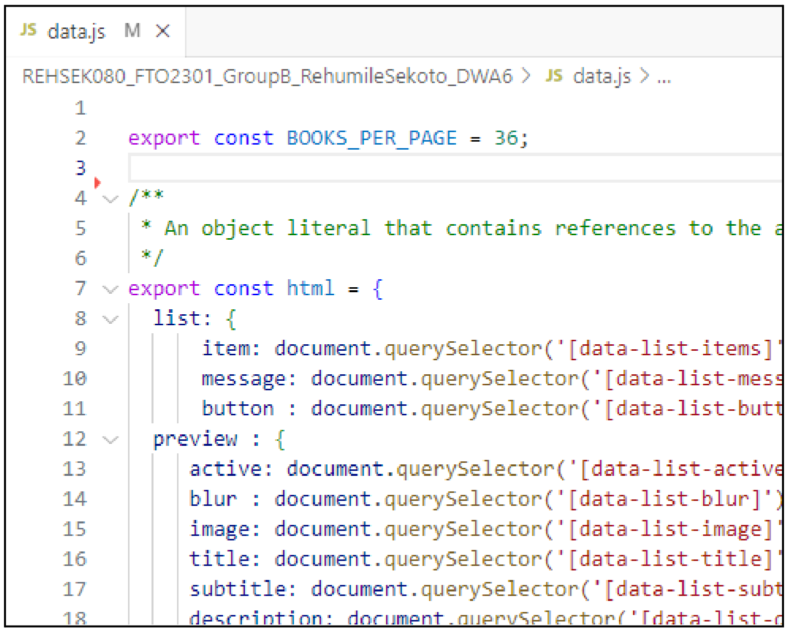
DWA_07.4 Knowledge Check_DWA7

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

The best abstraction in the Book Connect codebase is:

- Data abstractions. In this context, the authors object, genres object, books array and the html object. This kind of abstraction provides a clear separation of data from the rest of the code. They encapsulate relevant data and make it easily accessible to the other parts of the codebase.

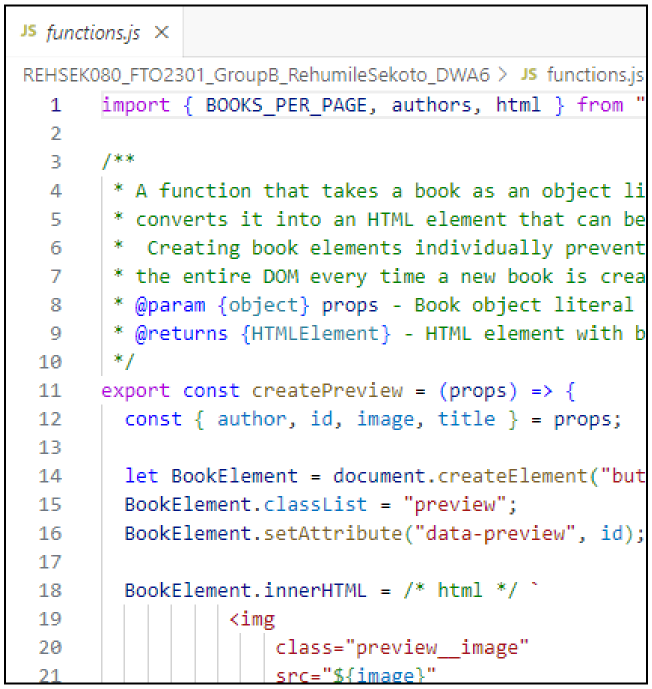


JS data.js M ×

REHSEK080_FTO2301_GroupB_RehumileSekoto_DWA6 > JS data.js > ...

