S.O.L.I.D. Principles

Of

Object Oriented Programming

SOLID principles are a set of principles that are used in Object Oriented Programming.

They set the standard of how to write the programs in any Object Oriented Programming language.

SOLID principles help to write a program that can more easily respond to changes, are easy to maintain and cost less time to work with.

SOLID principles can be applied to any OOP program.

SOLID principles are actually a set of 5 principles that are :

1. **S.R.P  (Single Responsibility Principle)**
2. **O.C.P (Open/Closed Principle)**
3. **L.S.P (Liskov Substitution Principle)**
4. **I.S.P (Interface Segregation Principle)**
5. **D.I.P (Dependency Inversion Principle)**

Here we will take a deep dive into the first three principles and will understand their implementation with the help of code examples.

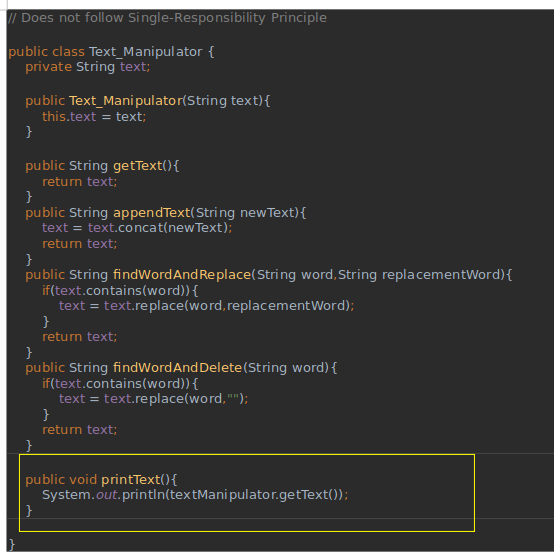
**S.R.P (Single Responsibility Principle)**

The **single-responsibility principle** states that a class, module or function should perform only one job. In other words, it should have all the responsibilities of a single functionality.

Martin explained this by saying **“a class should have only one reason to change”**. Here the “reason” is that we want to change the single functionality this class pursues. If we do not want this single functionality to change, we will never change this class because all components of the class should relate to that behavior.

SRP makes it easy to maintain the code as a class has only one responsibility and so when changing any functionality related to that class responsibility requires only a single class to modify.

Let us take an example to make it more clear.

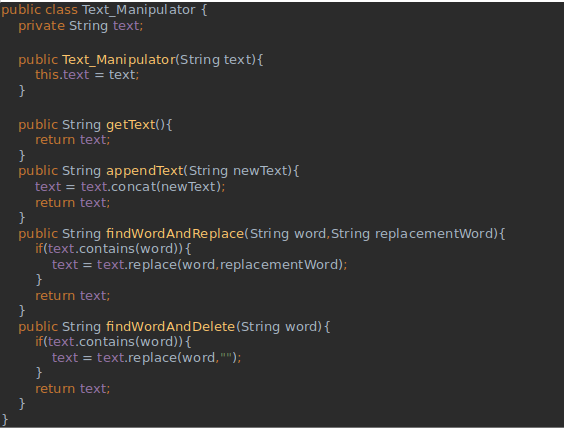


In the above example we have a Text\_Manipulator class that takes text as input and performs some manipulations over the text, but in the Text\_Manipulator class we have a method named printText that prints the text on the console.

Here the Text\_Manipulator class has all the methods related to a single responsibility that is manipulating the text in some manner except the printText method as it is printing the text to the console and is having a different responsibility of printing.So, when we will be required to change in the functionality of the printing we will have to modify the Text\_manipulator class and this is violating the single-responsibility principle as a **class should have only one reason to change.**

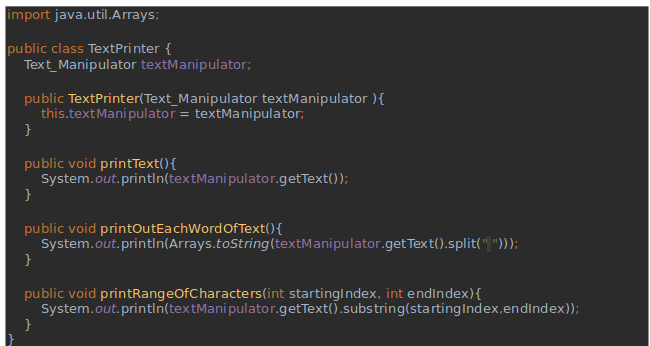
To achieve the SRP in the above example we will have to separate the printing responsibility by making a separate class for printing related tasks.

