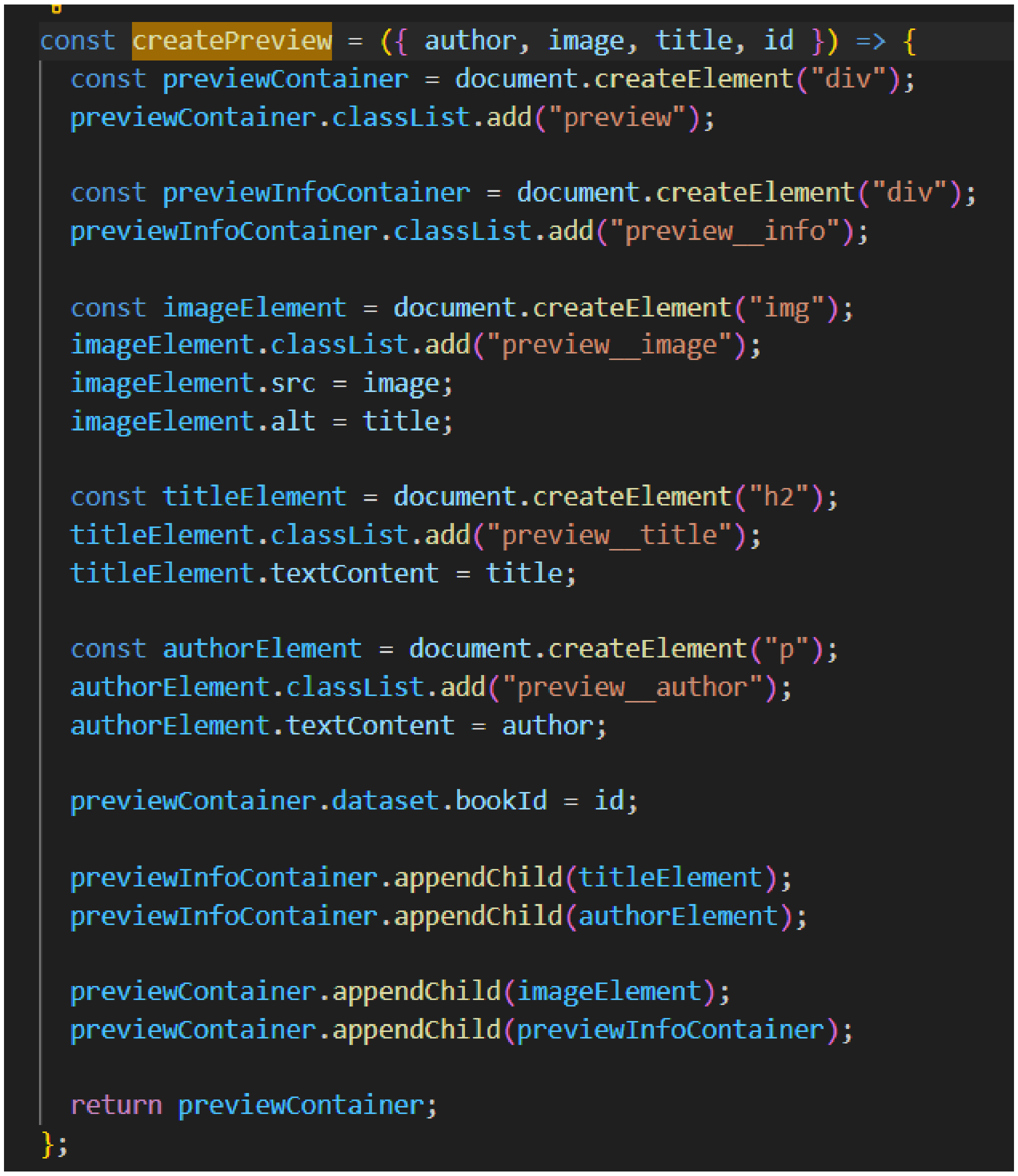
DWA_07.4 Knowledge Check_DWA7

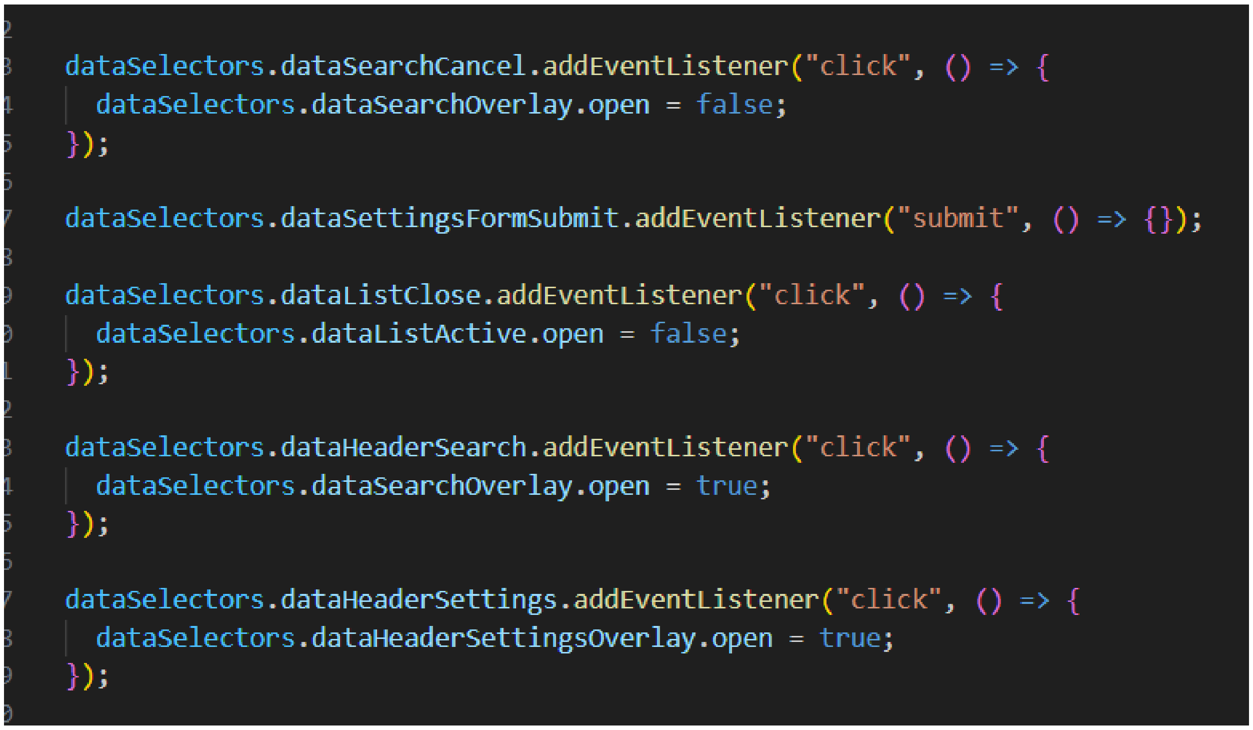
1. Which were the three best abstractions, and why?

Creating a function that will be re-used in more than one instance was one of the best abstractions because it made the code more manageable and prevented repetitive code that would do the same thing in multiple places in the file.



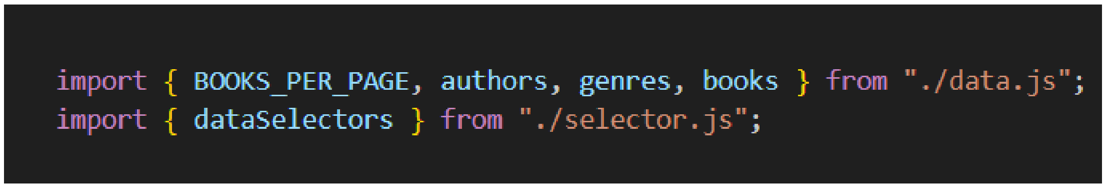
```javascript
const createPreview = ({ author, image, title, id }) => {
  const previewContainer = document.createElement("div");
  previewContainer.classList.add("preview");

  const previewInfoContainer = document.createElement("div");
  previewInfoContainer.classList.add("preview__info");

  const imageElement = document.createElement("img");
  imageElement.classList.add("preview__image");
  imageElement.src = image;
  imageElement.alt = title;

  const titleElement = document.createElement("h2");
  titleElement.classList.add("preview__title");
  titleElement.textContent = title;

  const authorElement = document.createElement("p");
  authorElement.classList.add("preview__author");
  authorElement.textContent = author;

  previewContainer.dataset.bookId = id;

  previewInfoContainer.appendChild(titleElement);
  previewInfoContainer.appendChild(authorElement);

  previewContainer.appendChild(imageElement);
  previewContainer.appendChild(previewInfoContainer);

  return previewContainer;
};
```