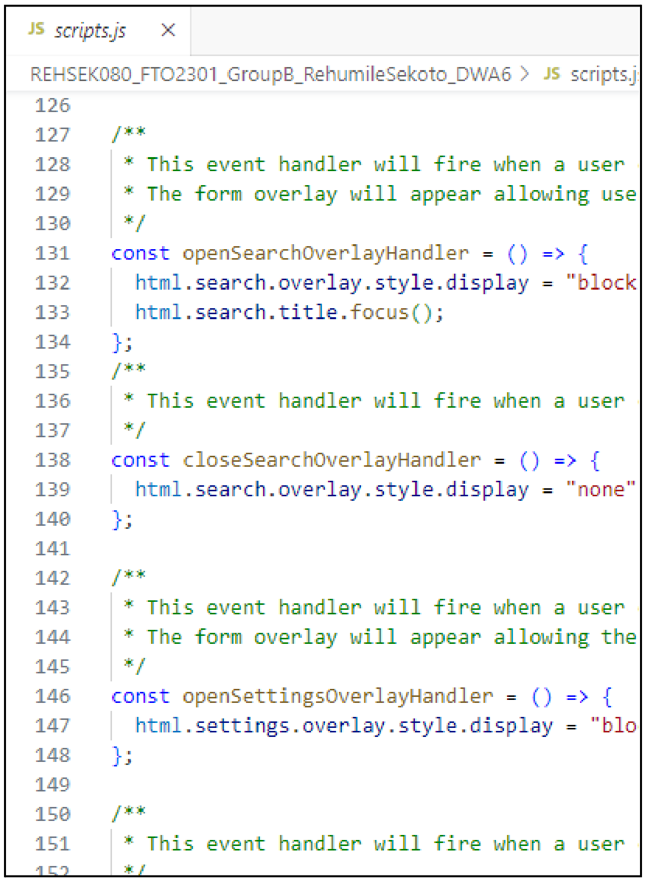
```

export const BOOKS_PER_PAGE = 36;

/**
 * An object literal that contains references to the a
 */
export const html = {
  list: {
      item: document.querySelector('[data-list-items]'
      message: document.querySelector('[data-list-mess
      button : document.querySelector('[data-list-butt
  preview : {
     active: document.querySelector('[data-list-active
     blur : document.querySelector('[data-list-blur]')
     image: document.querySelector('[data-list-image]'
     title: document.querySelector('[data-list-title]'
     subtitle: document.querySelector('[data-list-subt
     description: document.querySelector('[data-list-d
```

- Function Abstractions. So I created a separate module which includes functions not necessary for the main script. The populateDropMenu, setThemeColors, createPreviewFragment and displayBooks. This is a good abstraction because it encapsulates only the specific functionality which makes the code more modular and reusable. This also improves the organization and readability of the codebase.



```javascript
import { BOOKS_PER_PAGE, authors, html } from "

/**
 * A function that takes a book as an object li
 * converts it into an HTML element that can be
 *  Creating book elements individually prevent
 * the entire DOM every time a new book is crea
 * @param {object} props - Book object literal
 * @returns {HTMLElement} - HTML element with b
 */
export const createPreview = (props) => {
  const { author, id, image, title } = props;

  let BookElement = document.createElement("but
  BookElement.classList = "preview";
  BookElement.setAttribute("data-preview", id);

  BookElement.innerHTML = /* html */ `
          <img
            class="preview__image"
            src="${image}"
```

- Event Handler Abstractions. These abstractions only contain the essential actions to be taken when certain events occur. This provides a clear separation of concerns and improves the code organization.

