# DWA\_07.4 Knowledge Check\_DWA7

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

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

1. Functions: Functions in JavaScript allow you to encapsulate a block of code that can be reused and called multiple times.Which is a easy consept for me to under stand.
2. Objects: Objects are fundamental abstractions in JavaScript that allow you to group related data and functions together.This helps me compile information at an easier level.
3. Modules: Modules provide a way to encapsulate (Sammarize) related code into separate files.Because it is allowing for better code organization, reusability, and maintainability. Modules help prevent global namespace pollution by keeping variables and functions scoped within their respective modules. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

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

1. Classes: JavaScript introduced class syntax in ECMAScript 2015 (ES6), providing a way to define objects using a class-based approach.This is still a topic i have to still have to understand.

2. Callbacks: are functions that are passed as arguments to other functions and are invoked at a later point in time.The idea that i have go back to code and remember it for future reference is confusing.

3.Prototype-based Inheritance: JavaScript uses a prototype-based inheritance model, where objects can inherit properties and methods from other objects. Its not as challenging for developers coming from class-based inheritance backgrounds.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

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

The Interface Segregation Principle (ISP) can be applied to improve this abstraction. By favoring composition over inheritance and using interfaces or contracts, you can create clear boundaries between objects and minimize the complexity introduced by prototypal inheritance.

Monolithic or God components refer to large, complex components that handle multiple responsibilities, contain numerous methods, and have high coupling with other parts of the codebase. They violate the Single Responsibility Principle (SRP) and can be difficult to understand, maintain, and test.

The Dependency Inversion Principle (DIP) can be applied to improve this abstraction. Instead of tightly coupling functions to specific data structures or formats, use abstractions and pass dependencies through function parameters. This promotes loose coupling, improves reusability, and allows functions to be more flexible and adaptable.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_