Adding all query selectors into an object so that it reduces repetition of using the same query selectors in multiple places:



```
dataSelectors.dataSearchCancel.addEventListener("click", () => {
  dataSelectors.dataSearchOverlay.open = false;
});

dataSelectors.dataSettingsFormSubmit.addEventListener("submit", () => {});

dataSelectors.dataListClose.addEventListener("click", () => {
  dataSelectors.dataListActive.open = false;
});

dataSelectors.dataHeaderSearch.addEventListener("click", () => {
  dataSelectors.dataSearchOverlay.open = true;
});

dataSelectors.dataHeaderSettings.addEventListener("click", () => {
  dataSelectors.dataHeaderSettingsOverlay.open = true;
});
```

Importing variables from different javascript files to reduce the amount of code in the main file and improve readability.

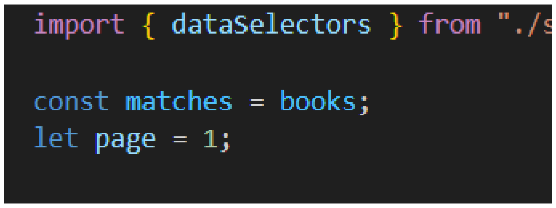


```
import { BOOKS_PER_PAGE, authors, genres, books } from "./data.js";
import { dataSelectors } from "./selector.js";
```

---

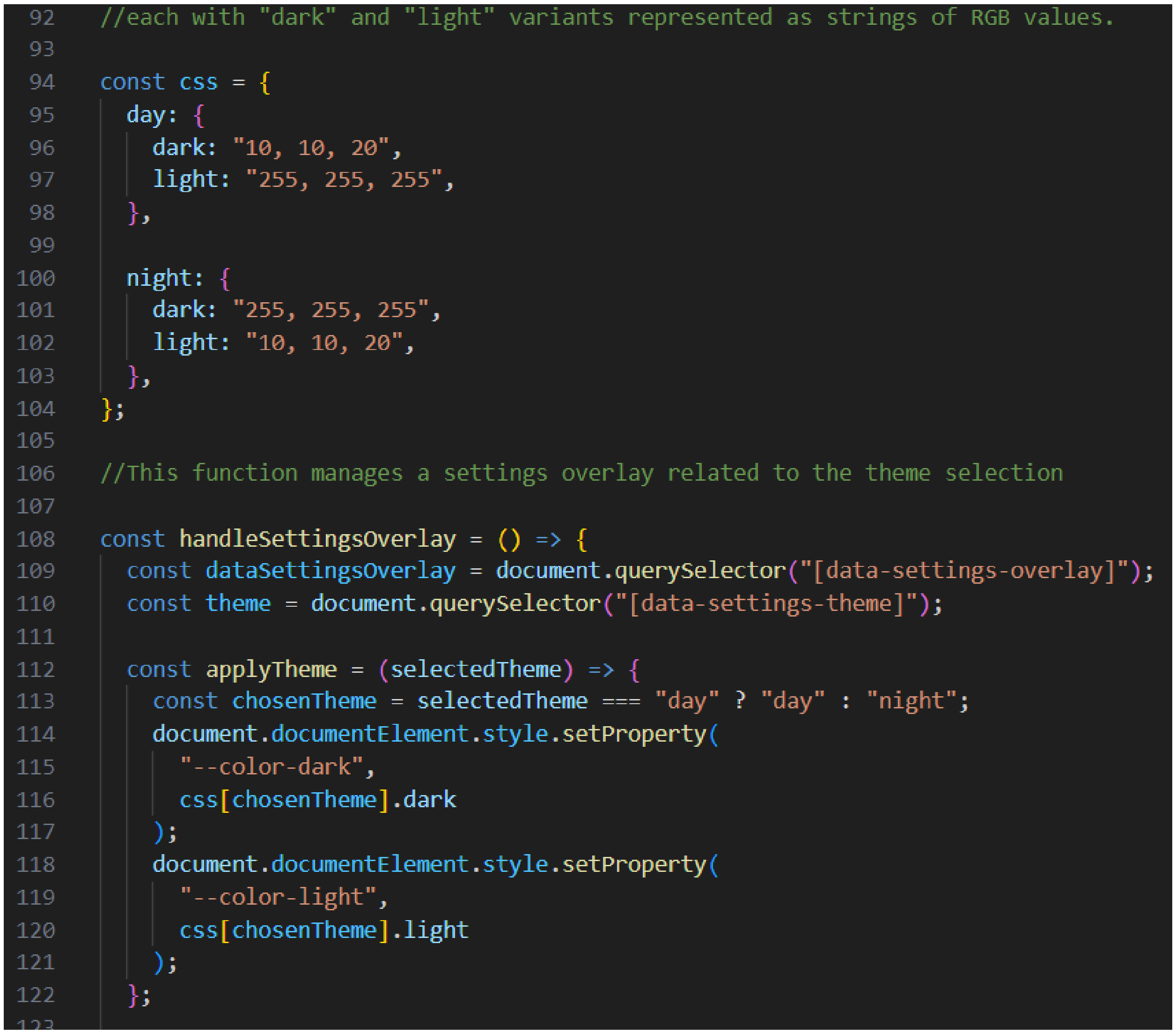
2. Which were the three worst abstractions, and why?

