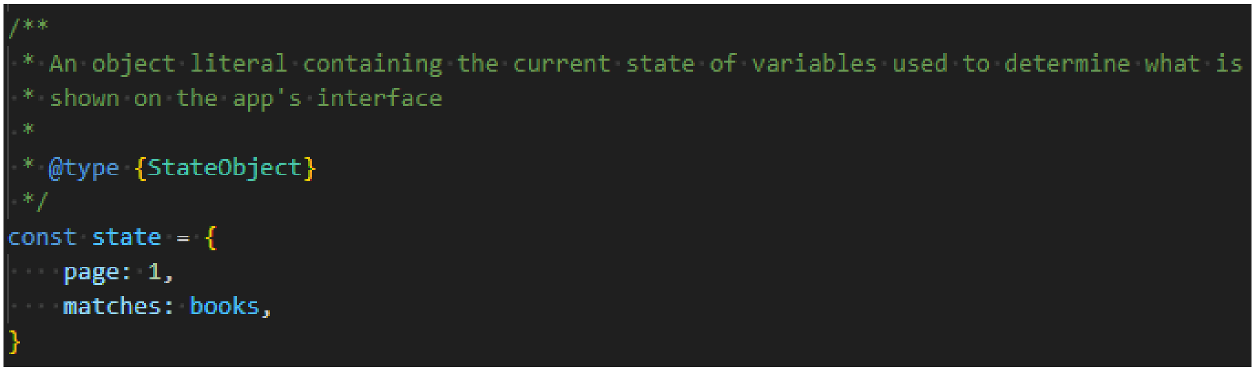
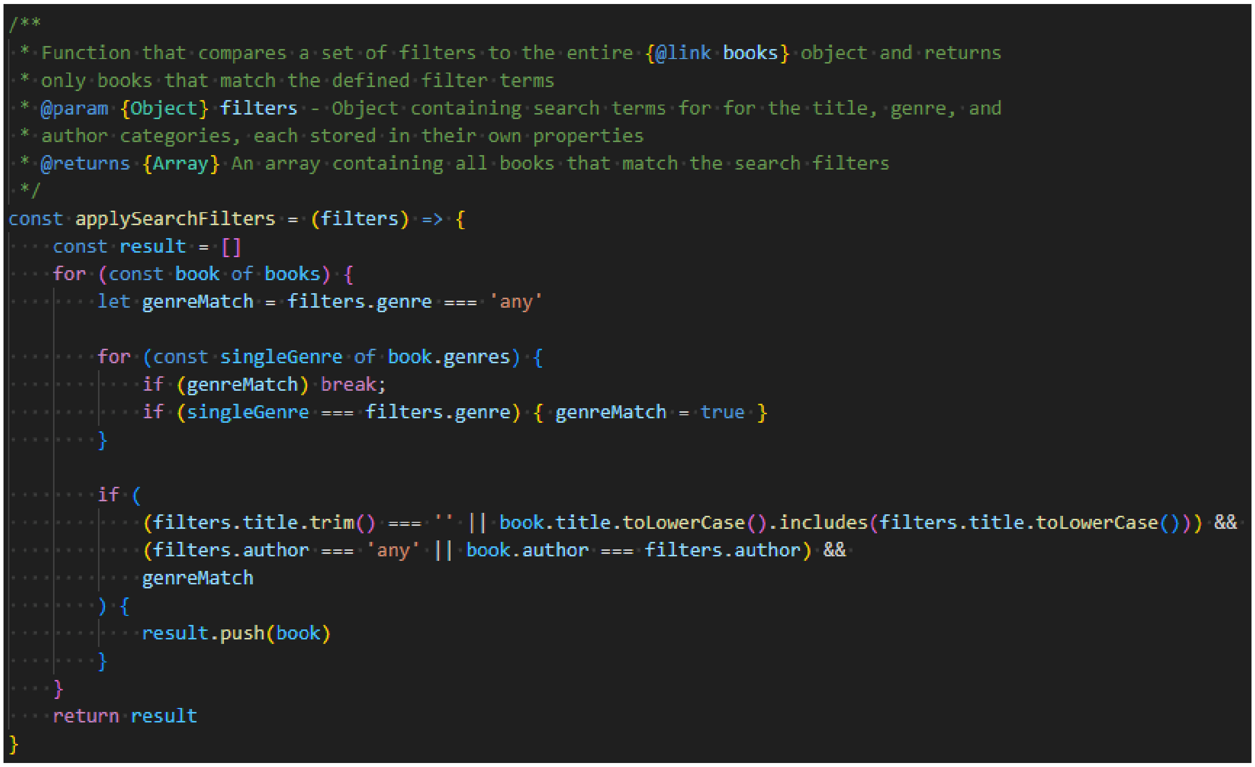
DWA_07.4 Knowledge Check_DWA7

