1. JavaScript Basics

Overview	
In this exercise, we will work with basic JavaScript logic before starting an application.	
Exercise Folder	lab-code
Time to Complete	20 minutes

Step 1: Use switch

1. Open ~/StudentWork/lab-code/conditions.html in the editor.
2. Make a `prompt()` method ask for a letter. The syntax is:
3. `var ans = prompt("Question", "Default Answer");`
4. Write a switch to determine whether:
5. a letter is a consonant or vowel.
6. Make the test case-insensitive.
7. Save the file and test by passing in an upper and lower-case vowel, and a consonant.

Step 2: Use the ternary operator

8. Open ~/StudentWork/lab-code/conditions.html in the editor.
9. Write a ternary operation to determine whether there should be an "s" on the end of something. Example: set a `prompt()` to ask for a number, and based on the number, determine whether to display the word "result" or "results."
10. Use the `prompt()` method to ask how many letters they like.
11. Save and test. If the number is one, it should say "You like 1 letter." All other numbers should say "You like 0 letters", "You like 2 letters", etc.

2. DEBUGGING

Overview	
In this exercise, we will handle exceptions.	
Exercise Folder	lab-code
Time to Complete	20 Minutes

Step 1: Create custom error messages

1. Open ~/StudentWork/lab-code/exceptions.html in the editor. We will use `try/throw/catch/finally`
2. Use the `prompt()` method to ask for a person's age.
3. Underneath, write out "Number is" and the variable.
4. Try whether the number is empty, NaN (use `isNaN()`), above 100 (too high) or below 1 (too low).
   - Throw exceptions (error messages) for each.
   - Catch the error message and alert it with "Input is " and the error.
5. Finally, alert "Thanks."
6. Save and test, setting the variable to different values. While the variable is an empty string, the error should say "Input is empty," if it is a letter, it will say "Input is not a number." It will say "Input is too high" for numbers above 100 or "Input is too low" for numbers below 1, such as negative numbers and zero.

Step 2: Work in strict mode

7. Open ~/StudentWork/lab-code/exceptions.html in the editor.
8. If you declared the variable `x` such as with the keyword `var`, take off the declaration.
9. Run the code again and note that when not in strict mode, it makes no difference whether the variables are declared or not.
10. Add a `"use strict";` expression at the top of the script. This must go at the top of the script or any functions to be applied to your code. The quotes make it an expression, so older browsers that don't understand it can ignore it.
11. Run the code again and note that it throws an error if variables are not declared.
12. Fix the error.
13. Save and test.

Step 3: Examine variables by writing to the `console.log()`

14. Open ~/StudentWork/lab-code/exceptions.html in the editor.
15. After the variable add `console.logc);` where *x* is your variable.
16. Save the file.
17. Open developer tools to see the variable printed out to the console.
    - You might want to set `x` to something other than an empty string.

Step 4: Set breakpoints

18. Open ~/StudentWork/lab-code/exceptions.html in the editor.
19. Before the variable declaration but under `"use strict";` add the `debugger;` statement and under that add `console.log(x);` where *x* is your variable.
20. Save and run the code as normal (not in developer mode); you will see no difference.
21. Open the browser in developer mode and run the code again. It will stop just before the variable declaration, so the variable will be unknown to the `console.log(x)` statement. There will be an

arrow that can be clicked on to continue the script. You might have to toggle over to the console tab in your browser's developer tools.

22. Under the variable declaration, add another `debugger;` statement and under that add `console.log(x);` again.
23. Save the file and run it in developer mode again.
24. The code will stop short of setting the variable as before. Clicking on the arrow will make the code run to the next breakpoint and then pause after the variable is set. You can click the arrow again to finish script execution.

3. JavaScript Built-In Objects

Overview	
In these exercises, you will use the Date object, Math object, string functions, and arrays.	
Exercise Folder	lab-code
Time to Complete	60 minutes

Step 1: Use the Date object

1. Open ~/StudentWork/lab-code/js/deadline.js in the editor. We are going to find the difference in days between today and the end of the year. Try `"use strict"`; and using `let` or `const` to set variables and constants.
2. First, get today's date. Then get the date for the end of the year, for example December 31.

```
const today = new Date();
const year = today.getFullYear();
let deadline = new Date(year, 11, 31);
```

3. Find the number of days between the two dates
4. Convert both to number of milliseconds since Unix epoch time

```
let differenceMS = deadline.getTime() - today.getTime()
```

5. Divide that number by the number of milliseconds in a day

```
let differenceDays = differenceMS/(1000 * 60 * 60 * 24);
```

6. Output that number. If you have an HTML element with the id "deadline" (e.g., `<p id="deadline"></p>`), you can use this code to write the result inside that element. Note the use of ES6 template strings.

```
document.getElementById("deadline").textContent = `Only $
{differenceDays} more days until the end of the year!`;
```

7. Save the .js file.
8. Add a reference to this script file, deadline.js, to the bottom of the index page, within paragraph tags. It will write the date out as if you had written the script right there, but the separate .js file makes this reusable on multiple pages.
9. Save the index.html page and open it in the browser. The last paragraph on the page should say how many days from now until the deadline.

Step 2: Use the Math object and arrays

10. Open `~/StudentWork/lab-code/js/random.js` in the editor. We are going to randomly generate facts about JavaScript. Try `"use strict"`; and using `let` or `const` to set variables and constants.
11. Create a function called generate. Inside:
    - Get a random fact by using `Math.random()` to get a random number, and using it as an index number on the array.

      ```
      randomFact = facts[Math.floor(Math.random()*facts.length)];
      ```

    - Write the random fact to the page.
12. Call the function.
13. Save random.js
14. Add a reference to this file to index.html in paragraphs and save.
15. Open index.html. The first paragraph should be a random fact that changes when the page is refreshed.

Step 3: Use the string and Arrays objects

16. Open `~/StudentWork/lab-code/js/capitalize.js` in the editor.
17. Define the `capitalize` Function (with default parameter):

- Create a function named `capitalize` that takes two parameters:
  - `sentence`: The sentence to be capitalized.
  - `lowercaseArticles`: A boolean value indicating whether to lowercase articles in the middle of the sentence. Make this parameter optional and set its default value to `true`.
- **Hint:** `function capitalize(sentence, lowercaseArticles = true) { ... }`
- **Make the Sentence Lowercase and Split into Words:** Inside the `capitalize` function, convert the `sentence` parameter to lowercase and split it into an array of words using the `split()` method with " " as the delimiter. Store the resulting array in a variable named `words`.
- **Hint:** `const words = sentence.toLowerCase().split(" ");`
- **Define Articles to Stay Lowercase:** Create an array called `articles` that contains all the common articles in English: "the", "a", "an", "and", "but", "or", "for", "nor", "so", "yet", "as", "at", "by", "for", "in", "of", "off", "on", "per", "to", "up", and "via".
- **Hint (you can copy):** `const articles = ["the", "a", "an", "and", "but", "or", "for", "nor", "so", "yet", "as", "at", "by", "for", "in", "of", "off", "on", "per", "to", "up", "via"];`
- **Capitalize Words:** Use the `map()` method to iterate over each word in the `words` array and apply a capitalization function. The capitalization function should:
  - Always capitalize the first word of the sentence (`index === 0`).
  - Only lowercase articles in the middle of the sentence if `lowercaseArticles` is `true`.
- **Hint:**

```
const capitalizedWords = words.map((word, index) => {
  if (index === 0) {
    return word.charAt(0).toUpperCase() + word.slice(1);
  }
  if (lowercaseArticles && articles.includes(word)) {
    return word;
  }
  return word.charAt(0).toUpperCase() + word.slice(1);
});
```

  Join the Capitalized Words: Join the capitalized words back into a sentence using the `join()` method with " " as the delimiter. **Hint:** `return capitalizedWords.join(" ");`
- **Use the Function:** Select an element with the tag name `h1` using `document.querySelector("h1")` and store it in a variable (e.g., `h1Element`). **Hint:** `const h1Element = document.querySelector("h1");`
- **Call the Function:** Show the user how to call the function with different options:
  - To have lowercase articles in the middle of the sentence:
    `h1Element.textContent = capitalize(h1Element.textContent, true);`
  - To capitalize all words except the first word:
    `h1Element.textContent = capitalize(h1Element.textContent, false);`

18. To use the default behavior (lowercase articles in the middle):
    `h1Element.textContent = capitalize(h1Element.textContent);`

Step 4: CSS Styling in JavaScript

19. Open `~/StudentWork/lab-code/js/nav.js` in the editor.

- Use `document.querySelectorAll(".mynav a")` to select all the `<a>` elements (links) that are descendants of an element with the class "mynav". Store these links in a variable called `links`.
- **Hint:** `const links = document.querySelectorAll(".mynav a");`
- **Loop Through the Links:** Use the `forEach()` method to iterate over each link in the `links` NodeList.
- **Hint (uses arrow function):** `links.forEach(link => { ... });`
- **Add Mouseenter Event Listeners:** Inside the `forEach()` loop:
  - For each `link`, add an event listener that listens for the `mouseenter` event (when the mouse cursor moves over the link).
  - When the `mouseenter` event occurs, execute a function.
- **Hint:** `link.addEventListener("mouseenter", function() { ... });`

5. Fade Other Links: Inside the event listener function-

    - Use another `forEach()` loop to iterate through all the links again (using a different variable name, like `otherLink`, to avoid confusion).
    - For each `otherLink`, set its `style.opacity` to "0.25" to make it semi-transparent.
- **Hint (uses forEach):**
  `links.forEach(otherLink => otherLink.style.opacity = "0.25");`
- **Highlight the Current Link:** Still inside the event listener function-
    - Use `this.style.opacity = "1";` to set the opacity of the link that triggered the `mouseenter` event back to 1 (fully opaque), effectively highlighting it.

Explanation of ES5 code in comments in solutions:

The ES5 code accomplishes the same thing, but with older JavaScript syntax:

- `var` instead of `const`: `var` was used for variable declaration in older JavaScript.
- `for` loop: Instead of `forEach()`, a traditional `for` loop iterates through the `links` NodeList using an index (`i`).
- Nested `for` loop: Another `for` loop (with index `j`) is used inside the event listener function to iterate over the links and fade them.

4. Overview of ES6 Features

Overview	
In this exercise, we will explore some of ES6's new features and create classes.	
Exercise Folder	lab-code
Time to Complete	45 Minutes

Let's convert some ES5 to ES6 and implement new features in `~StudentWork/lab-code/js/ES6features.js`.

`let` and `const` (Block Scoping)

- We will explore the concept of block-level scoping introduced in ES6 with `let` and `const`.
  - **`var`:** variables declared with `var` have function-level scope, meaning they are accessible throughout the entire function they are declared in, even outside of code blocks (`if`, `for`, etc.).
  - **`let`:** allows you to declare variables with block-level scope, meaning they are only accessible within the block of code (`{ }`) where they are defined. This helps avoid unintended variable reassignments or modifications.
  - **`const`:** used to declare constants, which are variables that cannot be reassigned after their initial value is set.

2. Arrow Functions (Concise Syntax)

- Arrow functions provide a shorter syntax for writing function expressions.
  - The traditional way of defining functions uses the `function` keyword.
  - Note the arrow function syntax: (`(parameters) => expression` or `(parameters) => { statements }`).

3. Template Literals (String Interpolation)

- Template literals make it easier to create strings with embedded expressions.
  - The old way of concatenating strings used the + operator.
  - Use backticks (``) and `${}` placeholders in template literals to embed variables and expressions directly into strings.

4. Default Parameters (Function Arguments)

- Default parameters let you set default values for function parameters if no argument is provided or if the argument is `undefined`.

5. Destructuring Assignment (Object and Array Extraction)

- Destructuring assignment provides a concise way to extract values from objects and arrays.
  - The old way of accessing object properties uses dot notation.
  - Destructuring can assign object properties or array elements to variables.

6. Rest and Spread Syntax (Variable Number of Arguments and Array Manipulation)

  - The rest parameter (`...`) allows a function to accept an indefinite number of arguments as an array.
  - The spread syntax (`...`) can be used to expand an array into individual elements.
  - The context determines whether this operator is rest or spread.

7. Classes and Inheritance (Object-Oriented Programming)

- ES6 classes provide a cleaner syntax for creating objects and implementing inheritance.

    - The old approach was to create constructor functions and use prototypes.
    - Define classes using the `class` keyword, constructors (`constructor`), methods, and inheritance (`extends`, `super`).

8. Generators (Iterators)

- Generators are special functions that can be paused and resumed, so you can generate a sequence of values over time.
    - Define a generator function using the `function*` syntax and the `yield` keyword.
    - Iterate over the values produced by a generator using a `for...of` loop.

9. Symbols (Unique Identifiers)

- Symbols are a new primitive data type in ES6 that creates unique identifiers.
    - Create a Symbol using `Symbol()`.
    - Symbols can be used as keys in objects to avoid naming collisions and create "hidden" properties.

10. Maps and Sets (Data Structures)

- **Maps:** a new data structure that allow you to store key-value pairs where the keys can be of any data type (not just strings like in objects).
- **Sets:** a new data structure that store unique values, preventing duplicates.

5. Using Events and the DOM

Overview	
In these exercises, you will use JavaScript DOM and events.	
Exercise Folder	lab-code
Time to Complete	75 minutes

Step 1: Creating a Client-Side Search

1. Select the Search Input:

- In `~StudentWork/lab-code/js/filter.js`, Use `document.querySelector("#search")` to select the HTML element with the ID "search".

2. (Optional) Add a Keyup Event Listener:

- Attach an event listener to the search input that listens for the `keyup` event (when a key is released). This will trigger the search functionality every time the user types something in the search field. **Hint:** `.addEventListener("keyup", () => { ... });`

3. Get the Search Term:

- Inside the event listener function, get the value entered in the search input field (`document.getElementById("search").value`) and convert it to uppercase using `toUpperCase()`. Store this uppercase search term in a variable called `filter`

4. Get the Table Rows:

- Select the table element with the ID "infoTable" (`document.getElementById("infoTable")`) and then get all of its `<tr>` elements (table rows) using `getElementsByTagName("tr")`. Store these rows in a variable named `tr`.

5. Loop Through the Rows:

- Use a `for` loop to iterate through each row in the `tr` HTMLCollection. **Hint:** `for (let i = 0; i < tr.length; i++) { ... }`

6. Get Table Data:

- Inside the loop, for each row (`tr[i]`), get all its `<td>` elements (table data cells) using `getElementsByTagName("td")`. Store these cells in a variable named `td`. **Hint:** `const td = tr[i].getElementsByTagName("td");`

7. Filter the Rows (Search all fields):

- Check if the row has any table cells (`td.length`) and if the uppercase version of the row's inner HTML (`tr[i].innerHTML.toUpperCase()`) contains the `filter` text (using `indexOf(filter) > -1`).
- If both conditions are true, set the row's display style to "" (show the row). Otherwise, set it to "none" (hide the row). This effectively filters the table rows based on the search term.
- **Hint:** `tr[i].style.display = (td.length && tr[i].innerHTML.toUpperCase().indexOf(filter) > -1) ? "" : "none";`

8. (Optional) Filter by Name Field Only:

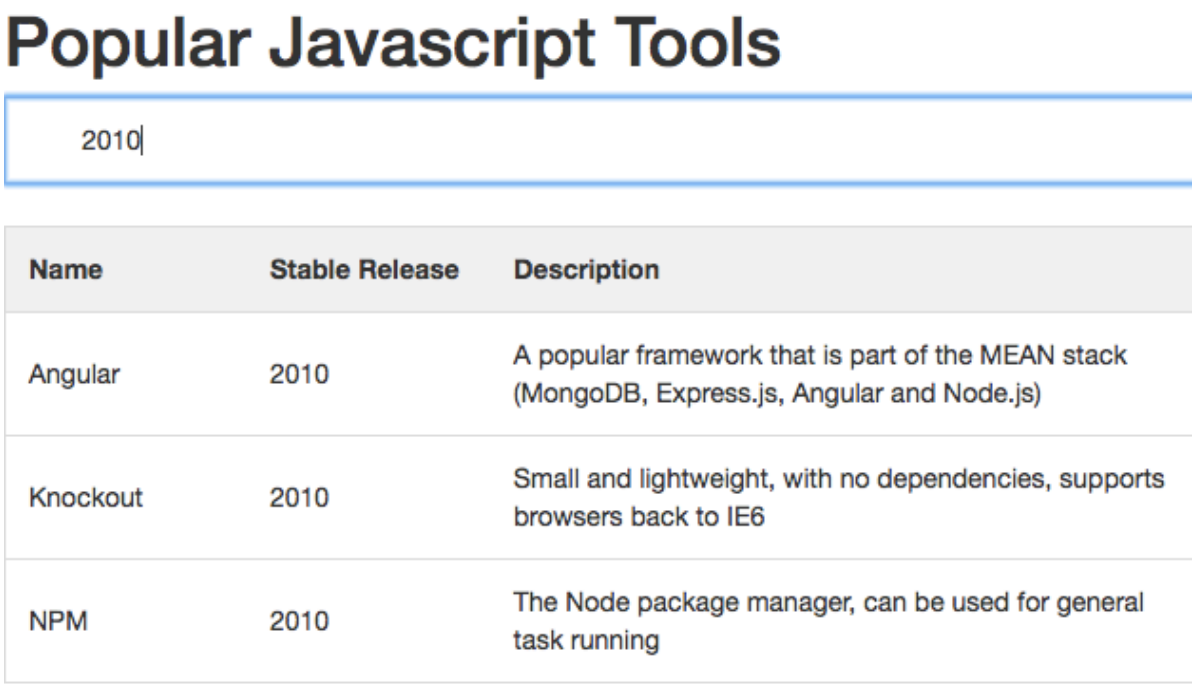
- If you want to search only within the first column (name field), you can modify the filtering logic to check `td[0].innerHTML.toUpperCase().indexOf(filter)` instead of `tr[i].innerHTML...`.

- **Hint:** `tr[i].style.display = (td.length && td[0].innerHTML.toUpperCase().indexOf(filter) > -1) ? "" : "none";` (This line is commented out in the example)

Explanation of ES5 code in comments in solutions:

The ES5 code achieves the same result but uses older JavaScript syntax:

- `var` instead of `const` and `let`: `var` was the primary way to declare variables in older JavaScript.
- `function()` instead of `() => { ... }`: ES5 used the `function` keyword to define anonymous functions.



Step 2: Creating Rotating Images

1. Open ~StudentWork/lab-code/js/locations.js and Initialize Slide Index:

- Create a variable called `slideIndex` and set it to 1. This variable will keep track of the currently active slide. **Hint:** `let slideIndex = 1;`

2. Show Initial Slide: Call the `showSlides()` function (which you'll define later) with `slideIndex` as the argument to display the first slide.**Hint:** `showSlides(slideIndex);`

3. Add Event Listeners for Thumbnail Dots:

- **You will:**
  - Select all elements with the class "dot" using `document.querySelectorAll('.dot')`. These are likely thumbnail images representing each slide.
  - Use the `forEach()` method to iterate over these "dot" elements.
  - For each dot, add a 'click' event listener that calls the `currentSlide()` function with the dot's index + 1 as the argument (to jump to the corresponding slide).
- **Hint (uses forEach and arrow function):**

```
document.querySelectorAll('.dot').forEach((dot, index) => {
  dot.addEventListener('click', () => currentSlide(index + 1));
});
```

4. Define the `plusSlides()` Function:

- **You will:**
  - Create a function called `plusSlides()` that takes one parameter `n` (the number of slides to move forward or backward).
  - Inside the function, update `slideIndex` by adding `n` to it (`slideIndex += n`).
  - Call `showSlides()` with the updated `slideIndex`.
- **Hint:**

```
function plusSlides(n) {
      showSlides(slideIndex += n);
}
```

5. Define the `currentSlide()` Function:

- **You will:**
  - Create a function called `currentSlide()` that takes one parameter `n` (the index of the slide to show).
  - Inside the function, set `slideIndex` to `n`.
  - Call `showSlides()` with the updated `slideIndex`.
- **Hint:**

```
function currentSlide(n) {
  showSlides(slideIndex = n);
}
```

6. Define the `showSlides()` Function:

- **You will:**
  - Create a function called `showSlides()` that takes one parameter `n` (the index of the slide to show).
  - Inside the function:
    - Get all elements with the class "mySlides" and store them in `slides`.
    - Get all elements with the class "dot" and store them in `dots`.
    - If `n` is greater than the number of slides, reset `slideIndex` to 1 (loop back to the beginning).
    - If `n` is less than 1, set `slideIndex` to the last slide's index (loop back to the end).
    - Use a `for` loop to hide all slides (`slides[i].style.display = "none";`).
    - Use another `for` loop to remove the "active" class from all dots.
    - Show the slide at the `slideIndex - 1` position (`slides[slideIndex-1].style.display = "block";`).
    - Add the "active" class to the dot at the `slideIndex - 1` position.
- **Hint:**

```
function showSlides(n) {
  let i;
  let slides = document.getElementsByClassName("mySlides");
  let dots = document.getElementsByClassName("dot");
  if (n > slides.length) {slideIndex = 1}
  if (n < 1) {slideIndex = slides.length}
  for (i = 0; i < slides.length; i++) {
    slides[i].style.display = "none";
  }
  for (i = 0; i < dots.length; i++) {
    dots[i].className = dots[i].className.replace(" active", "");
  }
  slides[slideIndex-1].style.display = "block";
  dots[slideIndex-1].className += " active";
}
```

7. Set Up Automatic Slideshow:

- Use `setInterval()` to call the `plusSlides(1)` function every 2 seconds (2000 milliseconds) to automatically advance the slideshow.

```
setInterval(function() {
  plusSlides(1);
```

```
}, 2000);
```

Explanation of ES5 code in comments in solutions:

The ES5 code is functionally equivalent, with these key differences:

- `var` instead of `let`: `var` was used for variable declarations in older JavaScript.
- `function()` instead of `() => { ... }`: ES5 used the `function` keyword to define anonymous functions.
- IIFE for dot event listeners: The ES5 code uses Immediately Invoked Function Expressions (IIFEs) to create a closure and correctly handle the `index` variable in the loop for the dot event listeners. This was a common workaround for scoping issues in older JavaScript.

Step 3: Validating Form Data

1. 1. Open ~StudentWork/lab/code/js/contact.js and Select All Input Fields:

- Use `document.querySelectorAll("input")` to select all the input elements in the document. This will give you a NodeList containing all the input fields. Store this NodeList in a variable named `inputs`.
- **Hint:** `const inputs = document.querySelectorAll("input");`

2. Loop Through Each Input:

- Use the `forEach()` method to iterate over each input element in the `inputs` NodeList.
- **Hint:** `inputs.forEach(input => { ... });`

3. Initialize Error Message Variable:

- Inside the `forEach` loop, declare a variable named `errorMessage` and initialize it to `null`. This variable will store the error message element if one is created.
- **Hint:** `let errorMessage = null;`

4. Add Blur Event Listener:

- For each `input` element, add an event listener that listens for the `blur` event (when the input field loses focus).
- **Hint:** `input.addEventListener("blur", function() { ... });`

5. Validate Input Based on Type (using `switch`):

- Inside the `blur` event listener function:
  - Get the `id` of the input element (which is assumed to indicate the input type) and store it in `inputType`.
  - Use a `switch` statement to check the `inputType` and perform different validation based on the type:
    - "**phone**": Use a regular expression (`/^\d{10}$/`) to check if the input value is a 10-digit number.
    - "**email**": Use a regular expression to validate the email format.
    - "**zip**": Use a regular expression to validate the zip code format.
    - "**default**": For any other input type, check if the input value is not empty.
  - Set the `className` of the input element to "form-control" if the input is valid, or "form-control error" if it's invalid.
  - If an `errorMessage` exists, remove it and set `errorMessage` back to `null`.
- **Hint:**

```
const inputType = this.id;
let isValid = true;

switch (inputType) {
  case "phone":
```

```
      // ... validation logic ...
      break;
    case "email":
      // ... validation logic ...
      break;
    case "zip":
      // ... validation logic ...
      break;
    default:
      // ... validation logic ...
  }

  this.className = isValid ? "form-control" : "form-control error";

  if (errorMessage) {
    errorMessage.remove();
    errorMessage = null;
  }
```

6. Add Keydown Event Listener:

- Add another event listener to the input element that listens for the `keydown` event.
- **Hint:** `input.addEventListener("keydown", (event) => { ... });`

7. Trigger Blur on Enter or ArrowRight:

- Inside the `keydown` event listener, if the pressed key is "Enter" or "ArrowRight", trigger the `blur` event on the input element. This simulates moving to the next field or submitting the form.
- **Hint:**

```
if (event.key === "Enter" || event.key === "ArrowRight") {
      this.blur();
}
```

8. Disable Cut, Copy, and Paste:

- **You will:**
  - Loop through the events "cut", "copy", and "paste".
  - For each event type, add an event listener to the input that prevents the default action (disabling cut, copy, and paste).
  - If an `errorMessage` exists, remove it.
  - Call the `showError()` function (which you'll define later) to display an error message indicating that the action is disabled.
- **Hint:**

```
["cut", "copy", "paste"].forEach(eventType => {
      input.addEventListener(eventType, (event) => {
    // ... disable cut/copy/paste logic ...
  });
});
```

9. Remove Error Message on Input:

- Add an event listener to the input that listens for the `input` event (when the user types something). Inside the listener, if an `errorMessage` exists, remove it.
- **Hint:** `input.addEventListener("input", () => { ... });`

10. Define the `showError()` Function:

- Create a function called `showError()` that takes the `input` element and the `eventType` as arguments.
  - Create a `div` element to display the error message.

    - Set the text content of the error message.
    - Add a CSS class to the error message element.
    - Insert the error message into the DOM after the input field.
    - Use a `setTimeout` to smoothly animate the height of the error message for a better visual effect.
    - Return the `errorMessage` element.
- **Hint:**
```
function showError(input, eventType) {
  // ... create and display error message logic ...
}
```

Explanation of ES5 code in comments in solutions:

The ES5 code achieves the same functionality but with older JavaScript syntax:

- `var` instead of `const` and `let`: `var` was used for variable declarations in older JavaScript.
- Traditional `for` loop instead of `forEach()`: ES5 used `for` loops to iterate over collections.
- `event.keyCode` instead of `event.key`: ES5 used `keyCode` to identify keys.
- String concatenation instead of template literals: ES5 used the `+` operator to combine strings.

6. WORKING WITH CLASSES

Overview	
In this exercise, you will use JavaScript classes in ES6 and the localStorage API.	
Exercise Folder	lab-code
Time to Complete	60 minutes

1. Open ~/StudentWork/lab-code/js/quizquestions.js in the editor.

- The `Quiz` class is responsible for managing the quiz logic.
    - **`constructor():`** the constructor initializes the quiz with an array of `questions`, sets the initial `score` to 0, and sets the `currentQuestionIndex` to 0 to start from the first question.
    - **`guess():`** this method checks if the provided `answer` is correct for the current question. If it is, the `score` is incremented. Then, it moves to the next question by incrementing `currentQuestionIndex`.
    - **`getCurrentQuestion():`** this method returns the `Question` object for the current question based on `currentQuestionIndex`.
    - **`hasEnded():`** this method checks if the quiz has ended by comparing `currentQuestionIndex` to the length of the `questions` array.
- **Hint:**

```
class Quiz {

  constructor(questions) {
    this.score = 0;
    this.questions = questions;
    this.currentQuestionIndex = 0;
  }

  guess(answer) {
    if (this.getCurrentQuestion().isCorrectAnswer(answer)) {
      this.score++;
    }
    this.currentQuestionIndex++;
  }

  getCurrentQuestion() {
    return this.questions[this.currentQuestionIndex];
  }

  hasEnded() {
    return this.currentQuestionIndex >= this.questions.length;

 }
}
```

2. `Question` Class

- The `Question` class represents a single question in the quiz.
    - **`constructor():`** the constructor takes the `text` of the question, an array of `choices`, and the correct `answer` as parameters and stores them.
    - **`isCorrectAnswer():`** this method checks if the provided `choice` matches the correct `answer` for the question.
- **Hint:**

```
class Question {
```

```
    constructor(text, choices, answer) {
        this.text = text;
        this.choices = choices;
        this.answer = answer;
      }

      isCorrectAnswer(choice) {
        return this.answer === choice;
    }
}
```

3. `QuizUI` Object

- The `QuizUI` object is responsible for displaying the quiz and handling user interactions.
    - **`displayNext()`:** this method controls the flow of the quiz. It checks if the quiz has ended using `quiz.hasEnded()`. If it has, it calls `displayScore()`. Otherwise, it calls `displayQuestion()`, `displayChoices()`, and `displayProgress()` to show the current question.
    - **`displayQuestion()`:** this method displays the text of the current question by calling `populateIdWithHTML()` to set the HTML content of an element with the ID "question".
    - **`displayChoices()`:** this method displays the answer choices for the current question. It iterates through the `choices` array and calls `populateIdWithHTML()` to set the HTML content of elements with IDs "choice0", "choice1", etc. It also attaches click handlers to these elements using `guessHandler()`.
    - **`displayScore()`:** this method displays the final score when the quiz ends. It constructs an HTML string with the results and uses `populateIdWithHTML()` to set the content of an element with the ID "quiz".
    - **`populateIdWithHTML()`:** this is a utility method that takes an element ID and some HTML content and sets the `innerHTML` of the element with that ID.
    - **`guessHandler()`:** this method attaches a click handler to an answer choice element. When clicked, it calls `quiz.guess()` with the selected answer and then calls `displayNext()` to move to the next question.
    - **`displayProgress()`:** this method displays the current question number and the total number of questions. It can also include a progress bar visualization using a `<canvas>` element.
- **Hint:**

```
const QuizUI = {

      // ... (methods as described above) ...
};
```

4. Creating Questions and Quiz Instance

- **You will create:**
    - an array of `Question` objects is created using the `Question` class constructor. Each `Question` object represents a question with its choices and correct answer.
    - a new `Quiz` object is created using the `Quiz` class constructor, passing the `questions` array as an argument.
- **Hint:**

```
const questions = [

 new Question("Which one is a client-side language?", ["Python",
"JavaScript", "PHP", "Perl"], "JavaScript"),
  // ... more questions ...
];
```

```
const quiz = new Quiz(questions);
```

5. Starting the Quiz

- `QuizUI.displayNext()` is called to start the quiz. This will display the first question and its choices.

- **Hint:**
  `QuizUI.displayNext();`

Explanation of ES5 Code

- **Constructor Functions and Prototypes:** ES5 uses constructor functions (`function Quiz() {...}`) and prototypes (`Quiz.prototype.guess = ...`) to define classes and methods. This is the older way of achieving object-oriented programming in JavaScript.
- **`this` Context:** `this` refers to the object instance within the constructor function and methods.
- **Object Literal:** `QuizUI` is an object literal (`var QuizUI = { ... }`) containing methods.
- **`var`:** `var` was used for variable declarations in ES5.
- **String Concatenation:** ES5 used the + operator to concatenate strings.