The global variables are not encapsulated and documented as to what their purpose is, this can lead to confusion for the next developer as everything may be left to assumptions and that can end up introducing bugs.



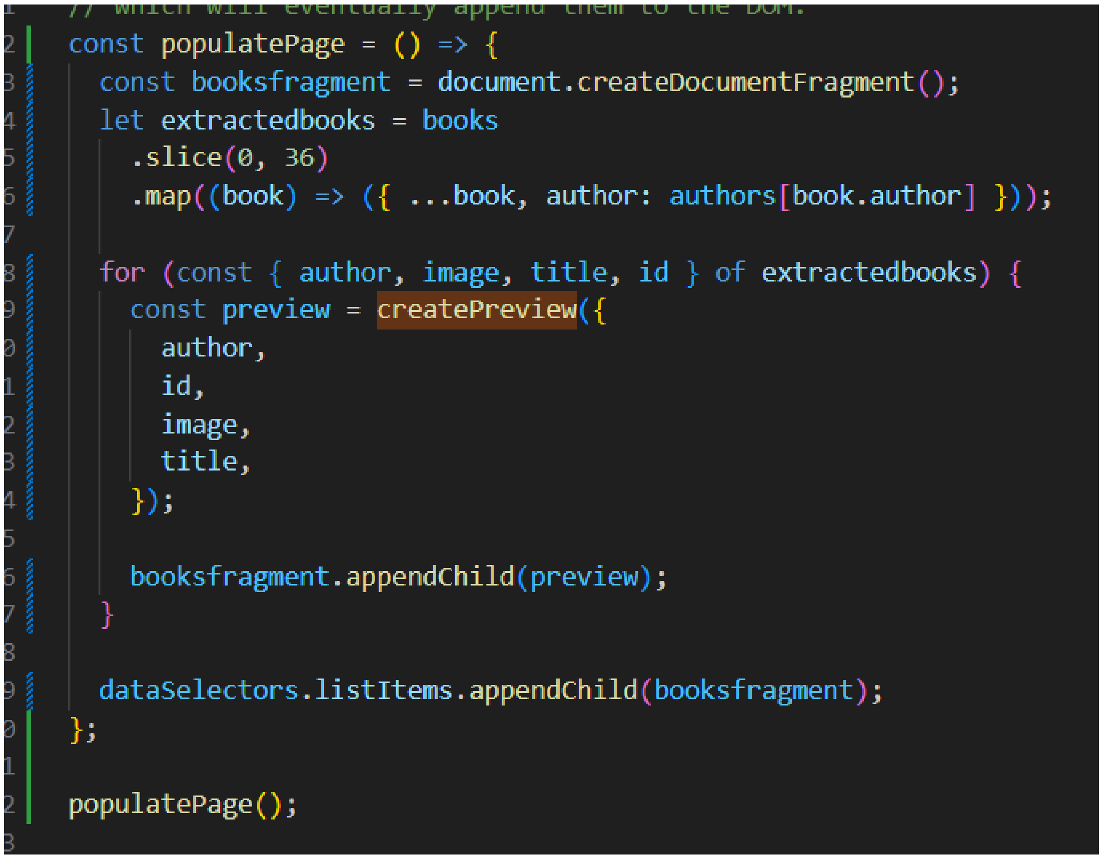
```
import { dataSelectors } from "./s

const matches = books;
let page = 1;
```