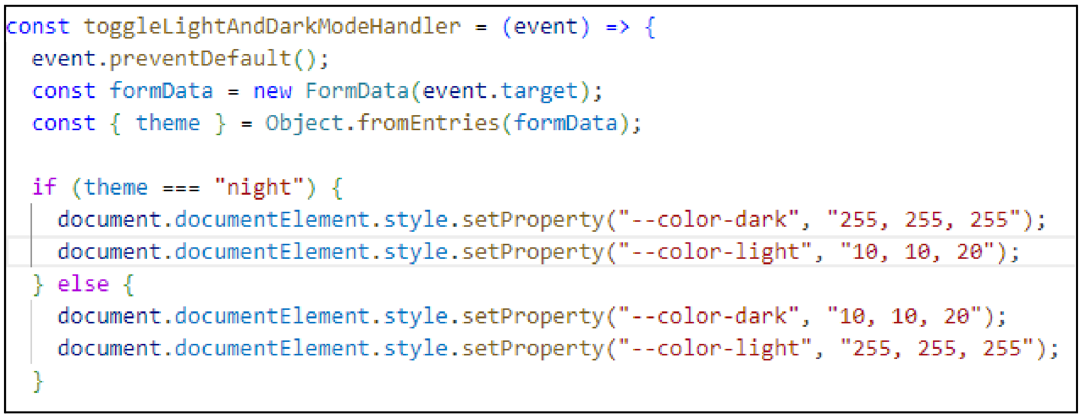


```javascript

 /**
  * This event handler will fire when a user
  * The form overlay will appear allowing use
  */
 const openSearchOverlayHandler = () => {
   html.search.overlay.style.display = "block
   html.search.title.focus();
 };
 /**
  * This event handler will fire when a user
  */
 const closeSearchOverlayHandler = () => {
   html.search.overlay.style.display = "none"
 };

 /**
  * This event handler will fire when a user
  * The form overlay will appear allowing the
  */
 const openSettingsOverlayHandler = () => {
   html.settings.overlay.style.display = "blo
 };

 /**
  * This event handler will fire when a user
  */
```

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

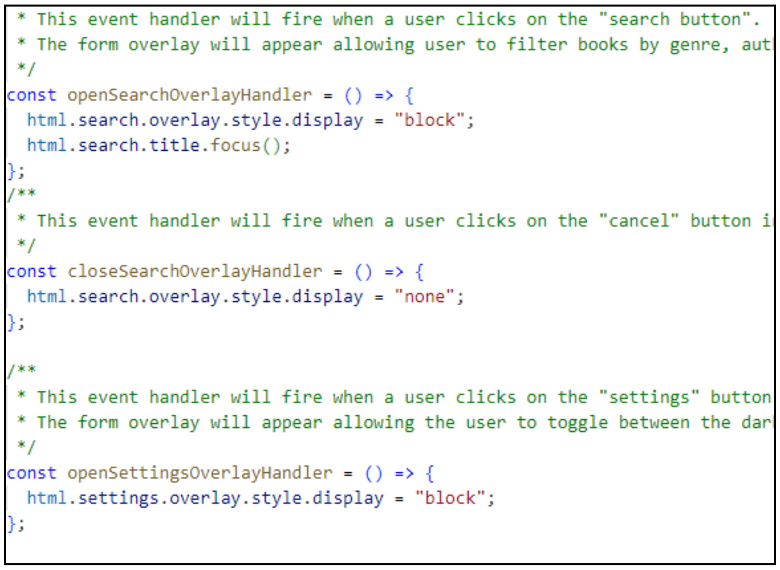
The worst abstraction from the book connect codebase was:

- Hardcoding the theme-specific styles in the event handler. It is not a good idea to do this because there will not be room to modify unless you change it from the handler itself.



```javascript
const toggleLightAndDarkModeHandler = (event) => {
  event.preventDefault();
  const formData = new FormData(event.target);
  const { theme } = Object.fromEntries(formData);

  if (theme === "night") {
    document.documentElement.style.setProperty("--color-dark", "255, 255, 255");
    document.documentElement.style.setProperty("--color-light", "10, 10, 20");
  } else {
    document.documentElement.style.setProperty("--color-dark", "10, 10, 20");
    document.documentElement.style.setProperty("--color-light", "255, 255, 255");
  }
```

