors, demonstrating that `product1` and `product2` are considered compatible with the `Product` interface due to duck typing.

The console output should display:
 Product ID: 1001, Name: Gaming Mouse
Product ID: 1002, Name: Ergonomic Keyboard

-

Observations:

Experiment 10.d: Function Types

Course Name: Typescript **Module Name:** Function Types

Objective: To declare an interface with a function type in TypeScript and demonstrate how to use it to define the type of a function variable and access its value (which would be invoking the function).

Procedure:

1. Create a new TypeScript file (e.g., `function_type_interface.ts`).
2. Declare an interface named `Calculator`.
    - Inside this interface, declare a function type property named `operate`.
    - This `operate` property should be a function type that takes two number parameters (e.g., `a: number, b: number`) and returns a number (`number`). The full function type declaration in the interface would be: `operate: (a: number, b: number) => number;`
3.
4. Declare a variable `addOperation` and annotate its type as the `Calculator` interface.
5. Assign an object literal to `addOperation` that conforms to the `Calculator` interface. This object should have an `operate` property, and the value of `operate` should be an actual function that takes two numbers and returns their sum.

6. Now you can "access the value" of the `operate` property, which means invoking the function assigned to it. Call `addOperation.operate(5, 3)` and log the result to the console.
7. Optionally, create another function (e.g., for subtraction) and assign it to another variable of type `Calculator`, and test it.
8. Compile and run your TypeScript file.
9. Observe the console output. It should show the result of calling the `operate` function defined within the `addOperation` object, demonstrating how to use function types in interfaces.
10. Refer to the provided link for detailed information about function types in TypeScript interfaces: https://infyspringboard.onwingspan.com/web/viewer/hands-on/lex_auth_012712948945346560918_shared?collectionId=lex_9436233116512678000_shared&collectionType=Course

Example Code Snippet (function_type_interface.ts):

```
TypeScript
// function_type_interface.ts

interface Calculator {
   operate: (a: number, b: number) => number; // Function type in interface
}

let addOperation: Calculator = { // Variable of type Calculator interface
   operate: function(a: number, b: number): number { // Function implementation for 'operate'
      return a + b;
   }
};

let subtractOperation: Calculator = {
   operate: (a: number, b: number): number => a - b // Arrow function implementation
};

const sumResult = addOperation.operate(5, 3); // Access and invoke the function
console.log("Addition Result:", sumResult); // Output: Addition Result: 8

const diffResult = subtractOperation.operate(10, 4);
console.log("Subtraction Result:", diffResult); // Output: Subtraction Result: 6
```

Expected Output:

- Compiling and running `function_type_interface.ts` should output to the console.
- The console output should show the results of `addOperation.operate(5, 3)` and `subtractOperation.operate(10, 4)`, which are 8 and 6 respectively, demonstrating the invocation of functions defined through a function type interface.

Observations:

Experiment 11.a: Extending Interfaces

Course Name: Typescript **Module Name:** Extending Interfaces

Objective: To declare and extend interfaces in TypeScript, creating a `productList` interface that extends properties from two other interfaces, `Category` and `Product`, and demonstrate variable creation of this extended interface type.

Procedure:

1. Create a new TypeScript file (e.g., `extending_interfaces.ts`).
2. Declare two interfaces:
   - `Category` interface with properties like `categoryId: number` and `categoryName: string`.
   - `Product` interface with properties like `productId: number` and `productName: string` (you can reuse or slightly modify the `Product` interface from previous experiments if needed).
3.
4. Declare a third interface named `ProductList` that *extends* both `Category` and `Product` interfaces using the `extends` keyword (e.g., `interface ProductList extends Category, Product { ... }`). `ProductList` can also have its own unique properties if required (e.g., `quantity: number`).
5. Create a variable `myProductList` and annotate its type as `ProductList`.
6. Assign an object literal to `myProductList` that conforms to the `ProductList` interface. This object must include properties from `Category`, `Product`, and any unique properties defined in `ProductList` itself.
7. Log the `myProductList` object to the console to verify it conforms to the structure of the extended interface.
8. Compile and run your TypeScript file.
9. Observe the console output. It should display the `myProductList` object, which should have properties from all three interfaces (`Category`, `Product`, and `ProductList`).

