DWA_07.4 Knowledge Check_DWA7

---

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

-Modules:
Modules facilitate code reuse by allowing you to export and import functionality from one module to another. Instead of duplicating code or reinventing the wheel, you can leverage existing modules and libraries to solve common problems.
The code is organized into modules using the import statement. The BookList module is referenced at the beginning of the code.

-Data Access:
Once you have a well-defined data access layer, you can reuse it across multiple parts of your application or even across different projects. This saves development time and effort, as you don't have to rewrite the same data access logic for every component or project.
The books, authors, genres, and BOOKS_PER_PAGE values are imported from a separate data.js file.

-DOM Manipulation:
Various DOM manipulation functions and event listeners are used to interact with the HTML elements on the page. They allow you to separate your application logic from the DOM manipulation code, making it easier to reason about and test different parts of your application independently.

---

2. Which were the three worst abstractions, and why?
-functions
Lack of clarity: Poorly named or poorly defined functions can make code harder to understand. It's essential to choose meaningful names for functions and ensure they have clear and concise responsibilities.

-Variables
Debugging difficulties: Debugging code that heavily relies on variables as abstractions can be challenging. Since variables can have different values at different points in the

code, it becomes harder to trace the flow and understand the state of the program during runtime.

-Constants
Lack of flexibility: Constants, by definition, have fixed values that cannot be changed at runtime. This lack of flexibility can make it challenging to adapt the code to different scenarios or dynamically adjust behavior based on runtime conditions. If the abstraction needs to be parameterized or modified based on runtime inputs, constants may not be the best choice.

---

3. How can The three worst abstractions be improved via SOLID principles.
- Functions:
Single Responsibility Principle (SRP): Ensure that each function has a single, well-defined responsibility. Split large functions into smaller, more focused functions that perform specific tasks.
Open/Closed Principle (OCP): Design functions in a way that allows for easy extension without modifying existing code. Use parameters to pass in behavior or strategies to customize the function's behavior.
Liskov Substitution Principle (LSP): Ensure that any function overriding or implementing an interface follows the same contract and can be used interchangeably with the base function.
Interface Segregation Principle (ISP): Avoid creating functions with a large number of parameters or dependencies. Consider breaking them down into smaller, more specialized functions.
Dependency Inversion Principle (DIP): Encourage functions to depend on abstractions (interfaces or abstract classes) rather than concrete implementations. This promotes flexibility and testability.

-Variables:
Single Responsibility: Ensure that variables have a clear purpose and are used for a single task. Avoid using variables for multiple purposes, as it can lead to confusion and errors.

Good Naming Conventions: Use meaningful and descriptive names for variables to enhance readability and understanding.
Scope Limitation: Limit the scope of variables to the smallest possible context where they are needed. This helps avoid unintended side effects and reduces the chances of name collisions.
Encapsulation: Encapsulate variables within appropriate classes or modules to control access and enforce data integrity. This promotes better organization and maintainability.

Constants:
While SOLID principles are not typically applied directly to constants, you can follow some general practices for using constants effectively:
Naming Conventions: Use descriptive names for constants to convey their purpose and make the code more readable.
Grouping: Organize related constants into logical groups or enums to improve code organization and maintainability.
Encapsulation: Consider encapsulating constants within appropriate classes or modules to limit their visibility and prevent unintended modification.