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



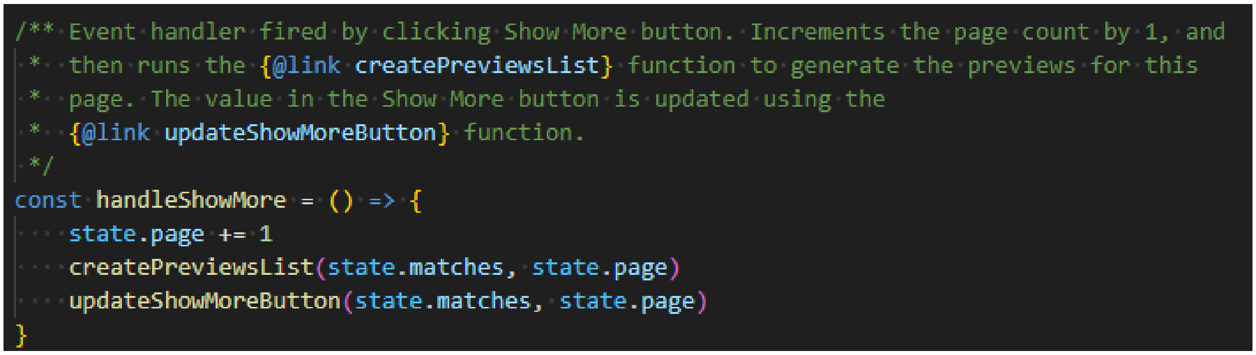
```
/**
 * An object literal containing the current state of variables used to determine what is
 * shown on the app's interface
 *
 * @type {StateObject}
 */
const state = {
    page: 1,
    matches: books,
}
```

The state object I added to keep track of changes of variables that affect what needs to be displayed at any given time is a good abstraction.It has a single purpose - to store state information of the app. It can be easily extended by adding more properties, without changing existing ones. Interfaces are focussed, as the user accesses only the property needed at any time. It also uses the dependency inversion principle, as it includes data in the form of other abstractions (e.g. books array) where possible.



```
/**
 * Function that compares a set of filters to the entire {@link books} object and returns
 * only books that match the defined filter terms
 * @param {Object} filters - Object containing search terms for for the title, genre, and
 * author categories, each stored in their own properties
 * @returns {Array} An array containing all books that match the search filters
 */
const applySearchFilters = (filters) => {
    const result = []
    for (const book of books) {
        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())) &&
            (filters.author === 'any' || book.author === filters.author) &&
            genreMatch
        ) {
            result.push(book)
        }
    }
    return result
}
```

The applySearchFilters function follows the SOLID principles fairly well, with the exception of LSP, which is hard to apply in the context of a function. It has a single well-defined purpose i.e. to return an array matching the search filters (single responsibility), and depends upon other abstractions rather than concrete data (dependency inversion), as it takes the filters object as its argument. The interface is limited to one purpose without ancillary methods. It could be open to extension, if you wanted to add more search filters. These can be added without interfering with the existing code.

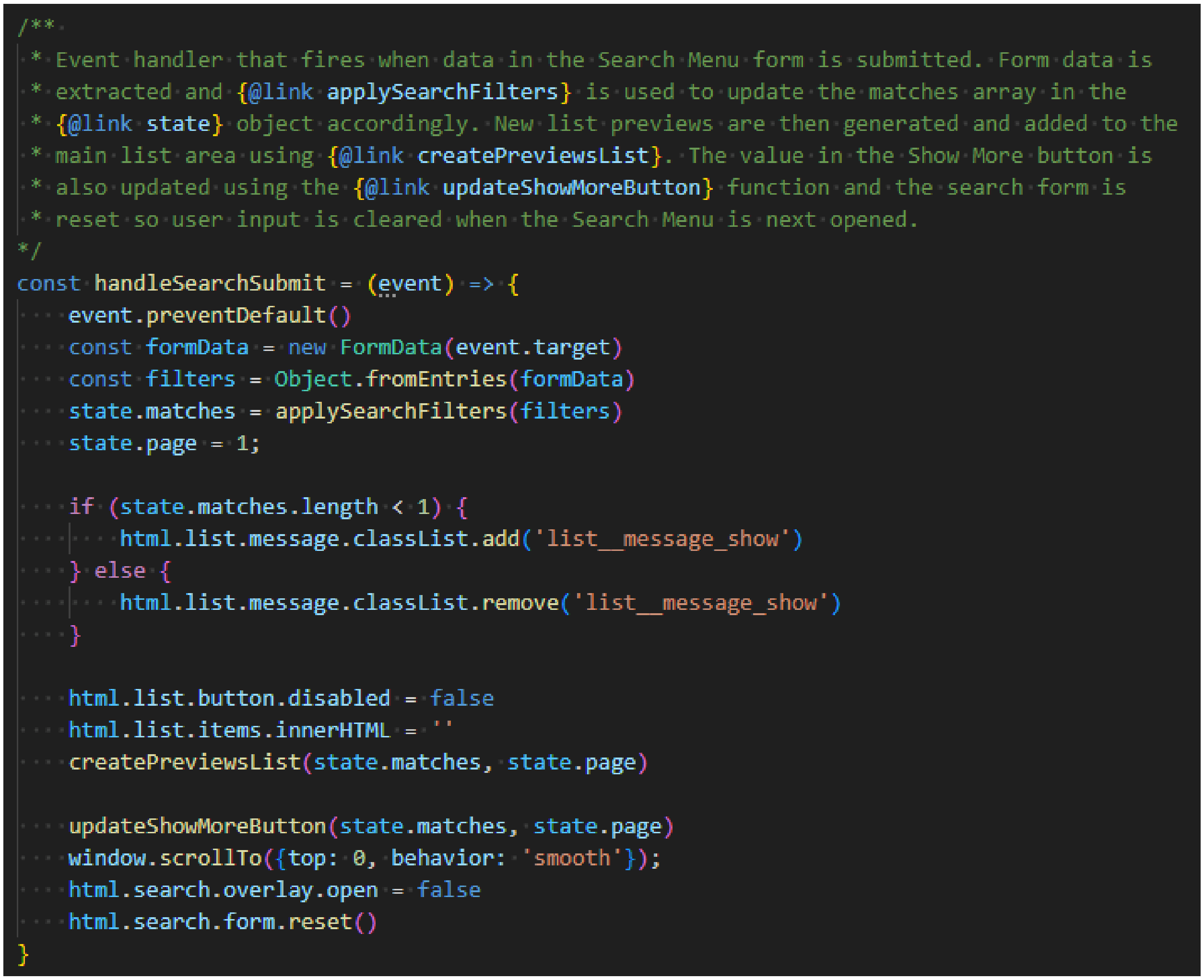


```
/** Event handler fired by clicking Show More button. Increments the page count by 1, and
 *  then runs the {@link createPreviewsList} function to generate the previews for this
 *  page. The value in the Show More button is updated using the
 *  {@link updateShowMoreButton} function.
 */
const handleShowMore = () => {
    state.page += 1
    createPreviewsList(state.matches, state.page)
    updateShowMoreButton(state.matches, state.page)
}
```

The handleShowMore() function is a very lean function with the very defined single purpose of displaying a new page of results. All of its functionality depends on other functions or objects (i.e. other abstractions), indicating good use of the dependency inversion principle. The interface is a single access point through an event listener. It is closed for modification, in that the functions it is using do not need to be changed, but it could be easily extended to perform other actions on click of the Show More button.

