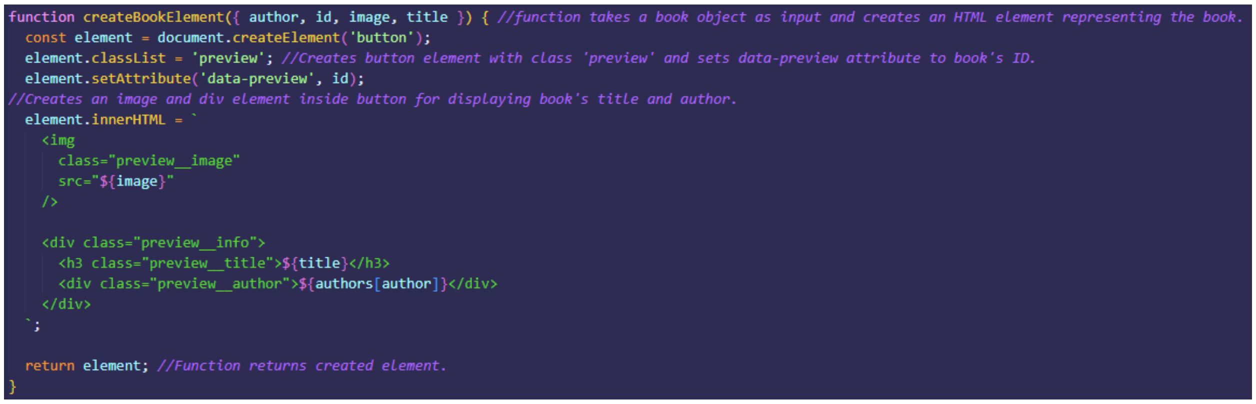
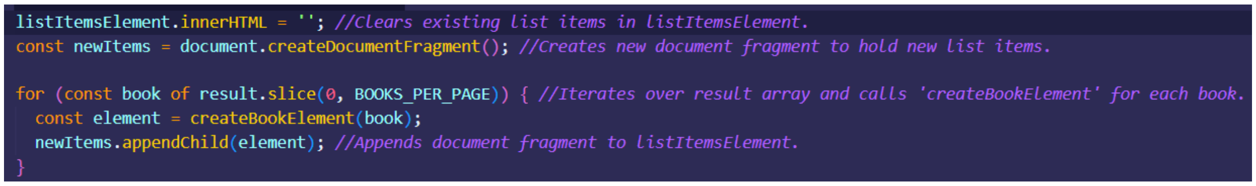
DWA_07.4 Knowledge Check_DWA7

1. Which were the three best abstractions, and why?



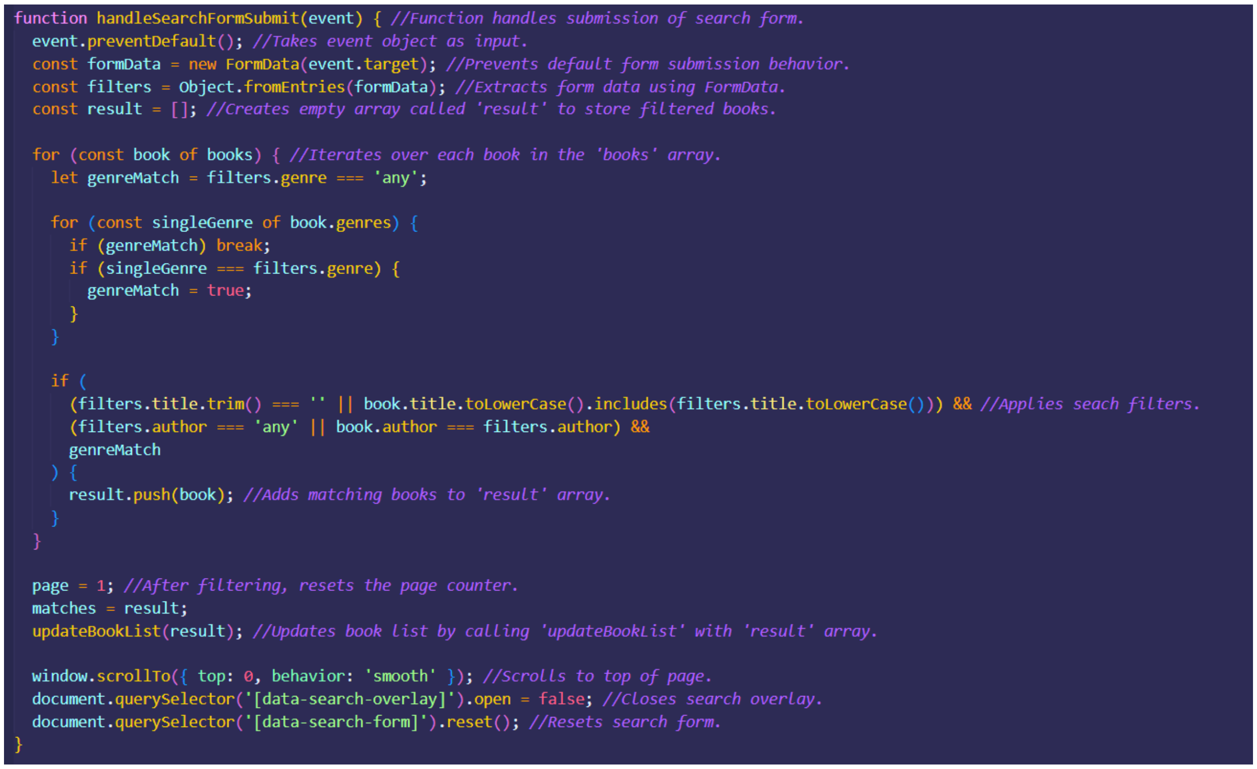
```
function createBookElement({ author, id, image, title }) { //function takes a book object as input and creates an HTML element representing the book.
  const element = document.createElement('button');
  element.classList = 'preview'; //Creates button element with class 'preview' and sets data-preview attribute to book's ID.
  element.setAttribute('data-preview', id);
//Creates an image and div element inside button for displaying book's title and author.
  element.innerHTML = `
    <img
      class="preview__image"
      src="${image}"
    />

    <div class="preview__info">
      <h3 class="preview__title">${title}</h3>
      <div class="preview__author">${authors[author]}</div>
    </div>
  `;

  return element; //Function returns created element.
}
```

-Firstly, I found the createBookElement function to be extremely useful as it encapsulates the process of creating an HTML element representing a book. Through abstracting this functionality into a separate function, the code is more modular and reusable. The main purpose of the function is to take a book object as input and return it as an HTML element representing a book. This allows the function to be used in different parts of the codebase whenever a book element needs to be created. It prevents code duplication, which is helpful in this case as there are too many book elements to account for individually. It also allows for adjustments to be made more easily if the structure or style of the book elements needs to be modified in the future (calling code remains unchanged since it relies on the abstraction provided by the function).



```
listItemsElement.innerHTML = ''; //Clears existing list items in listItemsElement.
const newItems = document.createDocumentFragment(); //Creates new document fragment to hold new list items.

for (const book of result.slice(0, BOOKS_PER_PAGE)) { //Iterates over result array and calls 'createBookElement' for each book.
  const element = createBookElement(book);
  newItems.appendChild(element); //Appends document fragment to listItemsElement.
}
```

-Secondly, the use of abstraction in this code is helpful because it encapsulates the process of generating and appending new list items to the listItemsElement. As per the point of abstraction, it makes the code easier to understand and more modular. The code is divided into clear steps; firstly the existing list items are cleared using listItemsElement.innerHTML = ‘ ‘ (which essentially clears any existing list items by setting its innerHTML to an empty string), secondly, a new document fragment is created using document.createDocumentFragment() and lastly, the loop iterates over the result array, calls the the createBookElement function for each book and appends the generated elements to the document fragment. The process ultimately allows for dynamically updating the list of books displayed on the web page based on the result array.

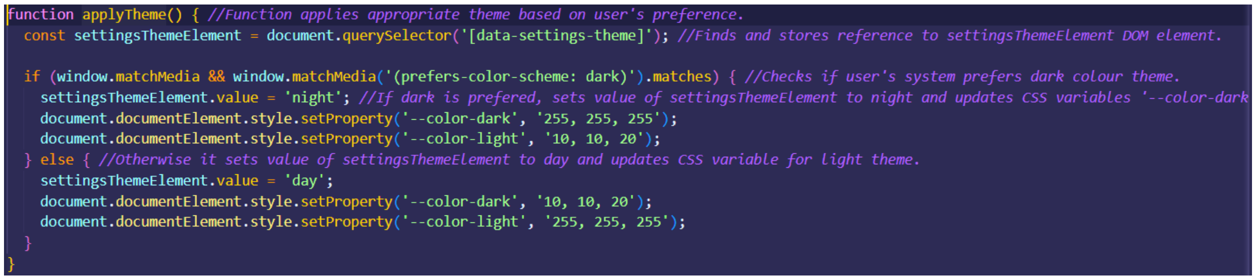


```
function handleSearchFormSubmit(event) { //Function handles submission of search form.
  event.preventDefault(); //Takes event object as input.
  const formData = new FormData(event.target); //Prevents default form submission behavior.
  const filters = Object.fromEntries(formData); //Extracts form data using FormData.
  const result = []; //Creates empty array called 'result' to store filtered books.

  for (const book of books) { //Iterates over each book in the 'books' array.
    let genreMatch = filters.genre === 'any';

    for (const singleGenre of book.genres) {
      if (genreMatch) break;
      if (singleGenre === filters.genre) {
        genreMatch = true;
      }
    }

    if (
      (filters.title.trim() === '' || book.title.toLowerCase().includes(filters.title.toLowerCase())) && //Applies seach filters.
      (filters.author === 'any' || book.author === filters.author) &&
      genreMatch
    ) {
      result.push(book); //Adds matching books to 'result' array.
    }
  }

  page = 1; //After filtering, resets the page counter.
  matches = result;
  updateBookList(result); //Updates book list by calling 'updateBookList' with 'result' array.

  window.scrollTo({ top: 0, behavior: 'smooth' }); //Scrolls to top of page.
  document.querySelector('[data-search-overlay]').open = false; //Closes search overlay.
  document.querySelector('[data-search-form]').reset(); //Resets search form.
}
```