Lets see how we can do this:



Here we have removed the printText method from the Text\_Manipulator class in order to follow the SRP and created a separate class TextPrinter for the responsibility related to printing of the text.

The Text\_Manipulator class is now handling only the responsibilities related to the manipulation of the text.



And the TextPrinter class is handling the responsibilities related to printing of the text.

Now both the classes have only one single responsibility and thus helping to achieve SRP.

**L.S.P (Liskov Substitution Principle)**

Liskov substitution principles define that the object reference of a superclass shall be replaceable with objects of its subclass without breaking the application.It states that :

**“If S is a subtype of T, then objects of type T may be replaced with objects of type S (i.e., an object of type T may be substituted with any object of a subtype S) without altering any of the desirable properties of the program”.**

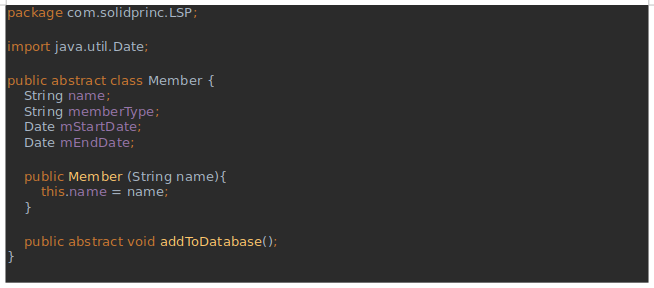
Functions that use references to base classes must be able to use objects of the derived class without knowing it.

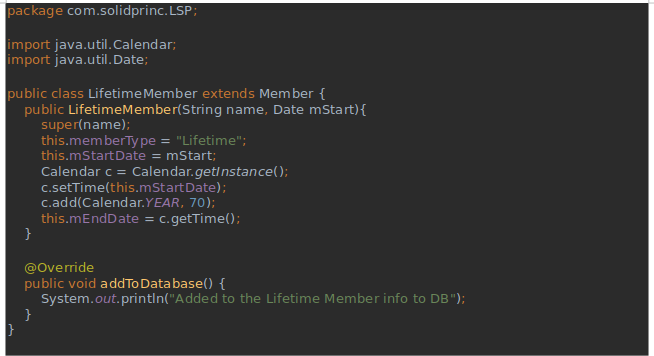
**Liskov Substitution principle (LSP)** is a particular definition of a subtyping relation, called (strong) behavioral subtyping.

In other words, a subclass should completely inherit the behavior of its parent class and their objects are replaceable with the base class object without making any unintended modifications.

Let us understand this principle more with a help of an example:

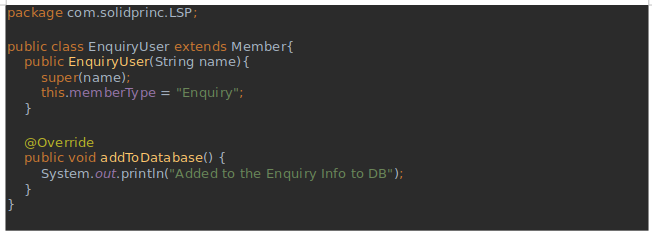
Below in the code we have an abstract class Member that takes multiple string input with an abstract method **“addToDatabase”.**