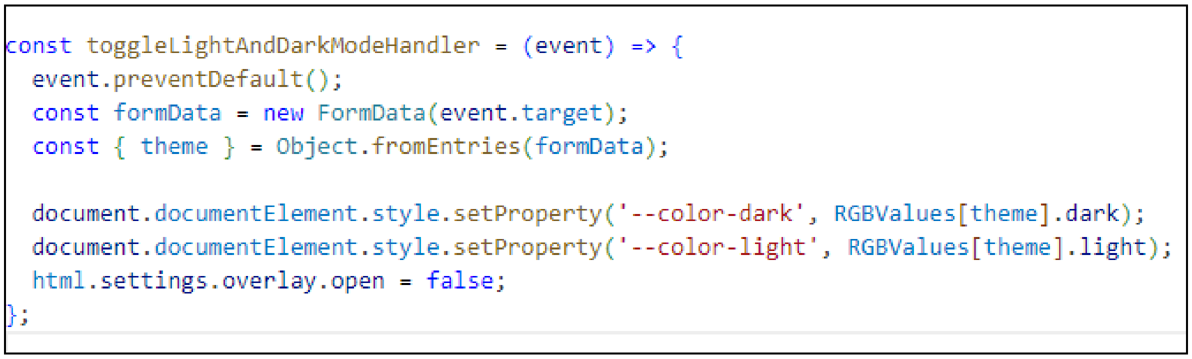
- Another example of a bad abstraction is having multiple responsibilities in one function. The filter books handler has the responsibility of filtering the books and also handling the submission of the book which violates the Single Responsibility Principle.
- The last abstraction that I would consider to be bad was the event handlers having direct html manipulation which leads to it being tightly coupled between the JavaScript Code and the HTML structure. Meaning if the HTML structure changes, it may require modification in multiple places, which makes the code harder to maintain



```javascript
 * This event handler will fire when a user clicks on the "search button".
 * The form overlay will appear allowing user to filter books by genre, aut
 */
const openSearchOverlayHandler = () => {
  html.search.overlay.style.display = "block";
  html.search.title.focus();
};
/**
 * This event handler will fire when a user clicks on the "cancel" button i
 */
const closeSearchOverlayHandler = () => {
  html.search.overlay.style.display = "none";
};

/**
 * This event handler will fire when a user clicks on the "settings" button
 * The form overlay will appear allowing the user to toggle between the dar
 */
const openSettingsOverlayHandler = () => {
  html.settings.overlay.style.display = "block";
};
```

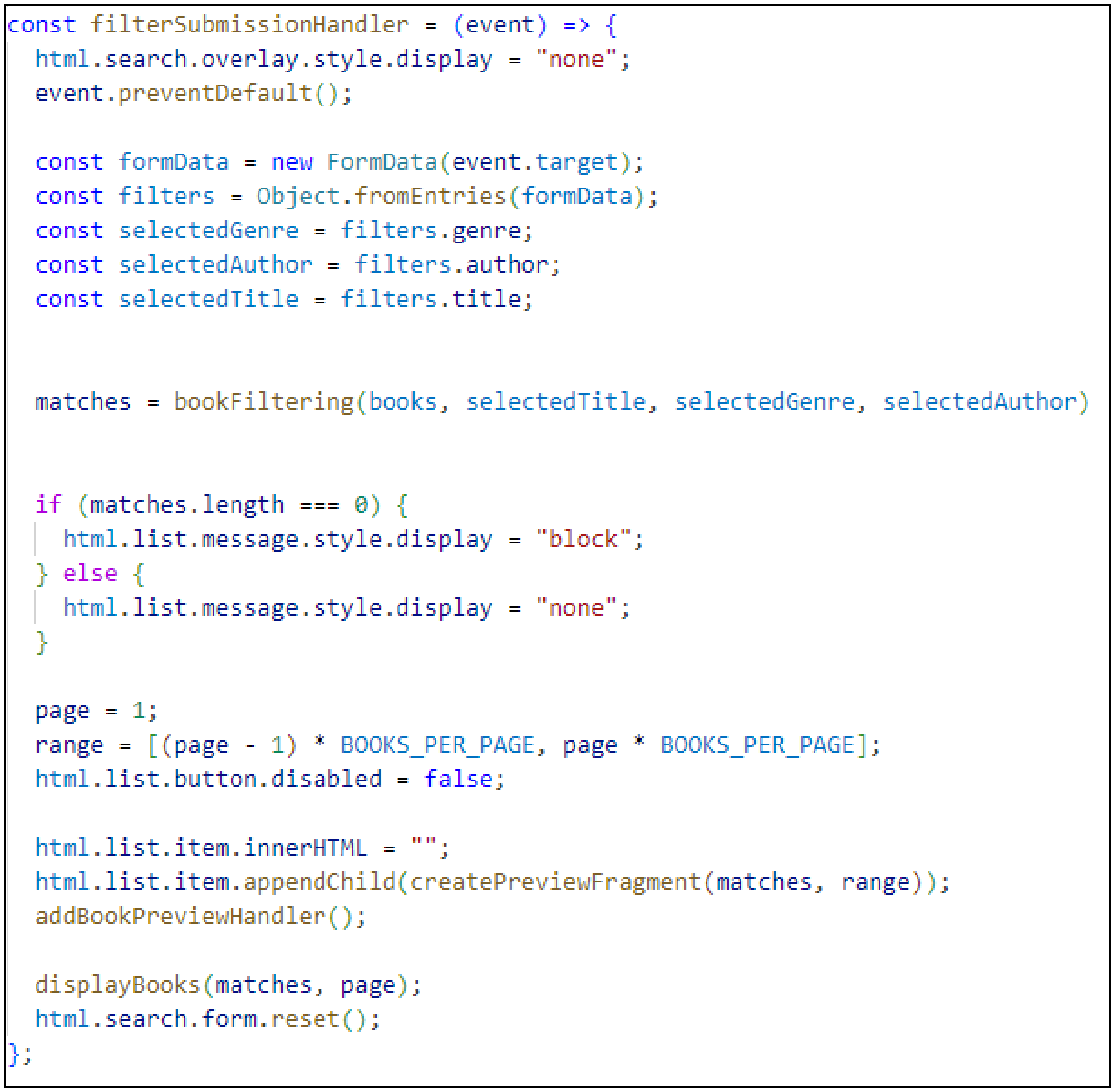
3. How can The three worst abstractions be improved via SOLID principles.

