DWA_07.4 Knowledge Check_DWA7

---

1. Which were the three best abstractions, and why?
    - Creating an object called data which references all the html elements. So elements do not have to be query selected each time they are needed. If you need to modify an element, you only need to modify it in the data object instead of combing through the code to see each time it is referenced. (SRP - the object has a single responsibility)
    - Creating objects containing the day and night theme values. If you want to add or change themes then you only need to change the values in the object rather than every time the theme is referenced in the code.

---

2. Which were the three worst abstractions, and why?
    - (Good abstraction but only in this question because i didn't use it originally) Using modules -separating relating functions into different script files and using imports and exports where necessary. If there are multiple functions relating to the same aspect of the app then separating them into different clearly named files can help with maintainability. You don't need to comb through the entire codebase to find a specific function. (SRP - each file has a single responsibility, DIP, RSP- small and focused on specific functionality)

---

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

SOLID principles

- **Single Responsibility Principle (SRP):** This principle states that a class or module should have only one reason to change. To apply SRP in JavaScript, you can create separate modules or classes that handle specific tasks or responsibilities. Each module or class should be responsible for a single functionality or feature, and should not be coupled with other functionalities.
- **Open-Closed Principle (OCP):** This principle states that software entities (classes, modules, functions, etc.) should be open for extension but closed for modification. In JavaScript, you can apply this principle by creating abstract classes or interfaces that define a set of methods or properties. Concrete classes can then inherit from these abstract classes or interfaces and implement their methods. This way, you can add new functionalities without modifying existing code. -***build on top of something but not rewrite it***
- **Liskov Substitution Principle (LSP):** This principle states that objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program. In JavaScript, you can implement LSP by ensuring that subclasses inherit all the properties and methods of their superclass, and do not alter their behaviour in unexpected ways.
- **Interface Segregation Principle (ISP):** This principle states that a client should not be forced to depend on methods it does not use. In JavaScript, you can apply ISP by creating small, focused interfaces that define only the methods that a client needs. This way, you can avoid creating large, bloated interfaces that are difficult to maintain and understand.
- **Dependency Inversion Principle (DIP):** This principle states that high-level modules should not depend on low-level modules. Instead, both should depend on abstractions. In JavaScript, you can implement DIP by using dependency injection, which allows you to pass dependencies to a module or class rather than creating them inside. This way, you can decouple modules and make them more reusable and testable.