

**CODE AND BLOG ASSIGNMENT ON**

**SOLID PRINCIPLES**

**SOLID Principles** are object-oriented design concepts relevant to software development.

There are five of Solid Principles. **SOLID** is an acronym that stands for five key design principles:

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

**1.Single Responsibility Principle (SRP)**

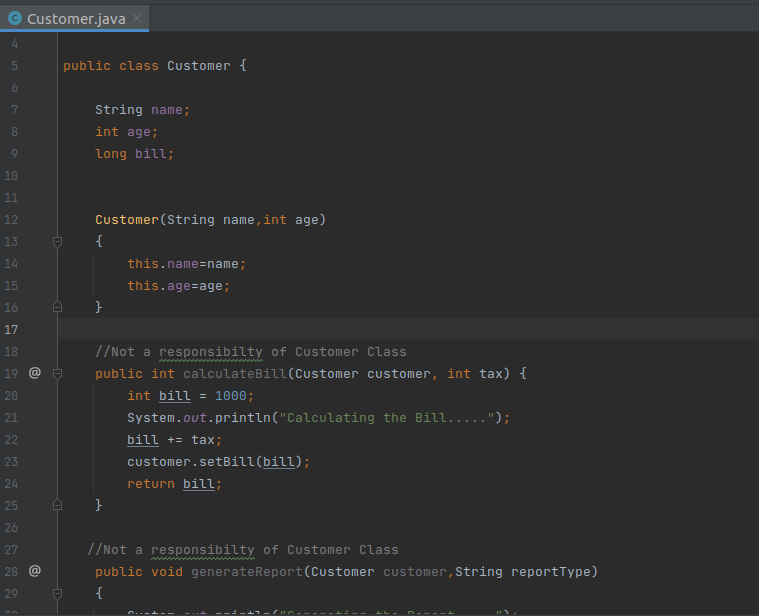
The single responsibility principle states that every Java class must perform a single functionality or single responsibilty. This means that a class will do only one job, which leads us to conclude, it should have only one reason to change.

If you have more than one reason to change the class then refactor the class into multiple classes according to functionality.

EXAMPLE:

Let’s take a look at an example of Single Responsibility Principle:

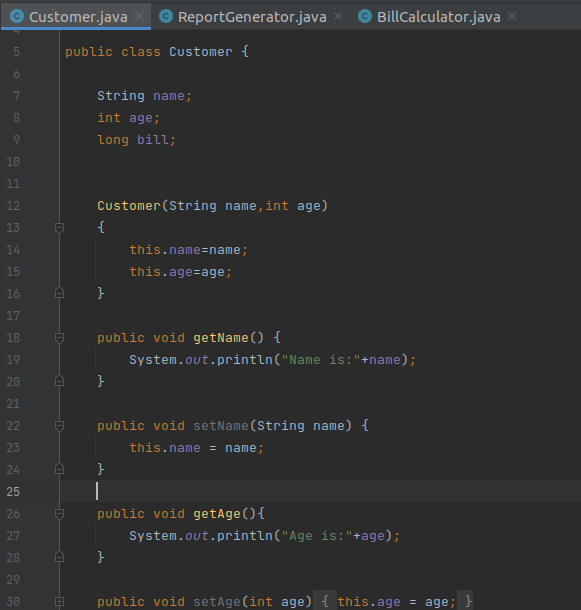
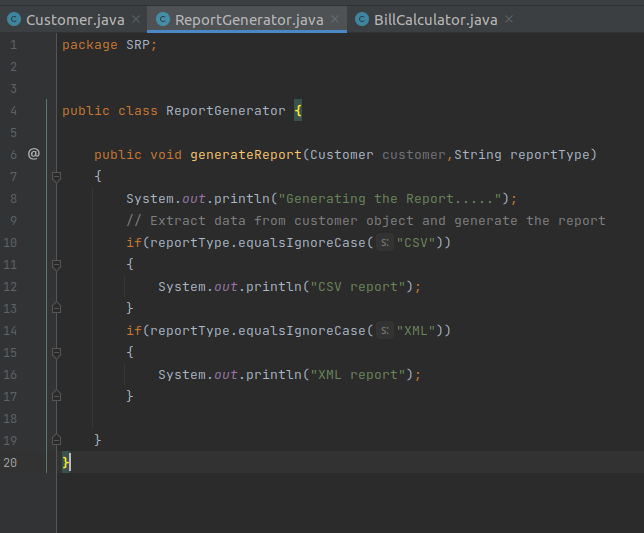
We have Customer class which have some responsibility like we have here,



