DWA_07.4 Knowledge Check_DWA7

---

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

Function Abstraction:

- This is one of the most fundamental forms of abstraction in JavaScript. You create functions to encapsulate a set of instructions or actions and give them a name, which can be called whenever needed. Functions abstract away the details of how a task is accomplished, making your code more modular and easier to understand.

Object Abstraction:

- Objects are a way to encapsulate related data and behavior. You can create objects to represent entities in your application, grouping properties (data) and methods (functions) that operate on that data. This abstraction helps in modeling real-world entities in your code.

Class Abstraction (ES6 and later):

- Classes are a way to define object blueprints and create instances of those objects. They provide a higher level of abstraction for creating objects with similar properties and methods. Classes help in implementing object-oriented programming concepts in JavaScript.

---

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

1. **Overuse of Inheritance:**
    - Excessive use of class hierarchies and inheritance can create tightly coupled code that is challenging to change and extend. Inheritance should be used judiciously, and composition or other techniques might be more appropriate.
    - **Over-Abstraction:**
        i. Creating too many layers of abstraction without a clear purpose can lead to unnecessary complexity. When developers over-abstract, it becomes difficult to trace how data and control flow through the code, making it harder to maintain and debug.
    - **Excessive Indirection:**
        i. Excessive use of indirection, like wrapping simple operations in multiple layers of abstraction, can make the code slow, difficult to understand, and harder to maintain. While some level of indirection can be useful, too much can hinder performance and clarity.

---

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

4. **Overuse of Inheritance:**
   Solution with SOLID: The Dependency Inversion Principle (DIP) encourages you to depend on abstractions, not concretions. Instead of relying heavily on inheritance, use interfaces and abstractions to define the structure and behavior of your classes. This promotes composition over inheritance, reducing the risk of overusing class hierarchies.

5. **Over-Abstraction:**
   Solution with SOLID: The Single Responsibility Principle (SRP) suggests that a class should have only one reason to change. Over-abstraction often occurs when a class tries to do too much. By adhering to the SRP, you can break down large, over-abstracted classes into smaller, focused classes, each responsible for a specific task. This reduces the need for excessive abstraction.

4. **Excessive Indirection:**
   Solution with SOLID: The Liskov Substitution Principle (LSP) emphasizes that subtypes should be substitutable for their base types. When using inheritance and polymorphism, ensure that subclasses do not introduce excessive indirection or radically change the behavior of their base classes. This keeps the code more predictable and easier to follow.
