DWA_07.4 Knowledge Check_DWA7

---

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

- Storing all the DOM element references (querySelectors) into a single object.

Why? The code uses the object to abstract away the process of accessing and manipulating DOM elements. By using the ‘selectors’ object, the code can easily access and manipulate the necessary DOM elements without directly interacting with the DOM API every time.

- Function Abstraction

Why? The code defines several functions that encapsulate specific tasks. These functions provide a higher-level abstraction by encapsulating specific operations and making the code more modular and reusable.

- filterBooks(filters)

Why? This function filters the books array based on the provided filters. It applies the filtering logic defined in the function and returns the filtered array. Abstracting the filtering logic into a separate function enhances code organization, readability, and reusability. It separates the concern of filtering books from other parts of the code, making it easier to understand and modify the filtering criteria in the future. It also promotes the principle of single responsibility by isolating the filtering functionality.

---

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

- `handleListButtonClicked()`:

why? This function handles the logic for the "Show More" button click event. It retrieves the next set of books to display and appends them to the list. While this function serves its purpose, it violates the single responsibility principle. It combines the task of retrieving the next set of books, appending them to the list, and updating the page variable. Splitting this function into smaller, more focused functions would improve code readability and maintainability.

- Inline event handlers:

In the code, there are several inline event handlers assigned directly to the DOM elements, such as `selectors.searchCancel.addEventListener('click', ...)`, `selectors.headerSearch.addEventListener('click', ...)`, and `selectors.listButton.addEventListener('click', ...)`. Inline event handlers can lead to less maintainable code, especially when multiple elements require the same event handling logic. It is considered a poor abstraction because it mixes HTML markup with JavaScript logic, making it harder to manage and modify the code in the long run. Utilizing event delegation or assigning event listeners programmatically would be a better approach.

- Excessive use of global variables:

The code utilizes multiple global variables, such as `page`, `matches`, and `selectors`. While using global variables can provide easy access to data and DOM elements, it can lead to potential issues, such as naming conflicts and reduced code modularity. It is generally considered a better practice to encapsulate related data and functions within objects or modules to avoid polluting the global namespace. Encapsulating these variables within a higher-level object or module would improve code organization and reduce the potential for naming conflicts.

---

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

To improve the three mentioned abstractions based on the SOLID principles, the following approaches can be considered:

- `handleListButtonClicked()`:

This function violates the Single Responsibility Principle (SRP) by combining multiple tasks. To improve it, we can split it into smaller functions, each responsible for a specific task. By separating these responsibilities, each function can have a clear purpose and adhere to the SRP. Additionally, we can apply the Dependency Inversion Principle (DIP) by abstracting dependencies (e.g., the `page` variable) behind interfaces or function arguments, making the code more modular and flexible. Inline event handlers: Instead of using inline event handlers, we can adhere to the Open-Closed Principle (OCP) by implementing event delegation or programmatically assigning event listeners. - Event delegation: Instead of assigning event listeners to individual elements, we can assign a single event listener to a common parent element (e.g., `document` or a specific container element).

- Excessive use of global variables:

To address the issue of excessive global variables and promote better code organization and encapsulation, we can apply the Dependency Inversion Principle (DIP) and encapsulate related data and functions within objects or modules. - Create an object or module to encapsulate related variables (e.g., `page` and `matches`). This object can provide methods to interact with and modify these variables, ensuring encapsulation and controlled access. - Apply the Dependency Injection (DI) principle to inject this object or module as a dependency where needed, instead of relying on global variables directly. By using DI, we can provide better control over dependencies and improve testability and modularity.