Object-Oriented Programming is a methodology or paradigm to design a program using classes and objects. It simplifies software development and maintenance by providing some concepts:

ADVERTISEMENT

ADVERTISEMENT

- Object
- Class
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

Apart from these concepts, there are some other terms which are used in Object-Oriented design:

- Coupling
- Cohesion
- Association
- Aggregation
- Composition

Object

Any entity that has state and behavior is known as an object.

An Object can be defined as an instance of a class. An object contains an address and takes up some space in memory. Objects can communicate without knowing the details of each other's data or code. The only necessary thing is the type of message accepted and the type of response returned by the objects

Class

Collection of objects is called class. It is a logical entity.

A class can also be defined as a blueprint from which you can create an individual object. Class doesn't consume any space.

Inheritance

Inheritance is an oops concept When one object acquires all the properties and behaviors of a parent object. It provides code reusability. It is used to achieve runtime polymorphism.

Polymorphism

Polymorphism is an oops concept in which one *task is performed in different ways*. For example: to convince the customer differently, to draw something, for example, shape, triangle, rectangle, etc.

In Java, we use method overloading and method overriding to achieve polymorphism.

Another example can be to speak something; for example, a cat speaks meow, dog barks woof, etc.

Abstraction

Abstraction is the oops concept of hiding internal details and showing functionality. For example phone call, we don't know the internal processing.

In Java, we use abstract class and interface to achieve abstraction.

Encapsulation

Encapsulation is the oops concept of Binding (or wrapping) code and data together into a single unit. For example, a capsule is wrapped with different medicines.

A java class is an example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.

Association

Association represents the relationship between objects. Here, one object can be associated with one object or many objects. There can be four types of association between the objects:

- One to One
- One to Many
- Many to One, and
- Many to Many

Let's understand the relationship with real-time examples. For example, One country can have one prime minister (one to one), and a prime minister can have many ministers (one to many). Also, many MP's can have one prime minister (many to one), and many ministers can have many departments (many to many).

Association can be unidirectional or bidirectional.

Aggregation

Aggregation is a way to achieve Association. Aggregation represents the relationship where one object contains other objects as a part of its state. It represents the weak relationship between objects. It is also termed as a *has-a* relationship in Java. Like, inheritance represents the *is-a* relationship. It is another way to reuse objects.

Composition

The composition is also a way to achieve Association. The composition represents the relationship where one object contains other objects as a part of its state. There is a strong relationship between the containing object and the dependent object. It is the state where containing objects do not have an independent existence. If you delete the parent object, all the child objects will be deleted automatically.

Cohesion

Cohesion refers to the level of a component which performs a single well-defined task. A single well-defined task is done by a highly cohesive method. The weakly cohesive method will split the task into separate parts. The java.io package is a highly cohesive package because it has I/O related classes and interface. However, the java.util package is a weakly cohesive package because it has unrelated classes and interfaces.

Coupling

Coupling refers to the knowledge or information or dependency of another class. It arises when classes are aware of each other. If a class has the details information of another class, there is strong coupling. In Java, we use private, protected, and public modifiers to display the visibility level of a class, method, and field. You can use interfaces for the weaker coupling because there is no concrete implementation.

==

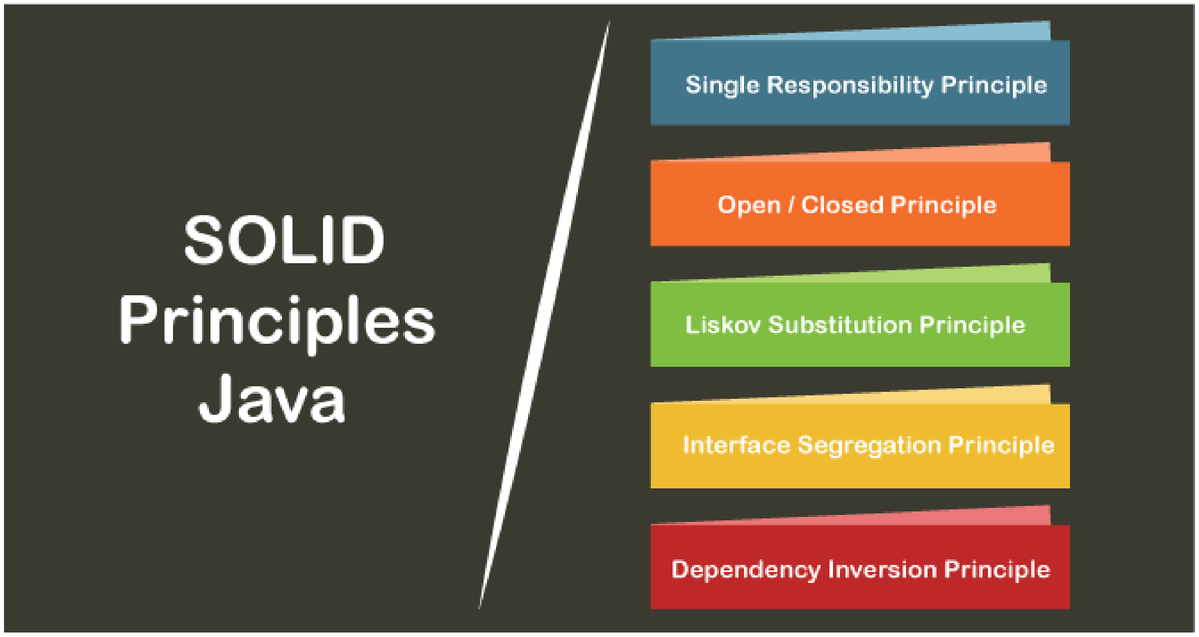
https://www.javatpoint.com/solid-principles-java

SOLID Principles Java

In Java, **SOLID principles** are an object-oriented approach that are applied to software structure design. It is conceptualized by **Robert C. Martin** (also known as Uncle Bob). These five principles have changed the world of object-oriented programming, and also changed the way of writing software. It also ensures that the software is modular, easy to understand, debug, and refactor. In this section, we will discuss **SOLID principles in Java** with proper example**.**

The word SOLID acronym for:

- Single Responsibility Principle (SRP)
- Open-Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)


SOLID
Principles
Java
Single Responsibility Principle
Open / Closed Principle
Liskov Substitution Principle
Interface Segregation Principle
Dependency Inversion Principle

Let's explain the principles one by one in detail.

Single Responsibility Principle

The single responsibility principle states that **every Java class must perform a single functionality**. Implementation of multiple functionalities in a single class mashup the code and if any modification is required may affect the whole class. It precise the code and the code can be easily maintained. Let's understand the single responsibility principle through an example.

ADVERTISEMENT

Suppose, **Student** is a class having three methods namely **printDetails(), calculatePercentage(),** and **addStudent().** Hence, the Student class has three responsibilities to print the details of students, calculate percentages, and database. By using the single responsibility principle, we can separate these functionalities into three separate classes to fulfill the goal of the principle.

Student.java

```
public class Student
{
public void printDetails();
{
//functionality of the method
}
pubic void calculatePercentage();
{
//functionality of the method
}
public void addStudent();
{
//functionality of the method
}
}
```

The above code snippet violates the single responsibility principle. To achieve the goal of the principle, we should implement a separate class that performs a single functionality only.

Student.java

```
public class Student
{
public void addStudent();
{
//functionality of the method
}
}
```

PrintStudentDetails.java

```
public class PrintStudentDetails
{
```

```
public void printDetails();

{

//functionality of the method

}

}
```

Percentage.java

ADVERTISEMENT

ADVERTISEMENT

```
public class Percentage

{

public void calculatePercentage();

{

//functionality of the method

}

}
```

Hence, we have achieved the goal of the single responsibility principle by separating the functionality into three separate classes.

Open-Closed Principle

The application or module entities the methods, functions, variables, etc. The open-closed principle states that according to new requirements **the module should be open for extension but closed for modification.** The extension allows us to implement new functionality to the module. Let's understand the principle through an example.

Suppose, **VehicleInfo** is a class and it has the method **vehicleNumber()** that returns the vehicle number.

VehicleInfo.java

```
public class VehicleInfo

{

public double vehicleNumber(Vehicle vcl)

{

if (vcl instanceof Car)
```

```
{
return vcl.getNumber();
if (vcl instanceof Bike)
{
return vcl.getNumber();
}
}
```

If we want to add another subclass named Truck, simply, we add one more if statement that violates the open-closed principle. The only way to add the subclass and achieve the goal of principle by overriding the **vehicleNumber()** method, as we have shown below.

VehicleInfo.java

```
public class VehicleInfo
{
public double vehicleNumber()
{
//functionality
}
}
public class Car extends VehicleInfo
{
public double vehicleNumber()
{
return this.getValue();
}
public class Car extends Truck
{
public double vehicleNumber()
{
return this.getValue();
}
```

Similarly, we can add more vehicles by making another subclass extending from the vehicle class. the approach would not affect the existing application.

ADVERTISEMENT

ADVERTISEMENT

Liskov Substitution Principle

The Liskov Substitution Principle (LSP) was introduced by **Barbara Liskov**. It applies to inheritance in such a way that the **derived classes must be completely substitutable for their base classes**. In other words, if class A is a subtype of class B, then we should be able to replace B with A without interrupting the behavior of the program.

It extends the open-close principle and also focuses on the behavior of a superclass and its subtypes. We should design the classes to preserve the property unless we have a strong reason to do otherwise. Let's understand the principle through an example.

Student.java

```
public class Student
{
private double height;
private double weight;
public void setHeight(double h)
{
height = h;
}
public void setWeight(double w)
{
weight= w;
}
...
}
public class StudentBMI extends Student
{
public void setHeight(double h)
{
super.setHeight(h);
super.setWeight(w);
}
public void setWeight(double h)
{
super.setHeight(h);
super.setWeight(w);
}
}
```

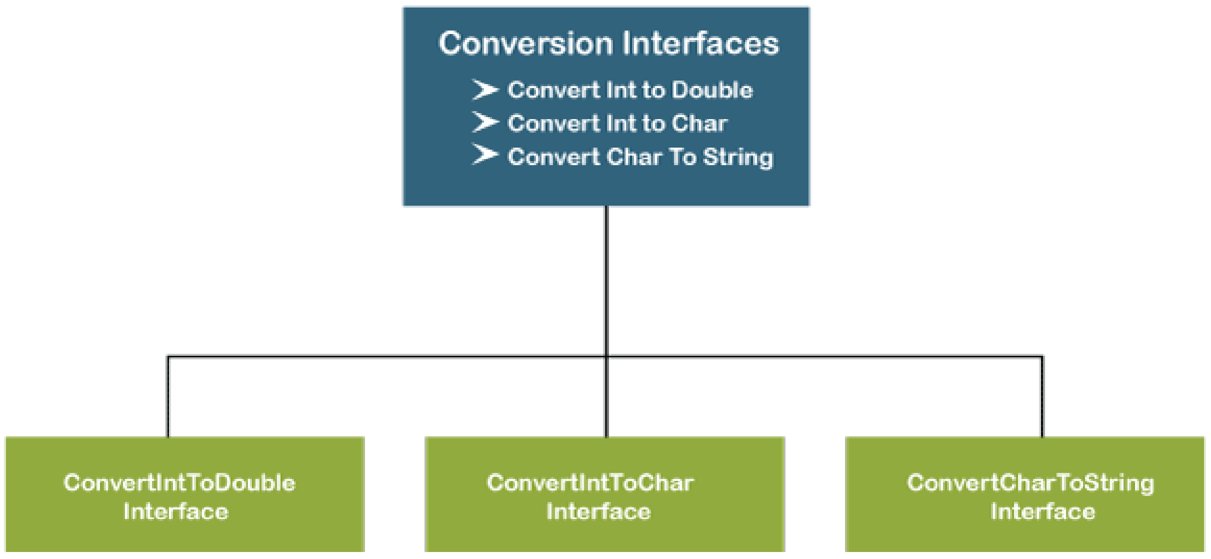
The above classes violated the Liskov substitution principle because the StudentBMI class has extra constraints i.e. height and weight that must be the same. Therefore, the Student class (base class) cannot be replaced by StudentBMI class (derived class).

Hence, substituting the class Student with StudentBMI class may result in unexpected behavior.

Interface Segregation Principle

The principle states that the larger interfaces split into smaller ones. Because the implementation classes use only the methods that are required. We should not force the client to use the methods that they do not want to use.

The goal of the interface segregation principle is similar to the single responsibility principle. Let's understand the principle through an example.