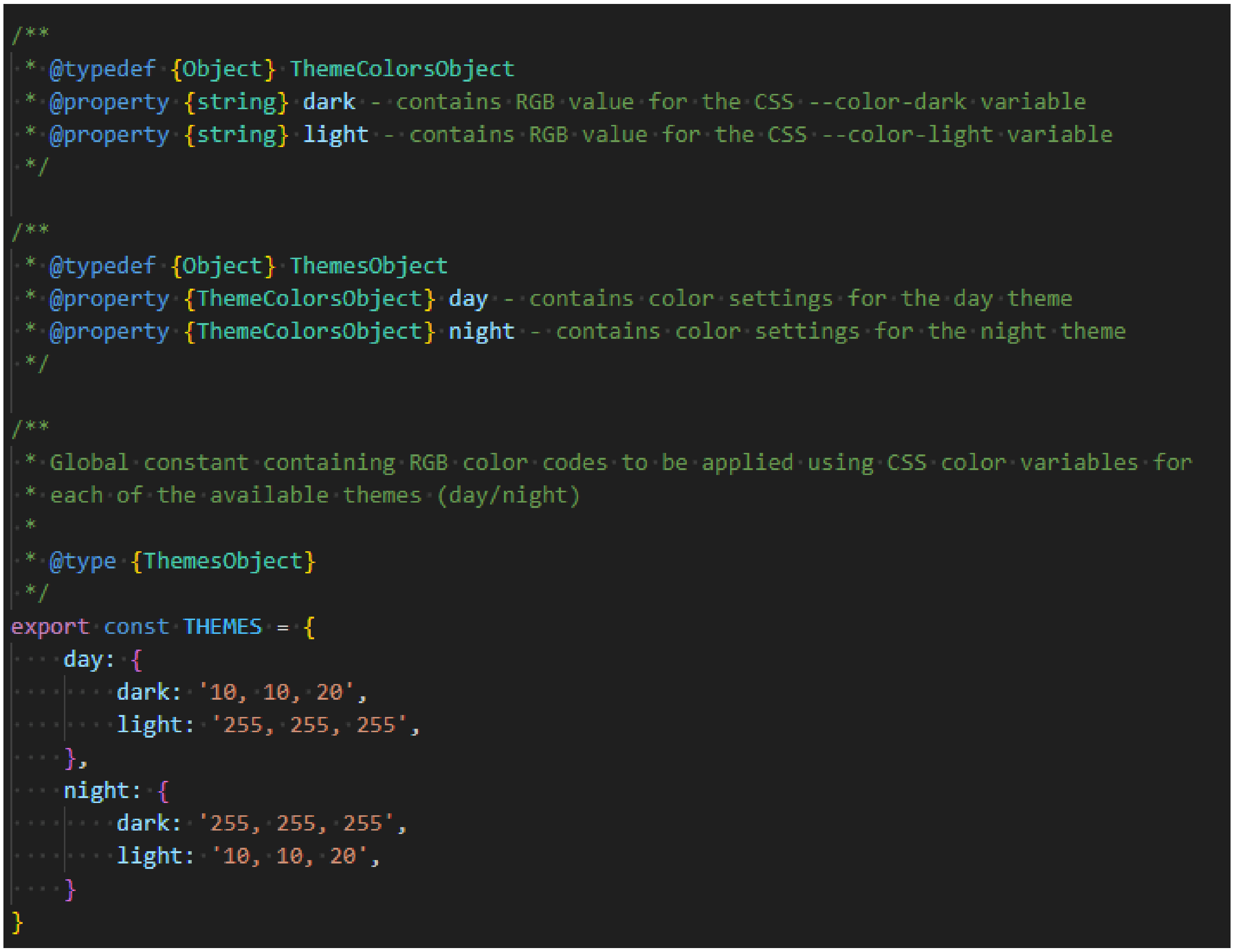
---

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



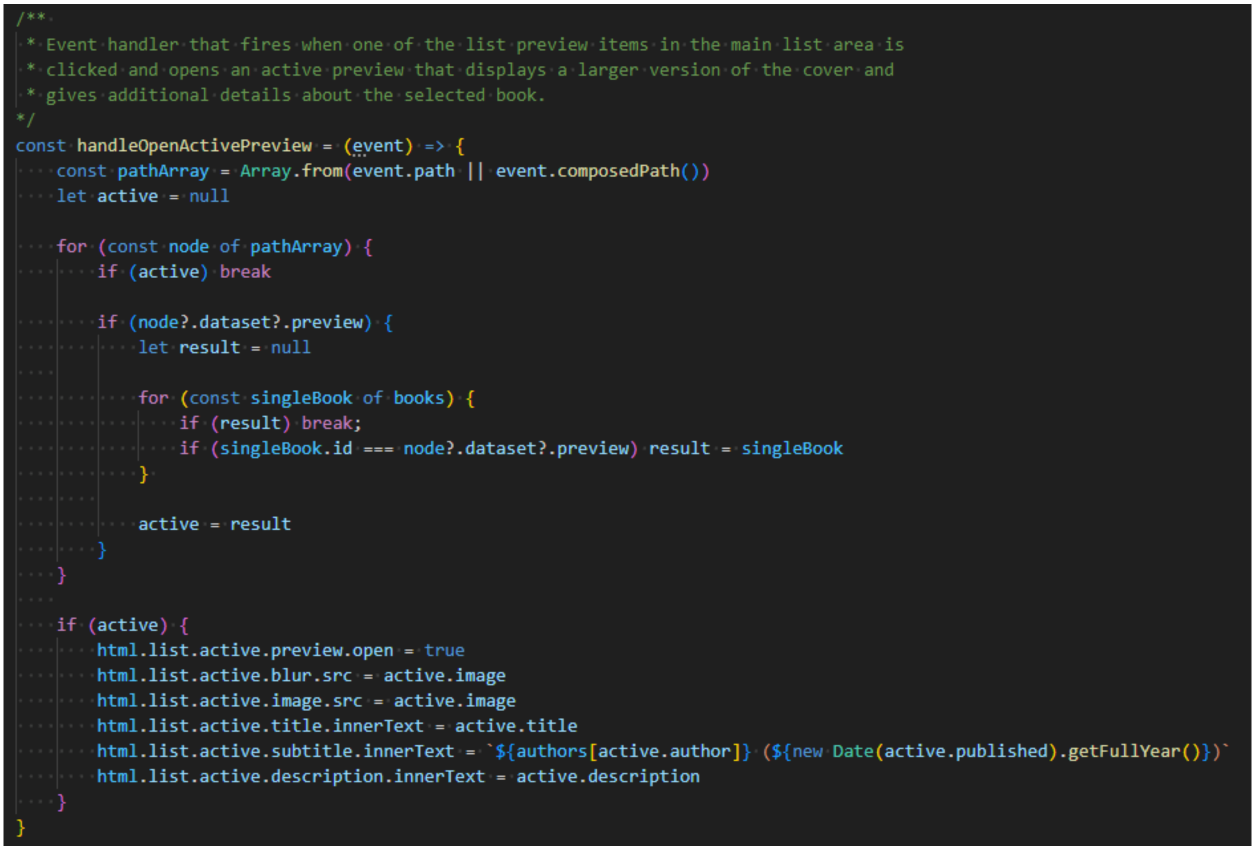
```
/**
 * Event handler that fires when data in the Search Menu form is submitted. Form data is
 * extracted and {@link applySearchFilters} is used to update the matches array in the
 * {@link state} object accordingly. New list previews are then generated and added to the
 * main list area using {@link createPreviewsList}. The value in the Show More button is
 * also updated using the {@link updateShowMoreButton} function and the search form is
 * reset so user input is cleared when the Search Menu is next opened.
*/
const handleSearchSubmit = (event) => {
    event.preventDefault()
    const formData = new FormData(event.target)
    const filters = Object.fromEntries(formData)
    state.matches = applySearchFilters(filters)
    state.page = 1;

    if (state.matches.length < 1) {
        html.list.message.classList.add('list__message_show')
    } else {
        html.list.message.classList.remove('list__message_show')
    }

    html.list.button.disabled = false
    html.list.items.innerHTML = ''
    createPreviewsList(state.matches, state.page)

    updateShowMoreButton(state.matches, state.page)
    window.scrollTo({top: 0, behavior: 'smooth'});
    html.search.overlay.open = false
    html.search.form.reset()
}
```