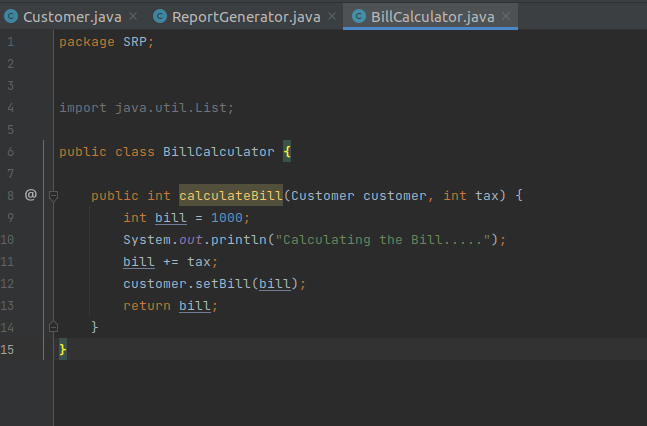
Here,calculation of bill and report generation should not be the responsibility of Customer.

Because there will also be problem when we want to add more functions or modify the existing one. So,we should create different classes for these functionalities. Like,

Customer.java ReportGenerator.java



BillCalculator.java



So the best practice is to isolate all the functions into different classes.

Hence ,the Single Responsibility Principle is achieved.

## 2.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.

The Liskov Substitution Principle (LSP) 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.

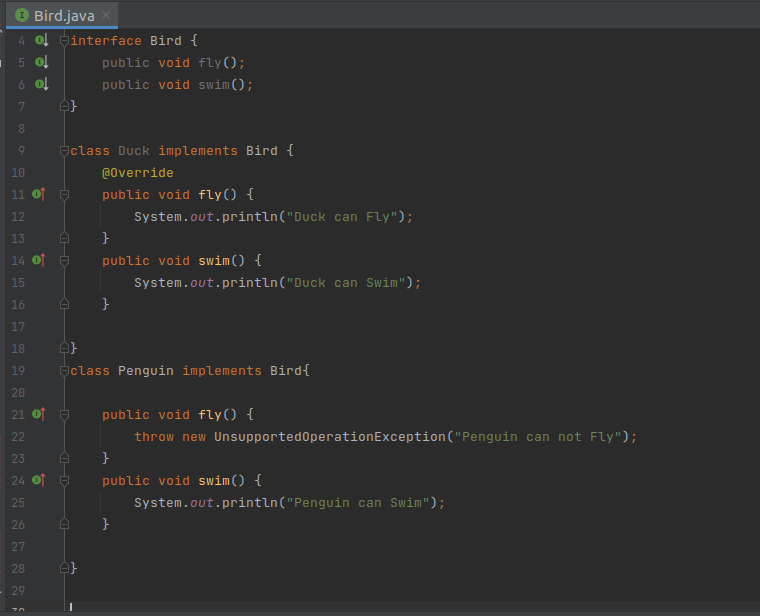
EXAMPLE:

Let’s take a look at an example of Single Responsibility Principle:

We have a Bird interface which has methods Fly() and swim().Then we are Defining a class Duck which will implement this Bird interface and will define method fly() and swim() in it.

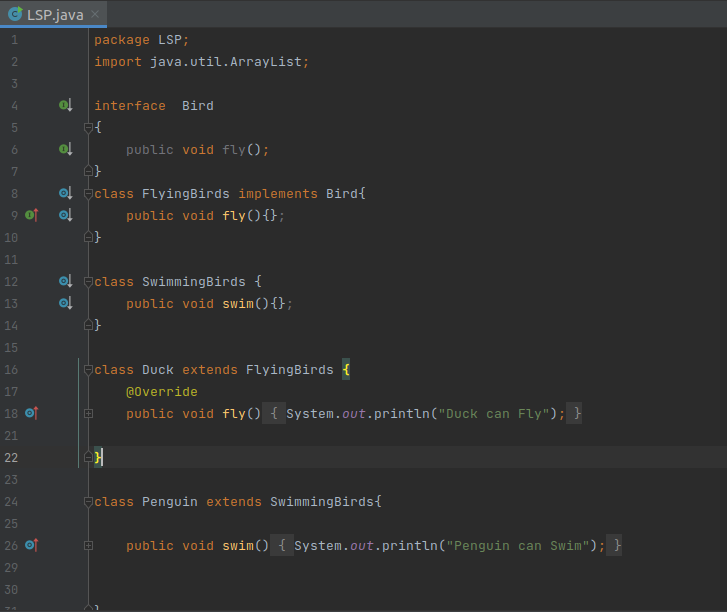
Now a new requirement comes to define Penguin bird which can’t fly but can swim. We can define this as:

Bird.java



Here,as soon as an Penguin instance is passed, it will throw an error.

The sub type(Penguin) is not replaceable for the super type(Bird). Hence violating the LSP. So,we can design this another way as:



Here,Flying birds implement bird interface and swimming bird does not. The sub type(Penguin) is replaceable for the super type(Bird).