- According to the Open-Closed Principle, entities(such as functions) should be open for extension but closed for modification. So instead of having hardcoded values in the toggleDarkAndLightMode function, rather create a separate object that will store the RGB values which can be modified and updated there instead of the functions itself.



```
const toggleLightAndDarkModeHandler = (event) => {
  event.preventDefault();
  const formData = new FormData(event.target);
  const { theme } = Object.fromEntries(formData);

  document.documentElement.style.setProperty('--color-dark', RGBValues[theme].dark);
  document.documentElement.style.setProperty('--color-light', RGBValues[theme].light);
  html.settings.overlay.open = false;
};
```

- According to the Single Responsibility Principle, a function should only have one reason to change, basically a single responsibility. To improve the abstraction of the filterBook handlers, I could separate the functionality of the filtering of the books and the submission of the form. This will create a reusable and modular component that handles the specific task of filtering books based on the given filters. This also makes it easier to test and maintain the code independently from other parts of the codebase. As you see from the example below, within the filterSubmissionHandler function, a function called bookFiltering is called which takes the books Array and the selected genre, author and title and stores it in the matches variable which will then be shown in the webpage



```
const filterSubmissionHandler = (event) => {
  html.search.overlay.style.display = "none";
  event.preventDefault();

  const formData = new FormData(event.target);
  const filters = Object.fromEntries(formData);
  const selectedGenre = filters.genre;
  const selectedAuthor = filters.author;
  const selectedTitle = filters.title;


  matches = bookFiltering(books, selectedTitle, selectedGenre, selectedAuthor)


  if (matches.length === 0) {
    html.list.message.style.display = "block";
  } else {
    html.list.message.style.display = "none";
  }

  page = 1;
  range = [(page - 1) * BOOKS_PER_PAGE, page * BOOKS_PER_PAGE];
  html.list.button.disabled = false;

  html.list.item.innerHTML = "";
  html.list.item.appendChild(createPreviewFragment(matches, range));
  addBookPreviewHandler();

  displayBooks(matches, page);
  html.search.form.reset();
};
```