DWA_07.4 Knowledge Check_DWA7

---

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

1. **Functions**, because functions allow code reuse and help in organizing logic.
2. **Objects**, because it is a way to represent and organize data in JavaScript. It also allows me to group related properties and methods together.
3. **Modules**, because it enables me to organize code into separate files or modules, making it easier to manage large codebases. It allows me to selectively export or import functionality from one module to another.

---

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

1. **Variable Hoisting**, because JavaScript hoists variable declarations to the top of their scope, which can lead to unexpected behaviour if developers are not aware of this. It can make code harder to reason about and can result in bugs and errors when variables are not declared where they are expected.
2. **Callbacks**, the extensive use of callbacks can lead to what is known as “callback hell” or “pyramid of doom”. This occurs when you have multiple nested callbacks, making the code difficult to read, understand, and maintain.
3. **Implicit type coercion**, because JavaScript has loose and dynamic typing, which allows implicit type coercion. While this can be convenient in some cases, it can also lead to unexpected results and bugs when different types are coerced in ways that developers did not intend.

---

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

1. **Variable Hoisting**:
    - **Single Responsibility Principle** (SRP): Ensure that variables are declared and initialized in the appropriate scope and location, adhering to the principle of having a single responsibility. This helps in maintaining code clarity and prevents unexpected behaviour caused by variable hoisting.
    - **Dependency Inversion Principle** (DIP): Encourage explicit dependencies and avoid relying on variables that are hoisted. By explicitly declaring variables where they are needed, you make code more predictable and easier to reason about.
2. **Callbacks**:
    - **Single Responsibility Principle** (SRP): Keep functions and callbacks focused on a single responsibility. Break down complex callbacks into smaller, more manageable functions with clear purposes. This helps to reduce the nesting and improve code readability.
    - **Open/Closed Principle** (OCP): Consider using higher-order functions or function composition to abstract away complex callback logic. By decoupling the callback functions from the specific implementation, you can make the code more modular and open for extension without modifying existing code.
3. **Implicit Type Coercion:**
    - **Liskov Substitution Principle** (LSP): Avoid relying on implicit type coercion, as it can lead to unexpected behaviour and breaks the principle of substitutability. Instead, use explicit type conversions or checks to ensure that code is more robust and predictable.
    - **Interface Segregation Principle** (ISP): When working with multiple types or objects, clearly define and adhere to the expected interfaces or contracts to avoid unintended type coercions. This helps to make it more explicit and reduces the chances of unexpected type conversions.
