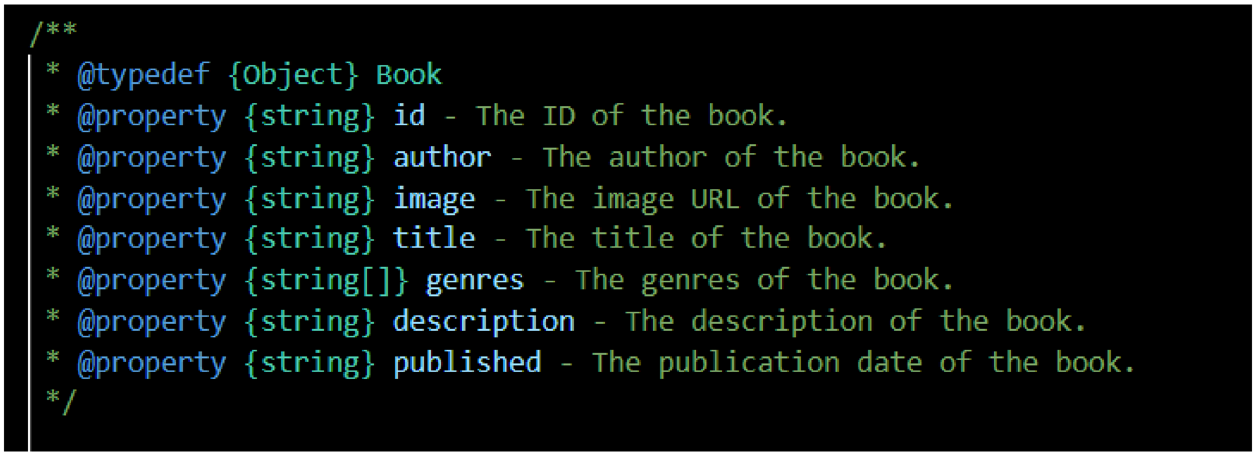
# DWA_07.4 Knowledge Check_DWA7

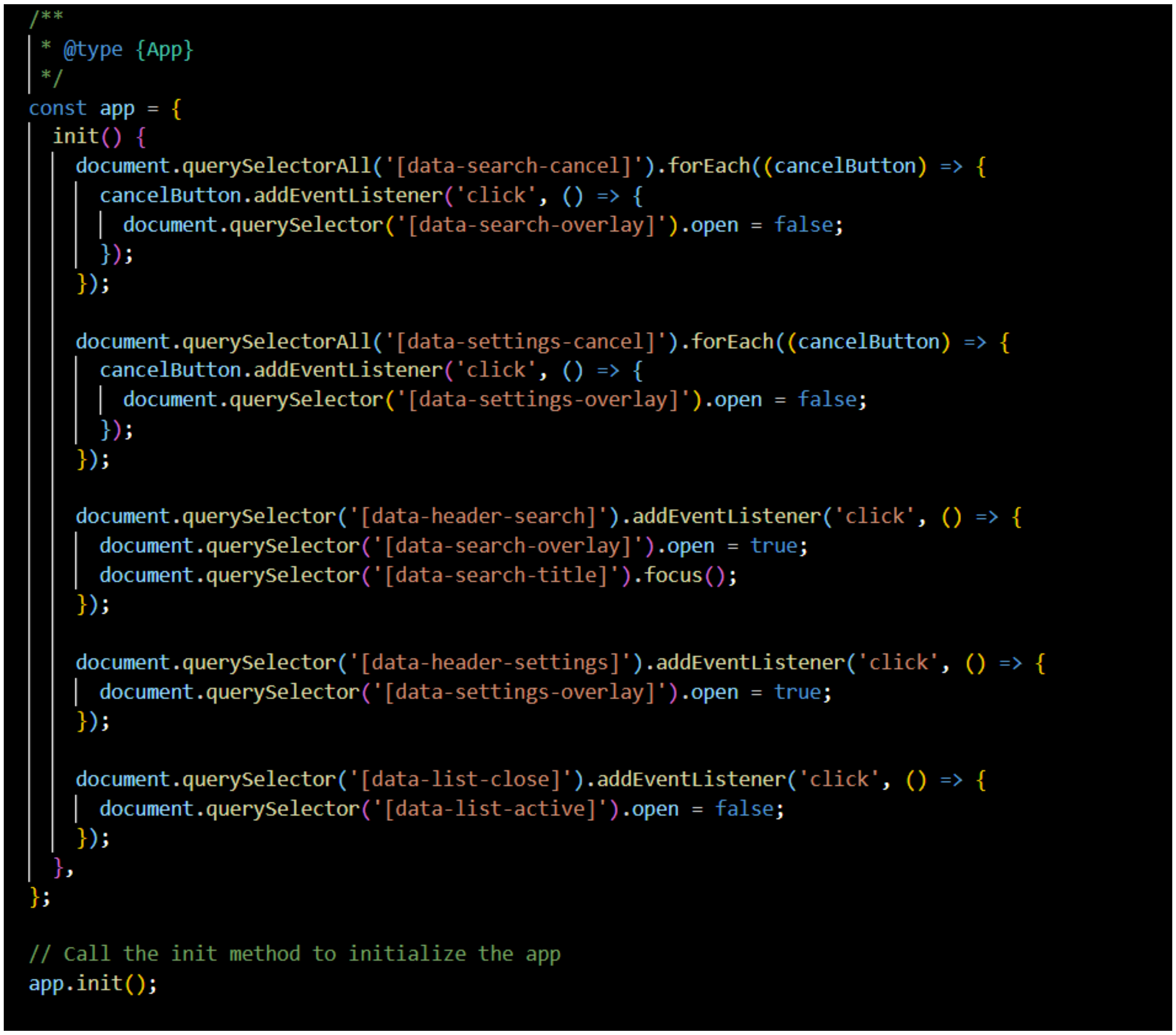
1. Which were the three best abstractions, and why?



```
/**
 * @typedef {Object} Book
 * @property {string} id - The ID of the book.
 * @property {string} author - The author of the book.
 * @property {string} image - The image URL of the book.
 * @property {string} title - The title of the book.
 * @property {string[]} genres - The genres of the book.
 * @property {string} description - The description of the book.
 * @property {string} published - The publication date of the book.
 */
```

**1. Book object/ typedef**
Using the Book abstraction, we can organize and manage book-related data in a consistent manner. It provides a clear structure for storing information such as the book's ID, author, image URL, title, genres, description, and publication date. This makes it easier to understand and work with book objects throughout the codebase.

This provides a form of abstraction by abstracting away the implementation details of the underlying objects and allowing developers to work with well-defined types.
Helps improve code organization, readability, and maintainability. Instead of scattering book-related properties and their associated logic across different parts of the code promoting a cleaner and more structured code, making it easier to understand and modify book-related functionality as needed.