Conversion Interfaces
Convert Int to Double
Convert Int to Char
Convert Char To String
ConvertIntToDouble Interface
ConvertIntToChar Interface
ConvertCharToString Interface

Suppose, we have created an interface named **Conversion** having three methods **intToDouble(), intToChar(),** and **charToString()**.

```
public interface Conversion

{

public void intToDouble();

public void intToChar();

public void charToString();

}
```

The above interface has three methods. If we want to use only a method intToChar(), we have no choice to implement the single method. To overcome the problem, the principle allows us to split the interface into three separate ones.

```
public interface ConvertIntToDouble

{

public void intToDouble();

}

public interface ConvertIntToChar

{

public void intToChar();

}

public interface ConvertCharToString

{

public void charToString();

}
```

Now we can use only the method that is required. Suppose, we want to convert the integer to double and character to string then, we will use only the methods **intToDouble()** and **charToString().**

```
public class DataTypeConversion implements ConvertIntToDouble, ConvertCharToString

{

public void intToDouble()

{
```

```
//conversion logic
}
public void charToString()
{
//conversion logic
}
}
```

Dependency Inversion Principle

The principle states that we must use abstraction (abstract classes and interfaces) instead of concrete implementations. High-level modules should not depend on the low-level module but both should depend on the abstraction. Because the abstraction does not depend on detail but the detail depends on abstraction. It decouples the software. Let's understand the principle through an example.

```
public class WindowsMachine
{
//functionality
}
```

It is worth, if we have not keyboard and mouse to work on Windows. To solve this problem, we create a constructor of the class and add the instances of the keyboard and monitor. After adding the instances, the class looks like the following:

```
public class WindowsMachine
{
public final keyboard;
public final monitor;
public WindowsMachine()
{
monitor = new monitor();  //instance of monitor class
keyboard = new keyboard(); //instance of keyboard class
}
}
```

Now we can work on the Windows machine with the help of a keyboard and mouse. But we still face the problem. Because we have tightly coupled the three classes together by using the new keyword. It is hard o test the class windows machine.

To make the code loosely coupled, we decouple the WindowsMachine from the keyboard by using the Keyboard interface and this keyword.

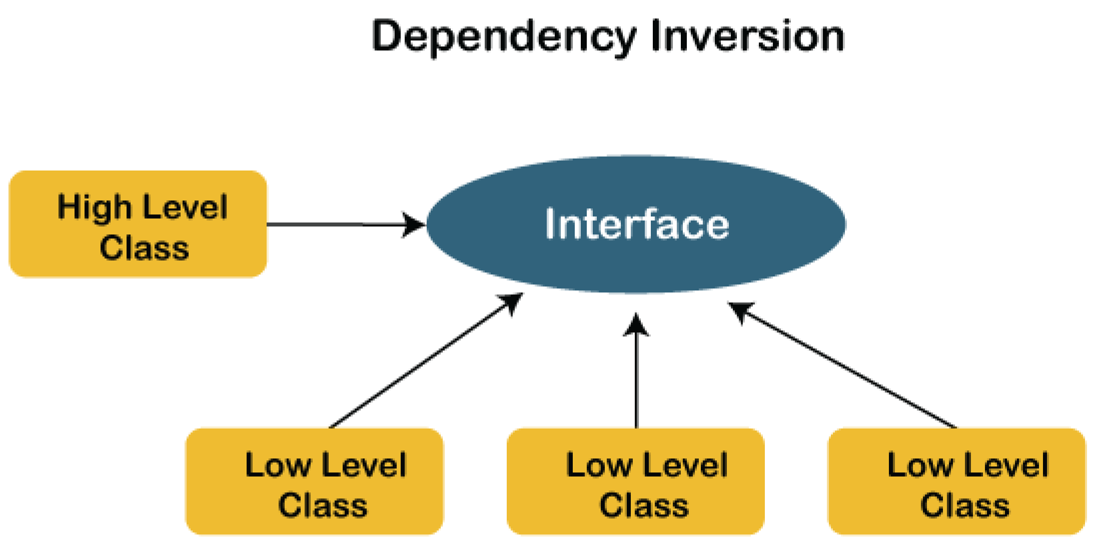
Keyboard.java

```
public interface Keyboard
{
//functionality
}
```

WindowsMachine.java

```
public class WindowsMachine
{
private final Keyboard keyboard;
private final Monitor monitor;
public WindowsMachine(Keyboard keyboard, Monitor monitor)
{
this.keyboard = keyboard;
this.monitor = monitor;
}
}
```

In the above code, we have used the dependency injection to add the keyboard dependency in the WindowsMachine class. Therefore, we have decoupled the classes.



Why should we use SOLID principles?

- It reduces the dependencies so that a block of code can be changed without affecting the other code blocks.
- The principles intended to make design easier, understandable.
- By using the principles, the system is maintainable, testable, scalable, and reusable.
- It avoids the bad design of the software.