10. Refer to the provided link for detailed information about extending interfaces in TypeScript: https://infyspringboard.onwingspan.com/web/viewer/hands-on/lex_auth_0127129516521390089 20_shared?collectionId=lex_9436233116512678000_shared&collectionType=Course

Example Code Snippet (extending_interfaces.ts):

```
TypeScript
// extending_interfaces.ts

interface Category {
   categoryId: number;
   categoryName: string;
}

interface Product { // Product interface - can be reused or redefined
   productId: number;
   productName: string;
}

interface ProductList extends Category, Product { // Extending multiple interfaces
   quantity: number; // Unique property of ProductList
}

let myProductList: ProductList = { // Variable of type ProductList interface
   categoryId: 10,
   categoryName: "Electronics",
   productId: 1001,
   productName: "Wireless Mouse",
   quantity: 50 // Property unique to ProductList
};

console.log("My Product List Item:", myProductList);
```

Expected Output:

- Compiling and running `extending_interfaces.ts` should output to the console.
- The console output should display the `myProductList` object, which should have properties from `Category` (`categoryId`, `categoryName`), `Product` (`productId`, `productName`), and `ProductList` itself (`quantity`).

Observations:

Experiment 11.b: Classes and Objects

Course Name: Typescript **Module Name:** Classes

Objective: To create classes and objects in TypeScript, specifically for a Mobile Cart application, creating `Product` class objects and placing them into a `productList` array.

Procedure:

1. Create a new TypeScript file (e.g., `mobile_cart_classes.ts`).
2. Define a class named `Product` in TypeScript.
    - This class should have properties relevant to a product (e.g., `productId: number`, `productName: string`, `price: number`, `description: string`).
    - Include a constructor in the `Product` class to initialize these properties when a new `Product` object is created.
    - Optionally, you can add methods to the `Product` class (e.g., a method to display product details).
3.
4. Create an empty array named `productList` of type `Product[]` to hold `Product` objects.
5. Create several instances (objects) of the `Product` class using the `new Product(...)` constructor, providing sample data for each product.
6. Push these newly created `Product` objects into the `productList` array.
7. Iterate through the `productList` array using a loop (e.g., `forEach`) and for each `Product` object in the array, log its properties to the console (or call a display method if you added one to the `Product` class).
8. Compile and run your TypeScript file.
9. Observe the console output. It should display the details of all `Product` objects that you created and added to the `productList` array, demonstrating class instantiation and object creation.
10. Refer to the provided link for detailed information about classes in TypeScript: https://infyspringboard.onwingspan.com/web/viewer/web-module/lex_3705824317381604400_shared?collectionId=lex_9436233116512678000_shared&collectionType=Course

Example Code Snippet (mobile_cart_classes.ts):

```
TypeScript
// mobile_cart_classes.ts

class Product {
   productId: number;
   productName: string;
   price: number;
   description: string;

   constructor(id: number, name: string, price: number, desc: string) {
      this.productId = id;
      this.productName = name;
      this.price = price;
      this.description = desc;
   }

   displayDetails(): void {
      console.log(`Product ID: ${this.productId}`);
      console.log(`Name: ${this.productName}`);
      console.log(`Price: $${this.price}`);
      console.log(`Description: ${this.description}`);
      console.log("---");
   }
}

const productList: Product[] = []; // Array to hold Product objects

// Create Product objects
const product1 = new Product(201, "Smartphone X", 899, "High-end smartphone with advanced
camera.");
const p[1]roduct2 = new Product(202, "Wireless Earbuds", 150, "Noise-cancelling earbuds for
immersive audio.");
const product3 = new Product(203, "Smartwatch", 250, "Fitness tracker and smartwatch in
one.");

productList.push(product1);
productList.push(product2);
productList.push(product3);

console.log("Product List Contents:");
```

[1] SOC LAB MANUAL(MEAN STACK) | M S N | ASST.PROFESSOR | DEPT OF MCA | BVCR RAJAHMUNDRY

```
productList.forEach(product => {
  product.displayDetails(); // Call method to display details
});
```

Expected Output:

- Compiling and running `mobile_cart_classes.ts` should output to the console.
- The console output should display the details of each `Product` object in the `productList` array, as formatted by the `displayDetails()` method (or however you choose to log the product properties).
-

Observations:

Experiment 11.c: Course Name: Typescript **Module Name:** [Module Name for 11.c needs to be determined from context if available]

Objective: [Objective for 11.c needs to be determined from context if available]

Procedure:

1. [Procedure for 11.c needs to be determined from context if available or exercise needs to be removed]
2. [Refer to the provided link if any for detailed instructions]

Example Code Snippet:

```
TypeScript
// [Code Snippet for 11.c needs to be determined from context]
// Example Placeholder - Replace with actual code
console.log("Experiment 11.c Code Example");
```

Expected Output:

-

Observations: