Exercise 1: Capitalization function using DOM, ES6 and Functions

1. Define the function

```
const capitalize = (sentence, lowercaseArticles = true) => {
```

This defines a function called capitalize using an ES6 arrow function.

It takes two parameters: sentence: a string of text to be capitalized, lowercaseArticles: an optional boolean (default is true). If true, small words like "the", "of", "in" are not capitalized unless they are the first word.

2. Use Set for articles:

```
const articles = new Set(["the", "a", "an", "and", "but", "or", "for",
"nor", "so", "yet", "as", "at", "by", "in", "of", "off", "on", "per",
"to", "up", "via"]);
```

A Set is used here for fast lookup (faster than using an array when checking if a word is in the list). These are common short connector words (articles, conjunctions, and prepositions) that are not usually capitalized in titles, unless they're the first word.

3. Convert to lower case:

```
return sentence
  .toLowerCase()
```

Converts the entire sentence to lowercase first to ensure consistent formatting before capitalizing the correct words

4. Split sentence into words:

```
.split(" ")
```

Splits the sentence into an array of words using the space character as the delimiter.

5. Use map to iterate over each word:

```
.map((word, i) =>
    i === 0 || !(lowercaseArticles && articles.has(word))
      ? word[0].toUpperCase() + word.slice(1)
      : word
  )
```

map() loops over every word in the array.

For each word: If it's the first word (i === 0) → always capitalize it. If lowercaseArticles is true and the word is in the articles list → leave it lowercase. Otherwise capitalize the first letter:

`word[0].toUpperCase()` converts the first character to uppercase.

`word.slice(1)` returns the rest of the word unchanged.

6. Join Words Back Into Sentence:

```
 .join(" ");
```

Joins the array of words back into a single string with spaces between them.

7. Apply to Web Page:

```
document.querySelector("h1").textContent =
capitalize(document.querySelector("h1").textContent);
```

Finds the first <h1> element on the page.

Gets its current text.

Applies the capitalize function to it.

Replaces the original text with the capitalized version.

Exercise 2: Form Validation with ES6, Functions, DOM:

1. Define Validation Rules

Include rules for:

Phone: 10-digit numbers only (1234567890)

Email: Basic format with @ and a domain

Zip:

U.S. 5-digit or ZIP+4 (12345 or 12345-6789)

Canadian format (A1B 2C3), case-insensitive

```
const validators = {
  phone: val => /^\d{10}$/.test(val),
  email: val => /^[\w.-]+@[\w.-]+\.(\w{2,})$/.test(val),
  zip: val => /^(\d{5}(-\d{4})?|[A-Z]\d[A-Z] ?\d[A-Z]\d)$/i.test(val)
};
```

2. Target Form and Inputs. Select:

The form with id="myForm"

All <input> elements inside the form

```
const form = document.getElementById("myForm");
const inputs = form.querySelectorAll("input");
```

3. Loop Through Each Input

Each input field gets three behaviors:

A. blur Event

When a user leaves the field:

It checks if the field is valid.

If not, it shows an error message like Invalid email.

B. input Event

As the user types:

Any visible error message is removed.

C. keydown Event

If the user presses Enter or ArrowRight, the field blurs (triggers validation).

D. Cut/Copy/Paste Blocked

These actions are prevented and show a message like paste is disabled in this field.

4. Error Handling Functions

Each field:

Shows a visible error using a sibling element (assumed to be a <div> after each <input>)

Adds/removes classes like .visible and .error to style errors

```
const showError = msg => {
  errorBox.textContent = msg;
  errorBox.classList.add("visible");
  input.classList.add("error");
};

const clearError = () => {
  errorBox.textContent = "";
  errorBox.classList.remove("visible");
  input.classList.remove("error");
};
```

5. On Form Submission

When the form is submitted:

All inputs should be checked for validity

If any are invalid:

Errors are shown

Submission is stopped

An alert appears: “Please fix the errors before submitting.”

```
form.addEventListener("submit", e => {
  let hasError = false;
  inputs.forEach(input => {
    const { id, value } = input;
    const isValid = validators[id]?.(value.trim());
    if (!isValid) {
      input.nextElementSibling.textContent = `Invalid ${id}`;
      input.nextElementSibling.classList.add("visible");
      input.classList.add("error");
      hasError = true;
    }
  });

  if (hasError) {
    e.preventDefault();
    alert("Please fix the errors before submitting.");
  }
});
```

In the HTML, make sure:

Each <input id="phone" | "email" | "zip"> has a sibling element right after it (e.g., <div class="error-message"></div>) for showing the error.

You have a <form id="myForm"> surrounding the inputs.

Exercise 3: ES6, Arrays, Functions, DOM: Image Slideshow

Goal:

Create an automatic slideshow (carousel) system that:

Switches slides every 2 seconds

Lets the user go to the previous/next slide via buttons

Lets the user click on a "dot" to jump to a specific slide

Visually highlights the current dot

Uses modern JavaScript (ES6+) for cleaner code

1. Initialize Slides and Dots

```
const slides = [...document.querySelectorAll('.mySlides')];
const dots = [...document.querySelectorAll('.dot')];
```

This grabs all elements in the HTML that match .mySlides (each slide) and .dot (each navigation circle).

The [...] spread turns them from NodeList to regular arrays, which allows us to use .map(), .forEach(), .every(), etc.

Vanilla JavaScript NodeList returned by `querySelectorAll()` doesn't have full array methods. Spreading them turns them into full arrays.

2. Display a Specific Slide

```
let slideIndex = 0;

const showSlides = n => {
  slideIndex = (n + slides.length) % slides.length;

  slides.map((s, i) => s.style.display = i === slideIndex ? 'block' :
'none');
  dots.map((d, i) => d.classList.toggle('active', i === slideIndex));
};
```

Sets the current slideIndex, using (n + slides.length) % slides.length so that:

If n goes below 0, it wraps to the last slide.

If n goes above the last index, it wraps to the first slide.

Then:

slides.map() hides all slides, except the current one (i === slideIndex).

dots.map() removes the active class from all dots and adds it only to the current one.

Why use .map()? It's concise and functional — it applies a transformation to each element without needing a for loop.

3. Change Slides with Arrows or Dots

```
const plusSlides = n => showSlides(slideIndex + n);
const currentSlide = n => showSlides(n);
```

plusSlides(-1) goes backward.

plusSlides(1) goes forward.

currentSlide(n) jumps to slide n.

4. Add Event Listeners

```
document.querySelector('.prev')?.addEventListener('click', () =>
plusSlides(-1));
document.querySelector('.next')?.addEventListener('click', () =>
plusSlides(1));
```

These lines listen for clicks on .prev and .next buttons to move slides.

Note: The ?. (optional chaining) means "only add the event listener if the element exists." This prevents errors if, say, .prev is missing.

```
dots.forEach((dot, i) => dot.addEventListener('click', () =>
currentSlide(i)));
```

When a dot is clicked, it jumps to that index using currentSlide(i).

.forEach() is used here because we're not returning a new array—we're just running a side effect (adding an event listener).

5. Handle Edge Case: All Dots Inactive

```
const ensureOneActiveDot = () => {
  if (dots.every(d => !d.classList.contains('active'))) {
    dots[0].classList.add('active');
    slides[0].style.display = 'block';
  }
};
```

What it does:

Uses .every() to check if every dot is NOT active.

If so, it fixes it by activating the first slide and dot.

.every() checks that all elements meet a condition. Here, we use it as a safeguard in case all dots get deactivated (a rare bug or external interference).

6. Auto Slide Change Every 2 Seconds

```
setInterval(() => {
  plusSlides(1);
  ensureOneActiveDot();
}, 2000);
```

What it does:

Every 2 seconds, the slideshow moves forward one slide.

It also calls ensureOneActiveDot() to double-check the state.

setInterval is a built-in browser method that repeats the function at regular time intervals (2000 ms = 2 seconds).

Exercise 4: ES6, Arrays, DOM: Client-Side Search

1. Set up a keyup event listener

```
document.querySelector("#search").addEventListener("keyup", () => {
```

This listens for every key press inside the search box.