Next time when you design software, keeps these five principles in mind. By applying these principles, the code will be much more clear, testable, and expendable.

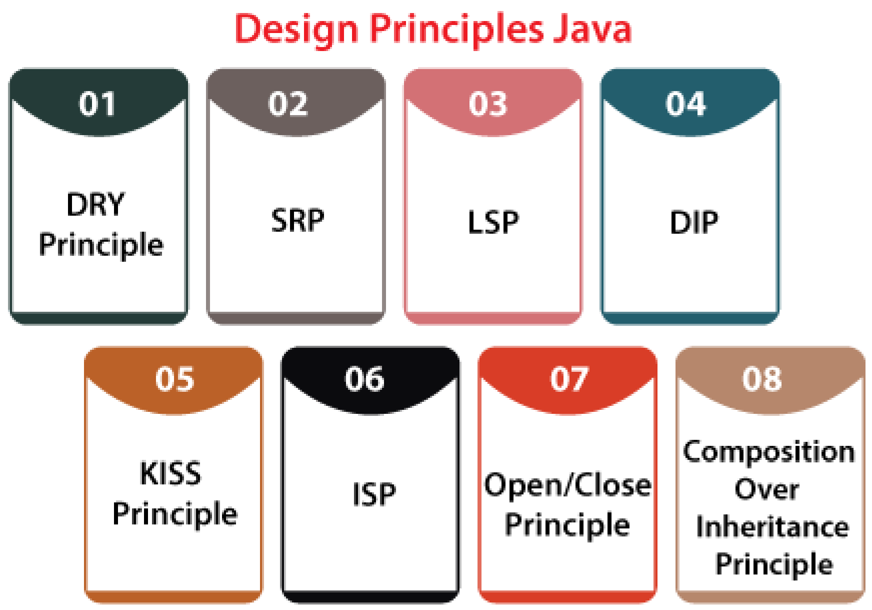
Also check below links for other design principles.

https://www.javatpoint.com/design-principles-in-java

Design Principles in Java

In Java, the design principles are the set of advice used as rules in design making. In Java, the design principles are similar to the design patterns concept. The only difference between the design principle and design pattern is that the design principles are more generalized and abstract.

The design pattern contains much more practical advice and concrete. The design patterns are related to the entire class problems, not just generalized coding practices.



There are some of the most important design principles are as follows:

SOLID Principles

1. SRP
2. LSP
3. ISP
4. Open/Closed Principle
5. DIP

Other Principles

ADVERTISEMENT

6. DRY Principles
7. KISS Principle
8. Composition Over Inheritance Principle

Let's understand all the principles one by one:

DRY Principle

The DRY principle stands for the **Don't Repeat Yourself** principle. It is one of the common principles for all programming languages. The DRY principle says:

Within a system, each piece of logic should have a single unambiguous representation.

Let's take an example of the DRY principle

```
public class Animal {
   public void eatFood() {
      System.out.println("Eat Food");
   }
}


public class Dog extends Animal {
   public void woof() {
      System.out.println("Dog Woof! ");
   }
}
public class Cat extends Animal {
   public void meow() {
      System.out.println("Cat Meow!");
   }
}
```

Both Dog and Cat speak differently, but they need to eat food. The common functionality in both of them is Eating food so, we can put this common functionality into the parent class, i.e., Animals, and then we can extend that parent class in the child class, i.e., Dog and Cat.

Now, each class can focus on its own unique logic, so there is no need to implement the same functionality in both classes.

ADVERTISEMENT

ADVERTISEMENT

```
Dog obj1 = new Dog();
obj1.eatFood();
obj1.woof();


Cat obj2 = new Cat();
obj2.eatFood();
```

```
    obj2.meow();
```

After compile and run the above code, we will see the following output:

Output:

```
Eat food

Cat Meow!

Eat food

Dog Woof!
```

Violation of the DRY Principle

Violations of the DRY principles are referred to as WET solutions. WET is an abbreviation for the following things:

1. Write everything twice
2. We enjoy typing
3. Write every time
4. Waste everyone's twice.

These violations are not always bad because repetitions are sometimes advisable to make code less inter-dependent, more readable, inherently dissimilar classes, etc.

SRP

SRP is another design principle that stands for **the Single Responsibility Principle**. The SRP principle says that in one class, there should never be two functionalities. It also paraphrases as:

The class should only have one and only one reason to be changed. When there is more than one responsibility, there is more than one reason to change that class at the same point. So, there should not be more than one separate functionality in that same class that may be affected.