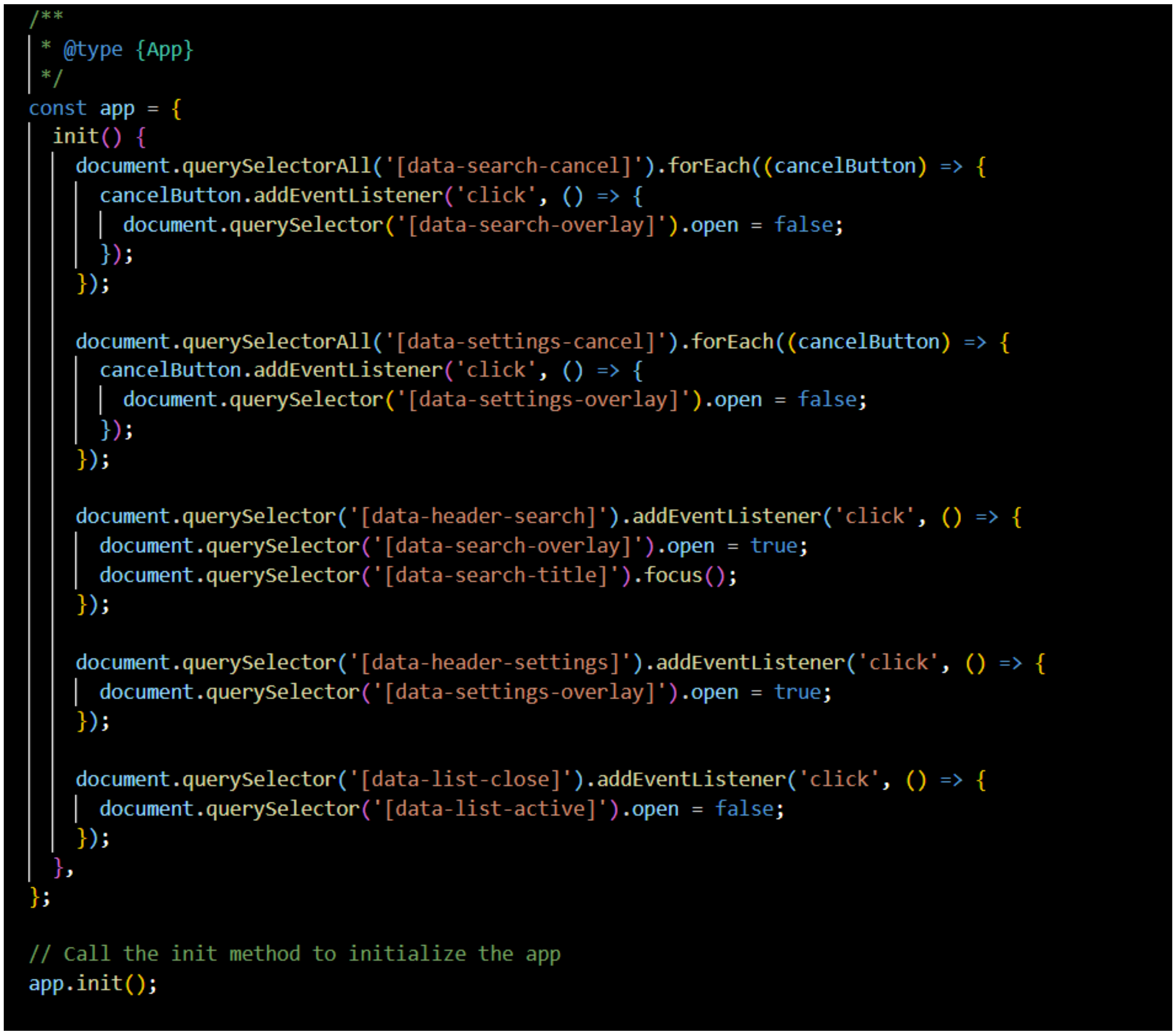
```
/**
 * @type {App}
 */
const app = {
  init() {
    document.querySelectorAll('[data-search-cancel]').forEach((cancelButton) => {
      cancelButton.addEventListener('click', () => {
        document.querySelector('[data-search-overlay]').open = false;
      });
    });

    document.querySelectorAll('[data-settings-cancel]').forEach((cancelButton) => {
      cancelButton.addEventListener('click', () => {
        document.querySelector('[data-settings-overlay]').open = false;
      });
    });

    document.querySelector('[data-header-search]').addEventListener('click', () => {
      document.querySelector('[data-search-overlay]').open = true;
      document.querySelector('[data-search-title]').focus();
    });

    document.querySelector('[data-header-settings]').addEventListener('click', () => {
      document.querySelector('[data-settings-overlay]').open = true;
    });

    document.querySelector('[data-list-close]').addEventListener('click', () => {
      document.querySelector('[data-list-active]').open = false;
    });
  },
};

// Call the init method to initialize the app
app.init();
```

**2. App.init**

All the event listeners are in the same function making it easier to initialize processes to be easily executed in a consistent and reusable manner across different parts of the codebase.