-Thirdly, abstraction is used in this piece of code to encapsulate the logic of handling the search form submission into a single function (handleSearchFormSubmit). It essentially abstracts away the details of form submission, data extraction, filtering and updating the book list while also making it more readable and reusable. Furthermore, it separates the concern of handling form submission and filtering books from other parts of the code, promoting the principle of single responsibility, where each function focuses on a specific task. It also prevents default form submission behavior which is helpful as it prevents the page from reloading. It extracts the data entered in the form using the FormData API and converts it into an object called filters. This object holds the search criteria entered by the user such as the title, author and genre. After filtering the books, the function resets the page counter to 1, assigns the filtered result array to the matches variable and calls the updateBookList function with the result array as an argument. The updateBookList function is responsible for updating the book list display based on the filtered books. Lastly, the function scrolls the page to the top using window.scrollTo() with a smooth scrolling behavior. It then closes the search overlay by setting the open property of the search overlay element to false, followed by resetting the search form to clear the form inputs. Essentially, the handleSearchFormSubmit function extracts the search criteria from the form, filters the books based on those criteria, updates the book list display, scrolls to the top of the page and finally resets the search form.

---

2. Which were the three worst abstractions, and why?



```javascript
function applyTheme() { //Function applies appropriate theme based on user's preference.
  const settingsThemeElement = document.querySelector('[data-settings-theme]'); //Finds and stores reference to settingsThemeElement DOM element.

  if (window.matchMedia && window.matchMedia('(prefers-color-scheme: dark)').matches) { //Checks if user's system prefers dark colour theme.
    settingsThemeElement.value = 'night'; //If dark is prefered, sets value of settingsThemeElement to night and updates CSS variables '--color-dark
    document.documentElement.style.setProperty('--color-dark', '255, 255, 255');
    document.documentElement.style.setProperty('--color-light', '10, 10, 20');
  } else { //Otherwise it sets value of settingsThemeElement to day and updates CSS variable for light theme.
    settingsThemeElement.value = 'day';
    document.documentElement.style.setProperty('--color-dark', '10, 10, 20');
    document.documentElement.style.setProperty('--color-light', '255, 255, 255');
  }
}
```

-Firstly, in the above code, abstraction is used to encapsulate the functionality related to applying a theme into the applyTheme function, abstracting the theme application logic and making it separate and reusable. It retrieves a reference to the settingsThemeElement DOM element and helps to find and store the element for later use. It also uses a conditional statement to check if the user prefers a dark color theme, allowing for the application of different CSS variables and values based on the condition. It encapsulates the theme-specific logic within the if-else block, providing abstraction for applying the appropriate theme. Although abstraction is present in this code, it is relatively simple and is focused on encapsulating specific tasks and logic, which in this case, are the applyTheme, handleSearchFormSubmit and handleListButtonClick functions.

---

3. How can The three worst abstractions be improved via SOLID principles.

1. -Single Responsibility Principle (SRP):
   - Split the updateBookList function into smaller functions, each responsible for a specific part of the updating process, such as clearing the list, creating book elements, and updating the button.
   - Move the logic for applying the theme to a separate function instead of combining it with the theme element retrieval.

Open/Closed Principle (OCP)

- Create an abstraction or interface for the theme application, allowing for different theme strategies to be implemented and easily swapped without modifying the code that applies the theme.
- Abstract the DOM element retrieval by introducing a separate module or class responsible for accessing and manipulating DOM elements.

Interface Segregation Principle (ISP)

- Analyze the functions and modules that rely on external dependencies, such as DOM elements or data sources. Ensure that they only depend on the necessary interfaces or abstractions and avoid exposing irrelevant methods or properties.

Dependency Inversion Principle (DIP)

- Introduce interfaces or abstractions to decouple modules from specific implementations. For example, create an interface for the theme application, allowing different theme strategies to be implemented without affecting the modules that use the theme.

2.