7. Using the Drag and Drop API and Implementing JSON

Overview	
In this exercise, you will use the drag and drop API to create a shopping cart, and use JSON to store data.	
Exercise Folder	lab-code
Time to Complete	60 minutes

Use the drag and drop API for a shopping cart, and use JSON to store data

1. Open ~/StudentWork/lab-code/js/cart.js in the editor.
2. **Set Up Product Data**

- Create an array named `productsData` to store the information about your products. Each product should be an object with properties like `id`, `name`, `price`, and `image` (the path to the product image). Make sure the ids are numbers with no quotes around them.

```
const productsData = [
  { id: 1, name: "JavaScript and JQuery: Interactive Front-End Web
Development", price: 10, image: "img/product1.jpg" },
  { id: 2, name: "Eloquent JavaScript", price: 15, image: "img/
product2.jpg" },
  { id: 3, name: "You Don't Know JS", price: 20, image: "img/
product3.jpg" },
  // ... more products
];
```

- **Select DOM elements:** Use `document.querySelector()` to select the important HTML elements that you'll need to interact with. These include:
  - The container where you'll display the products (e.g., `productsContainer`).
  - The shopping cart element (e.g., `shoppingCart`).
  - The list element within the shopping cart to display cart items (e.g., `shoppingCartList`).
  - Elements to display the product quantity, total price, and the "Empty Cart" button.
- **Hint:**

```
const productsContainer = document.querySelector('.products');
const shoppingCart = document.querySelector('.shopping-cart');
const shoppingCartList = document.querySelector('.shopping-cart-
list');
const productQuantity = document.querySelector('.product-quantity');
const totalPrice = document.querySelector('.total-price');
const emptyCartBtn = document.querySelector('.empty-cart-btn');
```

- **Initialize cart items:** Create an array named `cartItems` to store the items that the user adds to their cart. Initially, this array will be empty: `let cartItems = [];`
- **Create `renderProducts()`:** Define a function called `renderProducts()` to display the products on the page.
  - Clear the `productsContainer` (`innerHTML = ''`).
  - Use a `forEach()` loop to iterate through the `productsData` array.
  - For each product:
    - Create a `div` element (`document.createElement('div')`).
    - Add the class "product" to the `div` for styling.
    - Set the `draggable` attribute to `true` to make it draggable.
    - Use `dataset.productId` to store the product ID on the element.
    - Set the `innerHTML` of the `div` to include the product image, name, and price (using template literals for easier formatting).
    - Add the `div` to the `productsContainer`.
    - Attach a `dragstart` event listener to the `div` to handle the drag operation (call the `dragStart()` function, which you'll define later).

- **Hint:**
```
function renderProducts() {
  // ... (code as described above) ...
}
```
- **Create `renderCartItems()`:** Define a function called `renderCartItems()` to display the items in the shopping cart and update the cart summary.
  - Clear the `shoppingCartList` (`innerHTML = ''`).
  - Use a `forEach()` loop to iterate through the `cartItems` array.
  - For each item:
    - Create a `li` element (`document.createElement('li')`).
    - Set the `innerHTML` of the `li` to include the item name, price, quantity, and a "Remove" button (with the `data-product-id` attribute to store the product ID).
    - Add the `li` to the `shoppingCartList`.
    - Attach a `click` event listener to the "Remove" button to handle removing the item (call the `removeItem()` function, which you'll define later).
  - Update the cart summary:
    - Use `reduce()` to calculate the total quantity of items in the cart and update the `productQuantity` element.
    - Use `reduce()` again to calculate the total price and update the `totalPrice` element (format the price using `toFixed(2)`).
- **Hint:**
```
function renderCartItems() {
  // ... (code as described above) ...
}
```

- **Make the drag and drop handlers:** Define the following functions to handle the drag and drop events:
  - **`dragStart(event)`:**
    - Use `event.dataTransfer.setData()` to set the data that will be transferred during the drag operation (in this case, the `productId`).
  - **`dragOver(event)`:**
    - Use `event.preventDefault()` to allow a drop to occur on the target element.
  - **`drop(event)`:**
    - Use `event.preventDefault()` to prevent the default drop behavior.
    - Use `event.dataTransfer.getData()` to get the dragged data (`productId`).
    - Use `find()` to find the corresponding product in the `productsData` array.
    - Call the `addToCart()` function (which you'll define later) to add the product to the cart.
- **Hint:**
```
function dragStart(event) {
  // ...
}

function dragOver(event) {
  // ...
}

function drop(event) {
  // ...
}
```

- **Add cart functionality:** Define the following functions to manage the shopping cart:
  - **`addToCart(product)`:**
    - Use `findIndex()` to check if the product already exists in the `cartItems` array.
    - If it exists, increment the `quantity` property of the existing item.
    - If it doesn't exist, create a new object with the product data and a `quantity` of 1 (use the spread syntax `...product` to copy the product data), and add it to the `cartItems` array.
    - Call `renderCartItems()` to update the cart display.