The handleSearchSubmit function does not completely conform to the single responsibility principle, as it has multiple behaviors that need to be triggered when the search form is submitted. Other principles are generally met (except LSP).



```javascript
/**
 * @typedef {Object} ThemeColorsObject
 * @property {string} dark - contains RGB value for the CSS --color-dark variable
 * @property {string} light - contains RGB value for the CSS --color-light variable
 */

/**
 * @typedef {Object} ThemesObject
 * @property {ThemeColorsObject} day - contains color settings for the day theme
 * @property {ThemeColorsObject} night - contains color settings for the night theme
 */

/**
 * Global constant containing RGB color codes to be applied using CSS color variables for
 * each of the available themes (day/night)
 *
 * @type {ThemesObject}
 */
export const THEMES = {
    day: {
        dark: '10, 10, 20',
        light: '255, 255, 255',
    },
    night: {
        dark: '255, 255, 255',
        light: '10, 10, 20',
    }
}
```

The themes object has limited extendability due to the structuring of the type definition used - does not follow Open-Closed principle. It is limited to the structure of having only two color variables, which might not be the case in more complex color themes than just dark and light mode. Other principles are generally met (except LSP).



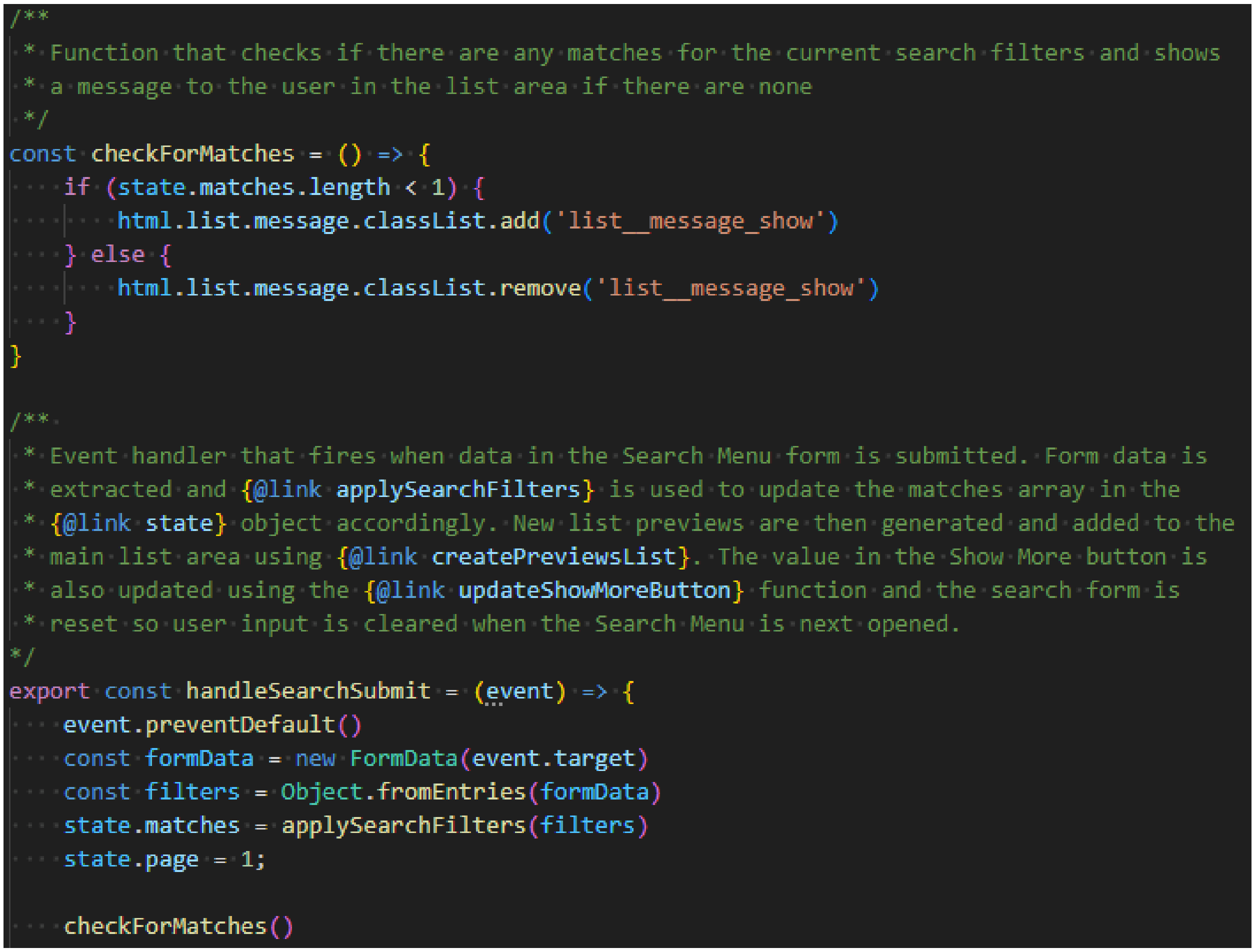
```
/**
 * Event handler that fires when one of the list preview items in the main list area is
 * clicked and opens an active preview that displays a larger version of the cover and
 * gives additional details about the selected book.
*/
const handleOpenActivePreview = (event) => {
    const pathArray = Array.from(event.path || event.composedPath())
    let active = null

    for (const node of pathArray) {
        if (active) break

        if (node?.dataset?.preview) {
            let result = null

            for (const singleBook of books) {
                if (result) break;
                if (singleBook.id === node?.dataset?.preview) result = singleBook
            }

            active = result
        }
    }

    if (active) {
        html.list.active.preview.open = true
        html.list.active.blur.src = active.image
        html.list.active.image.src = active.image
        html.list.active.title.innerText = active.title
        html.list.active.subtitle.innerText = `${authors[active.author]} (${new Date(active.published).getFullYear()})`
        html.list.active.description.innerText = active.description
    }
}
```