This directly affects the DOM elements on the page instead of creating new ones, which may end up causing issues in terms of re-usability and management of the themes on the page.



```
//each with "dark" and "light" variants represented as strings of RGB values.

const css = {
  day: {
    dark: "10, 10, 20",
    light: "255, 255, 255",
  },

  night: {
    dark: "255, 255, 255",
    light: "10, 10, 20",
  },
};

//This function manages a settings overlay related to the theme selection

const handleSettingsOverlay = () => {
  const dataSettingsOverlay = document.querySelector("[data-settings-overlay]");
  const theme = document.querySelector("[data-settings-theme]");

  const applyTheme = (selectedTheme) => {
    const chosenTheme = selectedTheme === "day" ? "day" : "night";
    document.documentElement.style.setProperty(
      "--color-dark",
      css[chosenTheme].dark
    );
    document.documentElement.style.setProperty(
      "--color-light",
      css[chosenTheme].light
    );
  };
```

This function is highly specific and tightly coupled to the DOM structure and specific global data (books, authors, genres). This makes the code less flexible and harder to reuse or test.



```
// which will eventually append them to the DOM.
const populatePage = () => {
  const booksfragment = document.createDocumentFragment();
  let extractedbooks = books
    .slice(0, 36)
    .map((book) => ({ ...book, author: authors[book.author] }));

  for (const { author, image, title, id } of extractedbooks) {
    const preview = createPreview({
      author,
      id,
      image,
      title,
    });

    booksfragment.appendChild(preview);
  }

  dataSelectors.listItems.appendChild(booksfragment);
};

populatePage();
```

---

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

1. **SOLID Principle**: Single Responsibility Principle (SRP) and Open/Closed Principle (OCP)
   Improvement: Modify these functions to accept parameters for the data they need to operate on, rather than relying on global variables. This change would make the functions more modular and easier to test. More generalized functions can be created for creating and appending DOM elements, which could be reused where necessary.

2. **SOLID Principle:** Single Responsibility Principle (SRP) and Dependency Inversion Principle (DIP)

Improvement: Separate the concerns by creating a ThemeManager module that handles theme data and logic, and a separate UI class or function that interacts with the DOM based on commands from the ThemeManager. This way, the theme management logic is decoupled from the DOM manipulation logic, making the codebase more maintainable and scalable.

3. **SOLID Principle**: Single Responsibility Principle (SRP), Open/Closed Principle (OCP), and Liskov Substitution Principle (LSP)
   Improvement: Seperate related functionalities into modules with well-defined interfaces. For instance, book management (pagination, filtering, displaying) could be encapsulated in a BookManager Module. This module would manage the state and operations related to books, making the code easier to understand and modify without affecting unrelated parts of the system. Utilize inheritance or interfaces to define common behaviors that allow for easy extension or substitution of components.