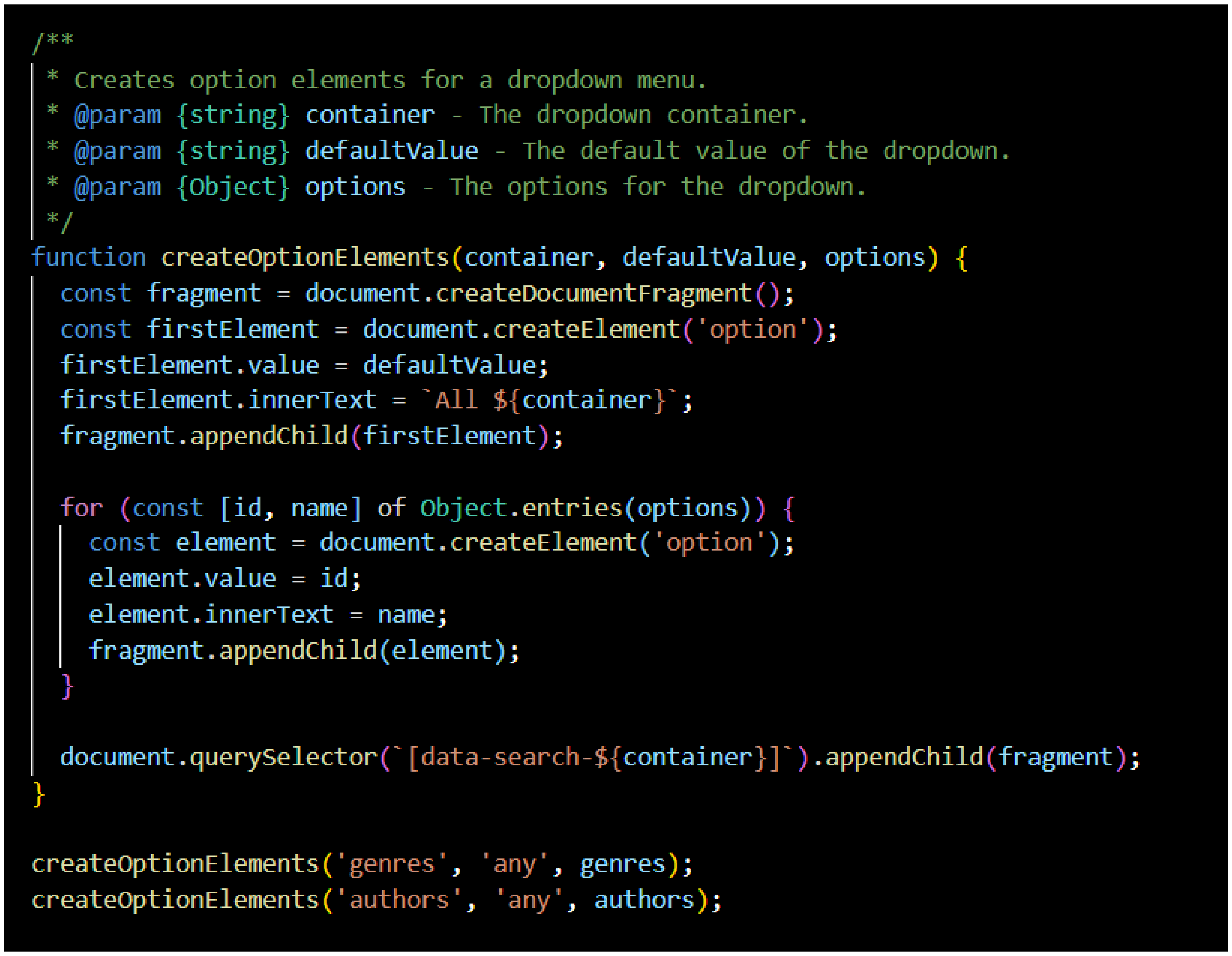
---

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



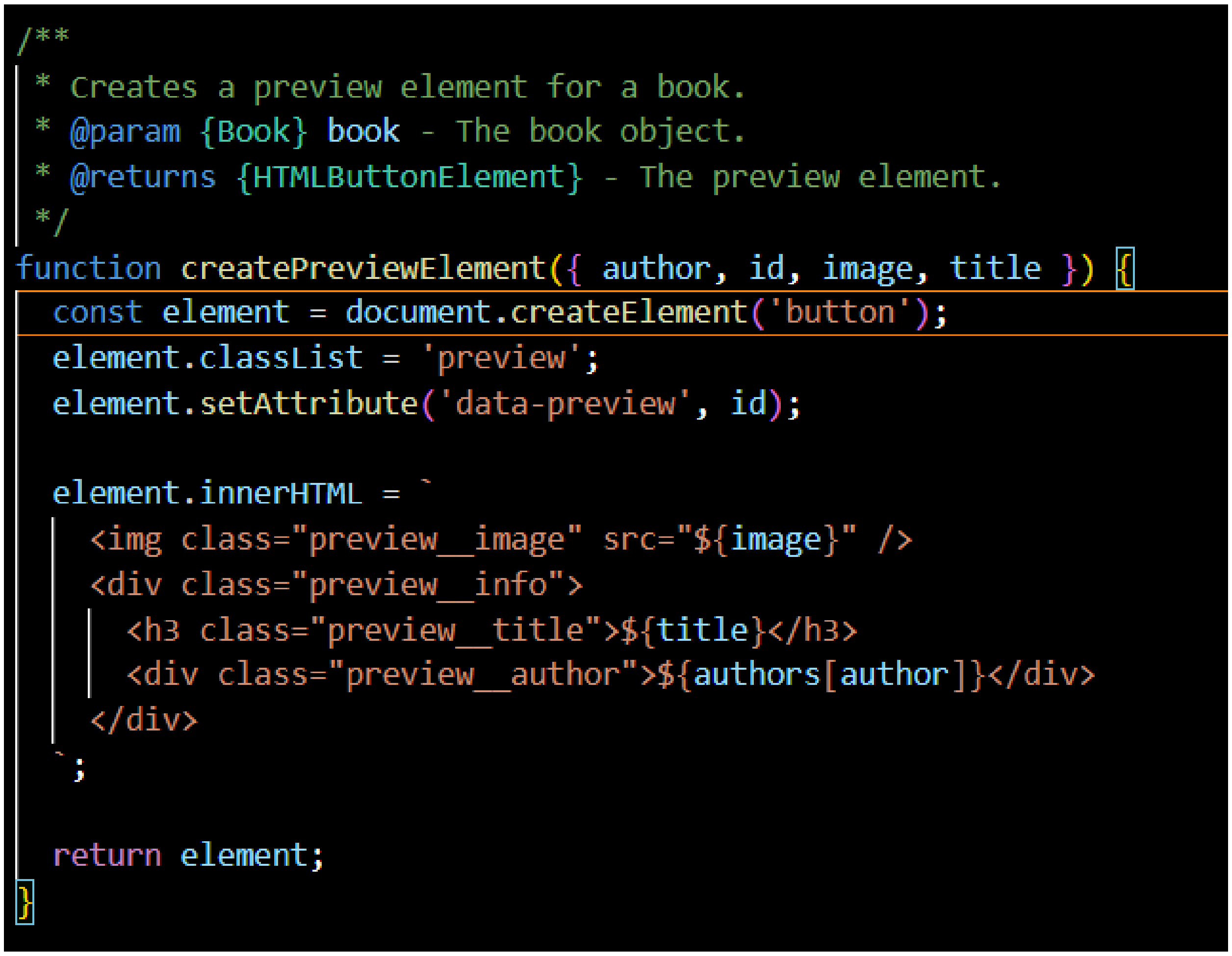
```
/**
 * @type {App}
 */
const app = {
  init() {
    document.querySelectorAll('[data-search-cancel]').forEach((cancelButton) => {
      cancelButton.addEventListener('click', () => {
        document.querySelector('[data-search-overlay]').open = false;
      });
    });

    document.querySelectorAll('[data-settings-cancel]').forEach((cancelButton) => {
      cancelButton.addEventListener('click', () => {
        document.querySelector('[data-settings-overlay]').open = false;
      });
    });

    document.querySelector('[data-header-search]').addEventListener('click', () => {
      document.querySelector('[data-search-overlay]').open = true;
      document.querySelector('[data-search-title]').focus();
    });

    document.querySelector('[data-header-settings]').addEventListener('click', () => {
      document.querySelector('[data-settings-overlay]').open = true;
    });

    document.querySelector('[data-list-close]').addEventListener('click', () => {
      document.querySelector('[data-list-active]').open = false;
    });
  },
};

// Call the init method to initialize the app
app.init();
```