The handleOpenActivePreview triggers two separate behaviors in response to its event listener which is against the Single Responsibility Principle. Other principles are generally met (except LSP).

---

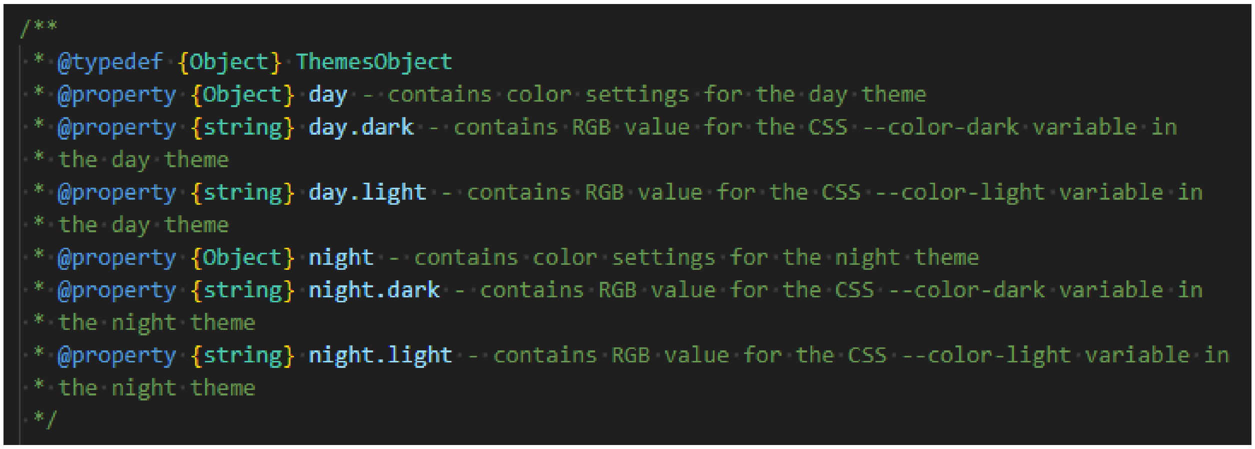
3. How can The three worst abstractions be improved via SOLID principles.

The handleSearchSubmit function could be improved by extracting the ancillary behavior of checking whether there are no matches for search results into its own function.



