**SOLID Principles:**

These Principles are used in OOD(Object Oriented Design) which helps us to keep the code

* modular
* maintainable
* less complex
* easy to understand
* avoid duplication

**Acronym of SOLID**

S - Single Responsibility Principle

O - Open/Closed Principle

L - Liskov Substitution Principle

I - Interface Segregation Principle

D – Dependency Inversion Principle

**Single Responsibility Principle:**

* This principle states that **each class should have** **one responsibility, one single purpose**. This means that a class will do only one job, which leads us to conclude it should have **only one reason to change**
* Any Method/Function/Class which we create should have one responsibility and should only have one reason to change
* Any Change request on class/function should be done by a single actor or group of actors who uses common thing

Helps in:

* Easy to maintain
* Easy to understand

Advantages:

* Adding multiple functionalities to single class can lead to complexity and hard to maintain
* Changing one functionality can effect other functionalities because they exist in a single class

**Examples:**

**Example 1:** We need to develop a Feature where we can manipulate a given text in different ways and Print the text in different ways

**Solutions:**  This feature can be developed in many ways, mentioning some below:

Approach 1: Developing both the features in a single place

Approach 2: Break Down the Feature into multiple parts where each part does one job

* One Class does the job of Text Manipulating
* One Cass does the job of Print the text

***Approach 2 will follow Single Responsibility principle***

**Example 2:** We need to develop a Feature where we need to add some Invoice and delete the invoices, generate reports and email reports

**Solutions:** This feature can be developed in many ways, mentioning some below:

Approach 1: Developing both the features in a single place

Approach 2: Break Down the Feature into multiple parts where each part does one job

* One Class for add and remove invoice
* One Class for generating Report
* Once Class for Sending email of report

***Approach 2 will follow Single Responsibility principle***

**Example 3: Feature to develop to compile a text and print the text**

**Solution:**

consider a module that compiles and prints a report. Imagine such a module can be changed for two reasons. First, the content of the report could change. Second, the format of the report could change. These two things change for different causes. The single responsibility principle says that these two aspects of the problem are really two separate [responsibilities](https://en.wikipedia.org/wiki/Interface_(computing)), and should, therefore, be in separate classes or modules. It would be a bad design to [couple](https://en.wikipedia.org/wiki/Coupling_(computer_programming)) two things that change for different reasons at different times.

The reason it is important to keep a class focused on a single concern is that it makes the class more robust. Continuing with the foregoing example, if there is a change to the report compilation process, there is a greater danger that the printing code will break if it is part of the same class.

**Example 4:**

**Consider a food delivery application that takes food orders, calculates the bill, and delivers it to customers. We can have 1 separate class for each of the tasks to be performed, and then the main class can just invoke those classes to get these actions done one after the other.**

**Link:** https://www.geeksforgeeks.org/single-responsibility-principle-in-java-with-examples/

**Open/Closed Principle:**

* Open for extension but closed for modification
* **Principle**: Once a class/function or component is written and tested we should avoid making changes to it
* Main Idea is to not update the existing classes/functions/object but extend there ability implement on top of it

**Helps in:**

* Maintainability
* Fexibility

**Advantages:**

* You don’t need to re-invent the wheel
* You focus on what is necessary
* You can avoid bugs

**Examples:**

Example 1: You have a class footballer which takes input of name,age and role. It is implemented using Switch cases. You need add some new roles into footballer and make it follow open/closed principle

Example 2: Update discount logic to support new discount rate

LINK:https://www.freecodecamp.org/news/open-closed-principle-solid-architecture-concept-explained/

**Liskov Substitution Principle:**

* Every Parent Class should be replaceable by it’s sub-class. If a class B extends a class A - then it should be possible to replace object of class A will object of class B without breaking any of the client’s behavior. In simple words**, child class should extend the capability of parent class and not narrow it down.**
* Eg: If class B is sub-type of class A, then object of class B should be able to replace the class A without breaking the behavior of the program

Eg: you have a bird class and have eagle and penguin classes and want to replace bird class with penguin class

<https://leetcode.com/discuss/interview-question/3146290/SOLID-Principles>

**Interface Segregation Principle:**

* ­­­Interfaces should be in such a way that client should not need to implement unnecessary functions that they don’t need
* Clients should not be forced to implement methods that they don’t use. Try to make your interfaces narrow enough that client classes don’t have to implement behaviors they don’t need.

Eg:

1. In a restaurant there are multiple actors chef, waiter, cleaner all can’t use a single interface because there responsibilities are different from role to role so we have to segregate the interfaces from role to role
2. In case of servers one interface of Cloud Provider can’t to used for every cloud provider because each provider has there own features

**Dependency Inversion Principle:**

* **Two Key Points:**
  + High-level modules should not depend on low-level modules. Both should depend on abstractions.
  + Abstractions should not depend on details. Details should depend on abstractions.
* High-level classes shouldn’t depend on low-level classes. Both should depend on abstractions. Abstractions shouldn’t depend on details. Details should depend on abstractions.
* Class should depend on abstract class rather than concrete classes

Eg:

<https://leetcode.com/discuss/interview-question/3146290/SOLID-Principles>