    - **`removeItem(event)`:**
        - Get the `productId` from the "Remove" button's `dataset`.
        - Use `filter()` to create a new `cartItems` array without the item to be removed.
        - Call `renderCartItems()` to update the cart display.
    - **`emptyCart()`:**
        - Set `cartItems` to an empty array (`[]`).
        - Call `renderCartItems()` to update the cart display.
- **Hint:**
```
function addToCart(product) {
  // ...
}

function removeItem(event) {
  // ...
}

function emptyCart() {
  // ...
}
```

- **Attach event listeners:** Attach the following event listeners:
    - `dragover` and `drop` listeners to the `shoppingCart` element.
    - `click` listener to the `emptyCartBtn`.
- **Hint:**
```
shoppingCart.addEventListener('dragover', dragOver);
shoppingCart.addEventListener('drop', drop);
emptyCartBtn.addEventListener('click', emptyCart);
```

- **Initial rendering:** Call `renderProducts()` and `renderCartItems()` to initially display the products and the (empty) cart.
- **Hint:**
```
renderProducts();
renderCartItems();
```

Differences in ES6 code and ES5 code:

`var` instead of `let` and `const`: ES5 uses `var` for all variable declarations. `var` has function-level scope, unlike `let` and `const`, which have block-level scope.

Function Expressions: ES5 uses function expressions (`function(product) { ... }`) for anonymous functions in `forEach()`, `findIndex()`, `filter()`, and event listeners. ES6 provides arrow functions (`(product) => { ... }`) which have a more concise syntax and handle the `this` context differently.

String Concatenation: ES5 uses the + operator to concatenate strings when creating HTML content. ES6 introduces template literals (backticks ``) which allow for easier string interpolation and multi-line strings.

Object Cloning: In ES5, `Object.create(product)` is used to create a copy of the `product` object before adding it to the cart. This is similar to using the spread syntax (`...product`) in ES6, which is a more modern and concise way to achieve the same result.

No Arrow Functions: ES5 doesn't have arrow functions, so traditional function expressions are used for event listeners and callbacks.

8. Using AJAX for the Cart

Overview	
In this exercise, we will learn how to implement AJAX	
Objective	Transform our shopping cart to use AJAX
Exercise Folder	lab-code
Time to Complete	60 minutes

Step 1: Set up the Development Environment

- You'll use `json-server` to create a mock API that serves your product data. This avoids the need to set up a full backend server for this example.
  - **Install Node.js and npm:** If you don't have them already, download and install Node.js and npm (Node Package Manager) from the official website (https://nodejs.org/).
  - **Make a directory called public and install `json-server` in the public directory:** Open your terminal or command prompt and run the following command:
    `npm install json-server`

    We might have to `npm install cors` if we get CORS errors in the network tab

2. Create `db.json`

- Create a file named `db.json` in your project directory. This file will act as your mock database.
  - Add your product data to `db.json` in JSON format. Each product should be an object with `id`, `name`, `price`, and `image` properties.
- **Example (`db.json`)**

```
{
  "products": [
    { "id": 1, "name": "JavaScript and JQuery", "price": 10,
"image": "img/product1.jpg" },
    { "id": 2, "name": "Eloquent JavaScript", "price": 15, "image":
"img/product2.jpg" },
    { "id": 3, "name": "You Don't Know JS", "price": 20, "image":
"img/product3.jpg" }
  ]
}
```

3. Run `json-server`

- In your terminal, navigate to your project directory and run the following command:
  `json-server --watch db.json`

  This starts `json-server` and serves your data from `db.json` on `http://localhost:3000/`. The `--watch` flag automatically reloads the server if you make changes to `db.json`.

4. Understanding How `json-server` Works

- The benefits of using `json-server`:
  - **Mock API:** `json-server` creates a RESTful API from your `db.json` file. You can make GET, POST, PUT, DELETE requests to this API just like you would to a real backend server.
  - **Rapid Prototyping:** This is great for quickly prototyping front-end applications without needing a real backend.
  - **Simplified Data Management:** You can easily manage your data in a simple JSON file.

5. Use the `public` folder to serve static files:

- Use the folder that you named `public` for your HTML, CSS, and JavaScript files.
  - **Place your `index.html`, `style.css`, and `cart.js` inside the `public` folder.**

6. Run a Static File Server (if needed)

- If you don't have a way to serve your static files (HTML, CSS, JavaScript), you can use a simple static file server.
  `json-server --watch db.json –static .`

  This starts a simple server on `http://localhost:8080/`. You can then access your `index.html` file by visiting that address in your web browser.

- **Fetch product data instead of using an object of products:**
  - Use the `fetch` API to retrieve product data from the `json-server` API you set up (`http://localhost:3000/products`).
    - Use `.then()` to handle the response and parse the JSON data.
    - Store the fetched products in the `productsData` array.
    - Call `renderProducts()` to display the products.
    - Attach the `drop` event listener to the `shoppingCart` element after the products are fetched.
    - Use `.catch()` to handle any errors during the fetch process.
  - **Hint:**

```
fetch('http://localhost:3000/products')
  .then(response => response.json())
  .then(products => {
    // ... (code as described above) ...
  })
  .catch(error => {
    console.error('Error fetching products:', error);
  });
```

- **`renderProducts(products)`:**
  - This function takes an array of `products` as input and dynamically creates product elements to display on the page. It's similar to the previous example, but now it receives the `products` data from the `fetch` call.
  - **Hint:**

```
function renderProducts(products) {
 // ... (code to create and display product elements) ...
}
```

- **How this would look in ES5:**
  - **`XMLHttpRequest`:** In ES5, you would use `XMLHttpRequest` to make the API request instead of `fetch`.
  - **`var`:** ES5 uses `var` for variable declarations.
  - **Function Expressions:** ES5 uses function expressions for anonymous functions.
  - **String Concatenation:** ES5 uses the `+` operator for string concatenation.