Hence, the Liskov Substitution Principle in achieved.

**3.Dependency Inversion Principle**

The general idea of this principle is: High-level modules, which provide complex logic, should be easily reusable and unaffected by changes in low-level modules, which provide utility features. To achieve that, we need to introduce an abstraction that decouples the high-level and low-level modules from each other.

The Dependency Inversion Principle consists of two parts:

* 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.

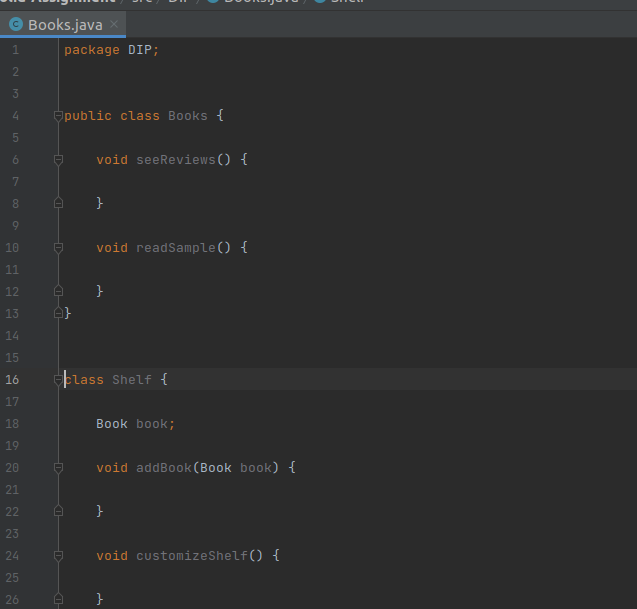
EXAMPLE:

Let’s take a look at an example of Single Responsibility Principle:

Suppose a bookstore asked us to build a new feature that enables customers to put their favorite books on a shelf. In order to implement the new functionality, we create a low-level Book class and a high-level Shelf class.

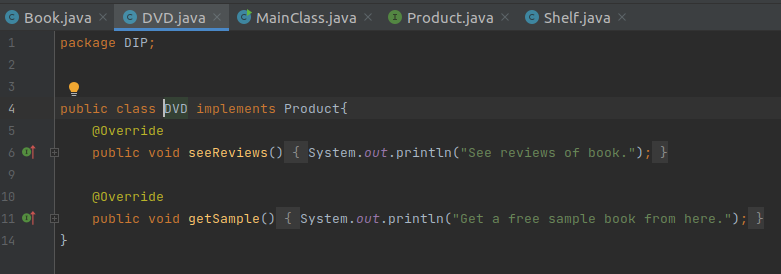
The book class will allow users to see reviews & read sample books.

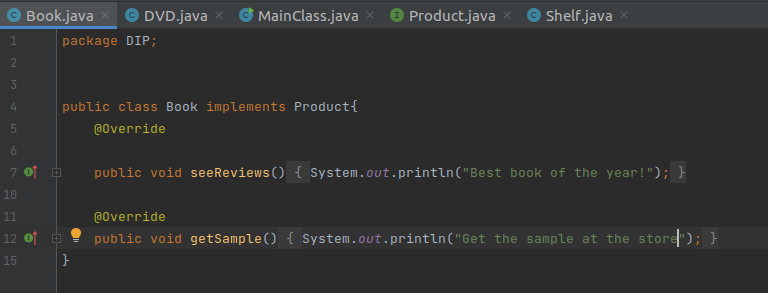
The shelf class will let them add a book to their shelf and customize the shelf.



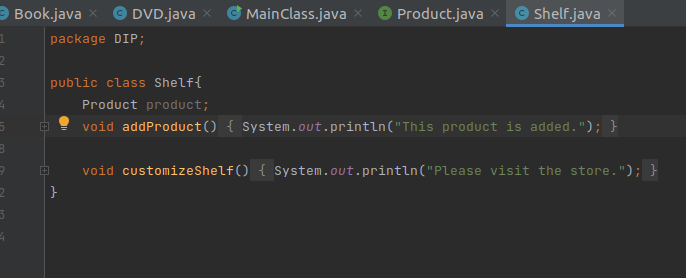
Here,the high-level Shelf class depends on the low-level Book, hence the above code violates the Dependency Inversion Principle. In order to fulfill the demand, we create a new DVD class.

It is to create an abstraction layer for the lower-level classes (Book and DVD).So we have added the Product interface, which both the classes will implement.





Now, Shelf can reference the Product interface instead of its implementations (Book and DVD). The refactored code also allows us to later introduce new product types (for instance, Magazine) that customers can put on their shelves, too.



Now,we’ve implemented the Dependency Inversion Principle, as in the refactored code, high-level classes don’t depend on low-level classes, either.

Hence, Dependency Inversion Principle is achieved