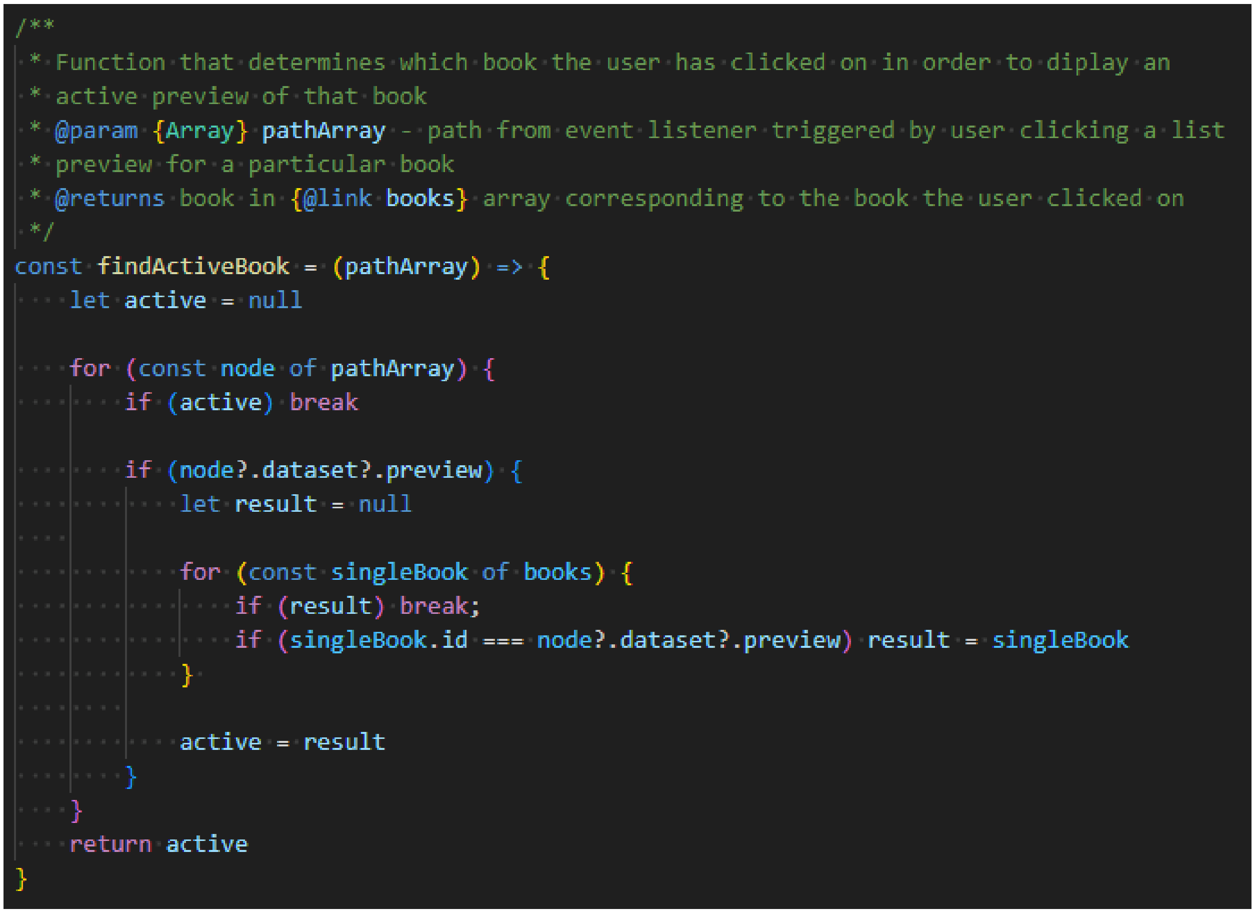
```
/**
 * Function that checks if there are any matches for the current search filters and shows
 * a message to the user in the list area if there are none
 */
const checkForMatches = () => {
    if (state.matches.length < 1) {
        html.list.message.classList.add('list__message_show')
    } else {
        html.list.message.classList.remove('list__message_show')
    }
}

/**
 * Event handler that fires when data in the Search Menu form is submitted. Form data is
 * extracted and {@link applySearchFilters} is used to update the matches array in the
 * {@link state} object accordingly. New list previews are then generated and added to the
 * main list area using {@link createPreviewsList}. The value in the Show More button is
 * also updated using the {@link updateShowMoreButton} function and the search form is
 * reset so user input is cleared when the Search Menu is next opened.
*/
export const handleSearchSubmit = (event) => {
    event.preventDefault()
    const formData = new FormData(event.target)
    const filters = Object.fromEntries(formData)
    state.matches = applySearchFilters(filters)
    state.page = 1;

    checkForMatches()
```

The themes object could be fixed by simply updating the ThemesObject type definition to be more broad, so it doesn't prevent themes with different color variable structures from causing an issue because they don't match the rigid structure.