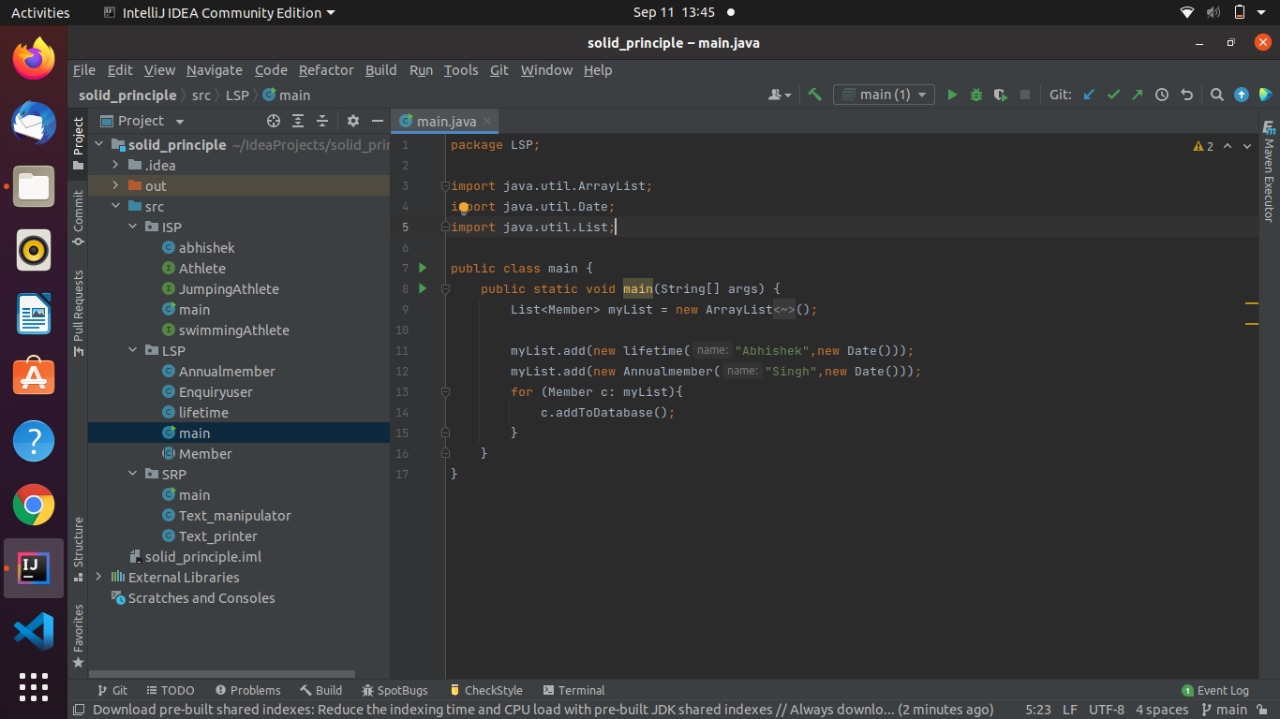


LifetimeMember Class is a subtype of class Member that is overriding the parent class addToDatabse method .

Two more classes of AnnualMember and Enquiry user are there that extend the Member Class.



Now to check whether the above example is following LSP principle or not we require a main class. So let’s see our main class ---



Here in the main class we have made a list of parent type i.e. “**Member”** class but at the run time we are assigning it with the instance of children classes.

In the loop we are running the “**addToDatabase ''** method over each member of the list to check for the LSP and the code worked flowlessly, hence satisfying the **Liskov Substitution Principle.**

Now if we add a **“addBooking”** method in the “**Member”** class then the subclass “**EnquiryUser”** would not be able to substitute this method completely as the Enquiry User does not make booking and overriding that method will have different behavior of that method in the subclass and this will violate the **Liskov Substitution Principle** that says a “**child class should completely substitute its parent class**”.

To achieve LSP is such a case we can make use of Interface to implement the method “**addBooking**” and subclasses will implement the method as per use.

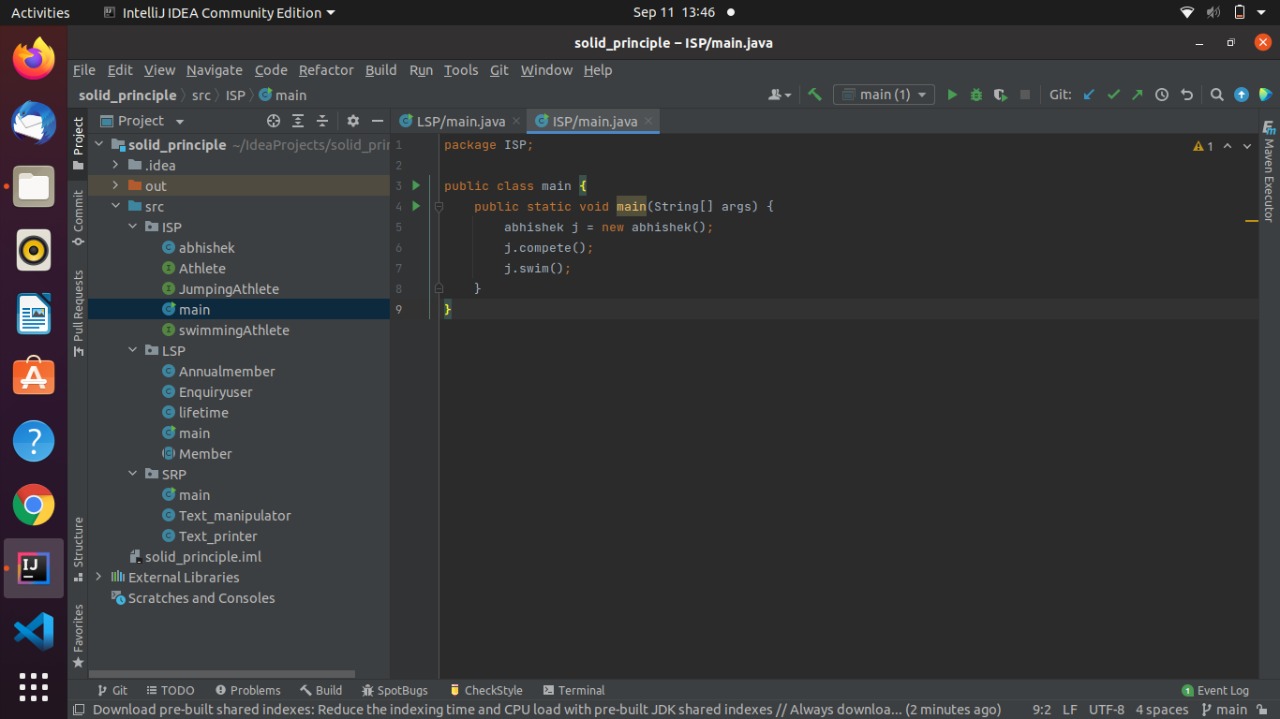
These are the first-three SOLID principles that we have learnt with the help of some good examples in java.