**1. App.init**
The function is doing more than it should, setting event listeners for different user interface elements.
It has multiple tasks / responsibilities.



```javascript
/**
 * Creates option elements for a dropdown menu.
 * @param {string} container - The dropdown container.
 * @param {string} defaultValue - The default value of the dropdown.
 * @param {Object} options - The options for the dropdown.
 */
function createOptionElements(container, defaultValue, options) {
  const fragment = document.createDocumentFragment();
  const firstElement = document.createElement('option');
  firstElement.value = defaultValue;
  firstElement.innerText = `All ${container}`;
  fragment.appendChild(firstElement);

  for (const [id, name] of Object.entries(options)) {
    const element = document.createElement('option');
    element.value = id;
    element.innerText = name;
    fragment.appendChild(element);
  }

  document.querySelector(`[data-search-${container}]`).appendChild(fragment);
}

createOptionElements('genres', 'any', genres);
createOptionElements('authors', 'any', authors);
```

**2. createOptionElements**
the function not only creates the option elements but also directly appends them to the DOM within the function itself. This means that the function is tightly coupled with the specific way the options are added to the DOM. This violates the Open-Closed Principle (OCP), which suggests that code should be open for extension but closed for modification.



```
/**
 * Creates a preview element for a book.
 * @param {Book} book - The book object.
 * @returns {HTMLButtonElement} - The preview element.
 */
function createPreviewElement({ author, id, image, title }) {
  const element = document.createElement('button');
  element.classList = 'preview';
  element.setAttribute('data-preview', id);

  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;
}
```

**3. createPreviewElement**
It has a limitation in the HTML structure of the preview element.
The function directly includes the HTML structure of the preview element within the function itself. This means that if we want to change the appearance or customize the structure of the preview element, we would need to modify the function itself. This tight coupling between the HTML structure and the function logic makes it less flexible and harder to modify.

---

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

**App.init:**

- Follow the Single Responsibility Principle by separating the tasks into separate functions and classes.
- Each function or class would be responsible for setting up the event  listener for a specific UI element.
- That will make the functions clear on what they do, allowing for better reusability to be able to use the code on other parts of the code if needed.

**createOptionElements:**

- Instead of directly appending the created option elements to the DOM within the function.
- Make the function return the created option elements as a result. Then, let the caller of the function be responsible for appending them to the DOM.
- This separation will allow for easier extension or modification of the DOM manipulation behavior without modifying the function itself.
- That will provide more flexibility in how the option elements can be used in different scenarios.

**createPreviewElement**

- Separate the HTML structure of the preview element from the function logic.
- A separate template or component can be used then the function could then use either one to create a preview element.
- By separating the HTML structure into a template or component, we can modify or customize the appearance of the preview element without needing to modify the core logic of the createPreviewElement function. It allows for more flexibility and easier customization of the preview element.