ADVERTISEMENT

ADVERTISEMENT

The SRP principle helps us to

1. Inherit from a class without inheriting or implementing the methods which our class doesn't need.
2. Deal with bugs.
3. Implement changes without confusing co-dependencies.

LSP

LSP is another design principle that stands for **Liskov Substitution Principle**. The **LSP** states that the derived class should be able to substitute the base classes without changing our code.

The LSP is closely related to the SRP and ISP, So, violation of either SRP or ISP can be a violation (or become) of the LSP. The reason for violation in LSP because if a class performs more than one function, subclasses extending it are less likely to implement those two or more functionalities meaningfully.

ADVERTISEMENT

ADVERTISEMENT

DIP

Another most important design principle is DIP, i.e., **Dependency Inversion Principle**. This principle states that the low and high levels are decoupled so that the changes in low-level modules don't require rework of high-level modules.

The high-level and low-level modules should not be dependent on each other. They should be dependent on the abstraction, such as interfaces. The Dependency Inversion Principle also states that the details should be dependent on the abstraction, not abstraction should be dependent on the details.

KISS Principle

ADVERTISEMENT

ADVERTISEMENT

It is another designing principle that stands for **Keep It Simple and Stupid Principle**. This principle is just a reminder to keep our code readable and simple for humans. If several use cases are handled by the method, we need to split them into smaller functions.

The KISS principle states that for most cases, the stack call should not severely affect our program's performance until the efficiency is not extremely important.

On the other hand, the lengthy and unreadable methods will be very difficult for human programmers to maintain and find bugs. The violation of the DRY principle can be done by ourselves because if we have a method that performs more than one task, we cannot call that method to perform one of them. So, we will create another method for that.

ISP

ISP is another design principle that stands for **Interface Segregation Principle**. This principle states that the client should never be restricted to dependent on the interfaces that are not using in their entirety. This means that the interface should have the minimal set of methods required to ensure functionality and be limited to only one functionality.

For example, if we create a Burger interface, we don't need to implement the addCheese() method because cheese isn't available for every Burger type.

Let's assume that all burgers need to be baked and have sauce and define that burger interface in the following way:

```
public interface Burger{
  void addSauce();
  void bakeBurger();
}
```

ADVERTISEMENT

ADVERTISEMENT

Now, let's implement the Burger interface in VegetarianBurger and CheeseBurger classes.

```
public class VegetarianBurger implements Burger {
public void addTomatoAndCabbage() {
    System.out.println("Adding Tomato and Cabbage vegitables");
  }
    @Override
    public void addSauce() {
    System.out.println("Adding sauce in vegetarian Burger");
  }
@Override
    public void bake() {
    System.out.println("Baking the vegetarian Burger");
  }
}
public class CheeseBurger implements Burger {
public void addCheese() {
    System.out.println("Adding cheese in Burger");
  }
@Override
public void addSauce() {
    System.out.println("Adding sauce in Cheese Burger");
```

```
        }
    @Override
        public void bake() {
        System.out.println("Baking the Cheese Burger");
        }
    }
```

The VagitarianBurger has Cabbage and Tomato, whereas the CheeseBurger has cheese but needs to be baked and sauce that is defined in the interface. If the addCheese() and addTomatoandCabbage() methods were located in the Burger interface, both the classes have to implement them even though they don't need both.

Open/Close Principle

Open/Closed Principle is another important design principle that comes in the category of the **SOLID** principles. This principle states that the methods and objects or classes should be closed for modification but open for modification. In simple words, this principle says that we should have to implement our modules and classes with the possible future update in mind. By doing that, our modules and classes will have a generic design, and in order to extend their behavior, we would not need to change the class itself.

We can create new fields or methods but in such a way that we don't need to modify the old code, delete already created fields, and rewrite the created methods.

The Open/Close principle is mainly used to prevent regression and ensure backward compatibility.

Composition Over Inheritance Principle

The Composition over inheritance principle is another important design principle in Java. This principle helps us to implement flexible and maintainable code in Java. The principle states that we should have to implement interfaces rather than extending the classes. We implement the inheritance when

1. The class need to implement all the functionalities
2. The child class can be used as a substitute for our parent class.

In the same way, we use Composition when

1. The class needs to implement some specific functionalities.