When the user types, the anonymous function is triggered.

2. Get the input value and convert it to uppercase

```
const filter = document.getElementById("search").value.toUpperCase();
```

Retrieves whatever the user typed into the search input.

Converts it to uppercase to make the search case-insensitive.

3. Get all <tr> (table rows)

```
const tr =
document.getElementById("infoTable").getElementsByTagName("tr");
```

This selects every row in the entire table (<thead> and <tbody> included).

We will later skip header rows automatically by checking for <td> cells.

4. Loop through each row

```
Array.from(tr).forEach(row => {
```

Converts the HTMLCollection of <tr> elements into an array.

Loops through each row in the table.

5. Check if the row has <td> cells

```
const tds = row.getElementsByTagName("td");
```

If the row has <td>s, it's a data row.

If not, it’s likely a header row, and we don’t want to hide those.

6. Search inside the row content

```
row.style.display =
  tds.length && row.innerHTML.toUpperCase().indexOf(filter) > -1
    ? ""
    : "none";
```

This checks if the uppercased row.innerHTML contains the search term (filter).

If it does:
→ style.display = "" (show the row)

If it doesn't:
→ style.display = "none" (hide the row)

Exercise 5: Classes - Single Page Application Quiz

This code powers a simple multiple-choice quiz app. It separates responsibilities into:

Quiz – tracks quiz state, score, and progression

Question – represents a single question and its correct answer

QuizUI – manages what's shown on the page and how users interact with it

1. class Quiz

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

score: starts at 0, increases with each correct answer

questions: an array of Question objects

currentQuestionIndex: keeps track of which question the user is currently on

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

guess(answer): checks the user's selected answer:

If correct, increments score

Always moves to the next question by increasing currentQuestionIndex

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

Returns the active Question object based on the current index.

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

Returns true when the quiz has gone through all questions.

2. class Question

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

text: the question itself

choices: an array of answer options

answer: the correct one

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

Compares the user's selected choice to the correct answer.

3. const QuizUI

The object that manages the display and user interaction.

4. displayNext()

```
displayNext() {
  if (quiz.hasEnded()) {
    this.displayScore();
  } else {
    this.displayQuestion();
    this.displayChoices();
    this.displayProgress();
  }
}
```

Main method that drives the quiz.

If the quiz is over, shows score.

Otherwise, updates:

The question text

The multiple-choice options

Progress (e.g. “Question 2 of 5”)

4. displayQuestion()

```
displayQuestion() {
  this.populateIdWithHTML("question", quiz.getCurrentQuestion().text);
}
```

Fills the HTML element with ID "question" with the text of the current question.

5. displayChoices()

```
displayChoices() {
  const choices = quiz.getCurrentQuestion().choices;

  for (let i = 0; i < choices.length; i++) {
    this.populateIdWithHTML("choice" + i, choices[i]);
    this.guessHandler("guess" + i, choices[i]);
  }
}
```

Gets all the possible answers for the current question.

For each choice:

Inserts it into the corresponding button or span (e.g., choice0, choice1, etc.)

Sets up event listener using guessHandler() to respond when the user selects an answer

6. displayScore()

```
displayScore() {
  const gameOverHTML = `<h2>Results</h2>
                         <h2> Your score is: ${quiz.score}/$
{quiz.questions.length}</h2>`;
  this.populateIdWithHTML("quiz", gameOverHTML);
}
```

Creates a message showing the final score

Replaces the content of the entire quiz container with that message

7. populateIdWithHTML(id, text)

```
populateIdWithHTML(id, text) {
  document.getElementById(id).innerHTML = text;
}
```

A reusable helper method that updates the HTML inside a given element by id

8. guessHandler(id, guess)

```
guessHandler(id, guess) {
  document.getElementById(id).onclick = () => {
    quiz.guess(guess);
    this.displayNext();
  };
}
```

Attaches a click handler to the answer button.

When clicked:

Calls quiz.guess() with the selected answer

Then calls displayNext() to update the display