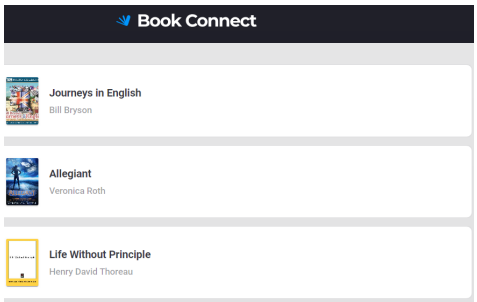
DWA_07.4 Knowledge Check_DWA7

---

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

Abstractions make things easier to understand and work with. They simplify complex tasks, help us be more efficient, and allow us to break things into smaller, manageable pieces. In software development and other areas, abstractions make it easier to organize our work, improve productivity, and build things in a modular way, where each part has a clear and specific purpose. Below, I placed the output as well as the code, so that the reader may have a visual display of what I mean, in case I am not able to explain myself well :)

1. Abstraction of Book Display:
    - The code makes it easier to show a book by putting its image, title, and author inside a button.
    - This abstraction makes it simple to change and update how the book looks on the screen, treating it as one complete piece, which is easy to understand and work with.




```
import { books, authors, genres, BOOKS_PER_PAGE } from './data.js'

let page = 1;
let matches = books

const starting = document.createDocumentFragment()

for (const { author, id, image, title } of matches.slice(0, BOOKS_PER_PAGE)) {
    const element = document.createElement('button')
    element.classList.add('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>
    `

    starting.appendChild(element)
}
```

2. Abstraction of Filtering:
    - Used the data-search-form event listener by grouping them together, so that I would be able to simplify the filtering process.
    - This abstraction helps organize and modularize the code by keeping the filtering process separate from other components, leading to improved code organization and modularity.

```
document.querySelector('[data-search-form]').addEventListener('submit', (event) => {
    event.preventDefault()
    const formData = new FormData(event.target)
    const filters = Object.fromEntries(formData)
    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)
        }
    }
```

3. Abstraction of UI (User Interface) Manipulation:
    - The code simplifies changing elements on the screen by automatically updating them when the user does something or when the data changes.
    - This abstraction makes it easier to handle the UI's current interface and reduces the need to manually manipulate the underlying HTML elements, resulting in code that is easier to maintain and understand.

```
const genreHtml = document.createDocumentFragment()
const firstGenreElement = document.createElement('option')
firstGenreElement.value = 'any'
firstGenreElement.innerText = 'All Genres'
genreHtml.appendChild(firstGenreElement)
```

---

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

1. Lack of Modularity in UI Manipulation:
    - The code doesn't organize the UI update logic into separate reusable functions or modules, making it harder to maintain, understand, and modify. It can result in duplicated code and difficulties in managing UI manipulation effectively.
2. Inline HTML Creation:
    - The code directly combines HTML markup within JavaScript code, making it more challenging to read, understand, and manage, particularly for complex HTML structures.
3. Limited Error Handling:
    - The code doesn't handle errors or validate certain operations, like accessing elements using querySelector. This can result in unexpected errors during runtime and make it difficult to identify and fix issues.

---

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

1. Lack of Modularity in UI Manipulation:
    - Apply the Single Responsibility Principle (SRP): Split the UI manipulation code into separate modules or functions, each responsible for a specific aspect of UI manipulation.
    - Create reusable functions or classes that encapsulate related UI manipulation logic, such as updating button states, handling theme changes, or managing overlays.
    - This separation of concerns will enhance modularity, maintainability, and readability of the codebase.
2. Inline HTML Creation:
    - Apply the Open/Closed Principle (OCP): Separate the HTML creation logic from the JavaScript code by adopting a templating or component-based approach.
    - Utilize a front-end framework or library that supports declarative UI rendering, such as React, Vue.js, or Angular.
    - Define reusable components or templates that represent the structure and behavior of UI elements, making the code more maintainable and readable.
3. Limited Error Handling:
    - Apply the Liskov Substitution Principle (LSP): Improve error handling by using appropriate abstractions and leveraging type checking or contracts.
    - Utilize TypeScript or other statically typed languages to catch potential errors at compile-time.
    - Implement proper error handling mechanisms, such as try-catch blocks, to gracefully handle runtime exceptions and provide meaningful feedback to users.