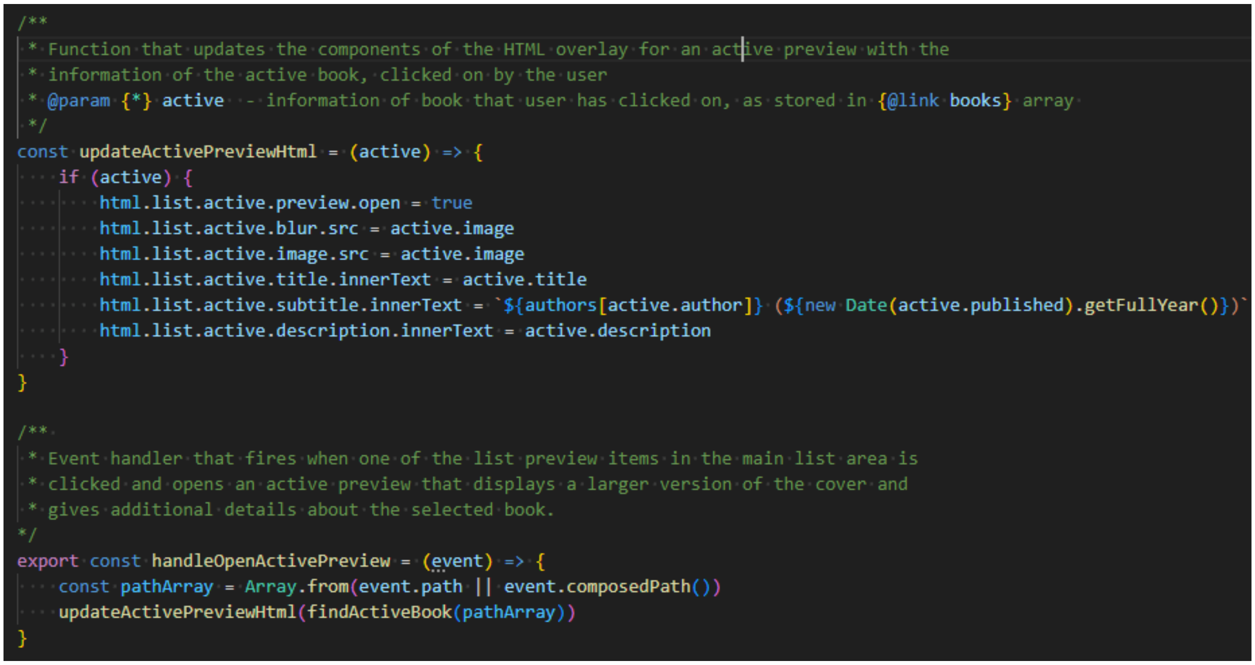


```
/**
 * @typedef {Object} ThemesObject
 * @property {Object} day - contains color settings for the day theme
 * @property {string} day.dark - contains RGB value for the CSS --color-dark variable in
 * the day theme
 * @property {string} day.light - contains RGB value for the CSS --color-light variable in
 * the day theme
 * @property {Object} night - contains color settings for the night theme
 * @property {string} night.dark - contains RGB value for the CSS --color-dark variable in
 * the night theme
 * @property {string} night.light - contains RGB value for the CSS --color-light variable in
 * the night theme
 */
```

The handleOpenActivePreview function could be improved by splitting it into two distinct functions, one that determines the active book and one that uses that information to update the HTML according to that book’s information.



```
/**
 * Function that determines which book the user has clicked on in order to diplay an
 * active preview of that book
 * @param {Array} pathArray - path from event listener triggered by user clicking a list
 * preview for a particular book
 * @returns book in {@link books} array corresponding to the book the user clicked on
 */
const findActiveBook = (pathArray) => {
    let active = null

    for (const node of pathArray) {
        if (active) break

        if (node?.dataset?.preview) {
            let result = null

            for (const singleBook of books) {
                if (result) break;
                if (singleBook.id === node?.dataset?.preview) result = singleBook
            }

            active = result
        }
    }
    return active
}
```



```
/**
 * Function that updates the components of the HTML overlay for an active preview with the
 * information of the active book, clicked on by the user
 * @param {*} active  - information of book that user has clicked on, as stored in {@link books} array
 */
const updateActivePreviewHtml = (active) => {
    if (active) {
        html.list.active.preview.open = true
        html.list.active.blur.src = active.image
        html.list.active.image.src = active.image
        html.list.active.title.innerText = active.title
        html.list.active.subtitle.innerText = `${authors[active.author]} (${new Date(active.published).getFullYear()})`
        html.list.active.description.innerText = active.description
    }
}

/**
 * Event handler that fires when one of the list preview items in the main list area is
 * clicked and opens an active preview that displays a larger version of the cover and
 * gives additional details about the selected book.
*/
export const handleOpenActivePreview = (event) => {
    const pathArray = Array.from(event.path || event.composedPath())
    updateActivePreviewHtml(findActiveBook(pathArray))
}
```

In addition - overall structure of modules could also be improved by moving the functions relating to each of the features (like Search menu, Settings Menu etc.) into their own files.
