***SOLID Principle***

1. **Single Responsibility Principle**:

***A class should have only one reason to change.***

Example: Think of a chef in a restaurant. The chef's main job is to cook delicious food. If the chef also starts managing the finances of the restaurant, things might get confusing. Similarly, in coding, a class should do one main thing. If a class handles both user authentication and sending emails, it becomes harder to manage

1. **Open/Closed Principle (OCP):**

**s*oftware entities (such as classes, modules, functions) should be open for extension but closed for modification.***

Example: Think of your favorite video game. Even if new levels are added, the old ones don't change. Similarly, when you create code, try to design it in a way that you can add new features without changing the existing ones.

1. **Liskov Substitution Principle (LSP):**

***Objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program.*** Subtypes must be substitutable for their base types without altering the desirable properties of the program.

Example: Think about a toy store that sells different types of toys. If a customer buys a toy car, they expect it to roll smoothly. If you replace the toy car with a toy boat and it doesn't roll, customers would be confused. Similarly, in code, if you have a base class and a subclass, the subclass should work just like the base class wherever you use it

1. **Interface Segregation Principle (ISP):**

***Clients should not be forced to depend on interfaces they do not use.*** In other words, keep the interfaces small and specific to avoid requiring classes to implement methods they don't need.

Example: Imagine you have a smartwatch. It has features like tracking your steps and showing notifications. But what if you only want a watch that shows the time? You don't need all the other complicated stuff. Similarly, in coding, when you design interfaces, make sure they have only the methods that are relevant to the classes that use them.

1. **Dependency Inversion Principle (DIP):**

Dependency inversion talks about the coupling between the different classes or modules. The approach where the higher classes are not dependent on the lower classes instead depend upon the abstraction of the lower classes.***High-level modules should not depend on low-level modules; both should depend on abstractions, and abstractions should not depend on details.***

Example: Think of building with blocks. If you have a structure made of small blocks, you can change the blocks easily without changing the whole structure. In coding, this means instead of directly using specific classes, you use more general interfaces or abstract classes. This way, you can swap out parts without affecting the rest of your code.