**I.S.P (Interface Segregation Principle)**

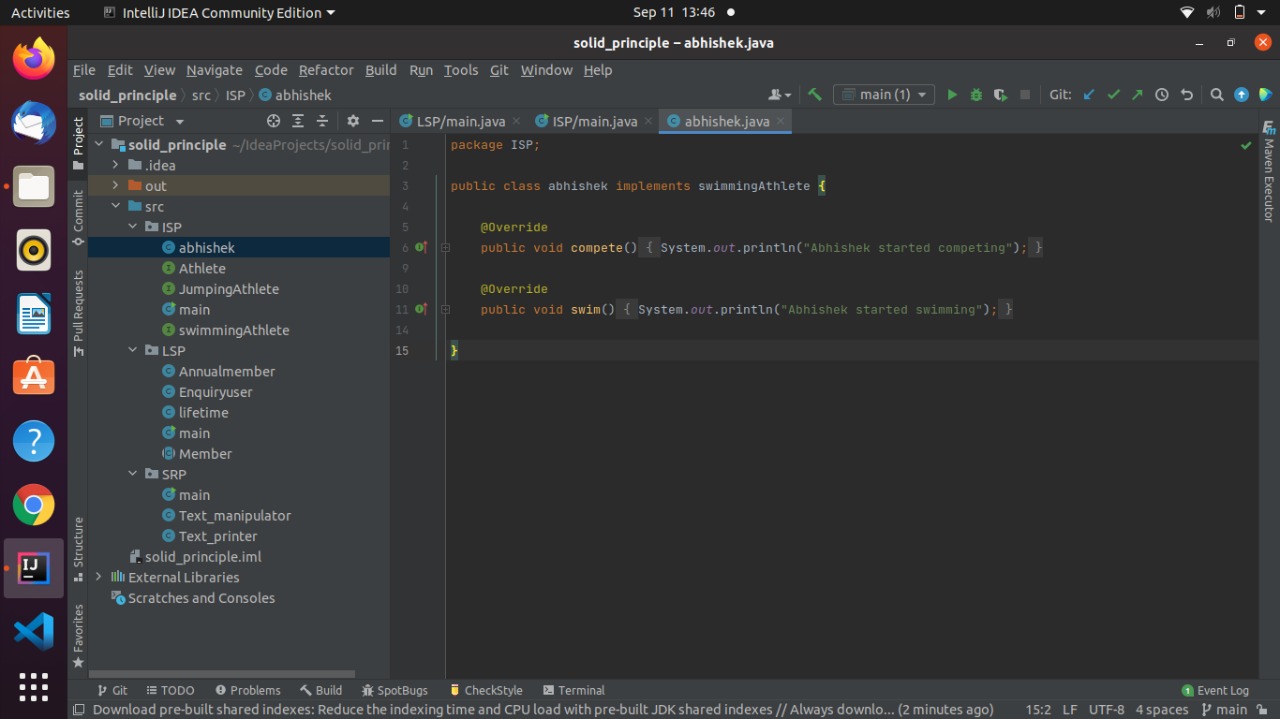
The interface segregation principle (ISP) states that no client should be forced to depend on methods it does not use.

Imagine an interface with many methods in our codebase and that many of our classes implement this interface, although only some of its methods are implemented.

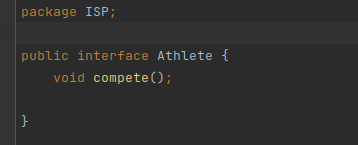
In our case, the Athlete interface is an interface with some actions of an athlete:



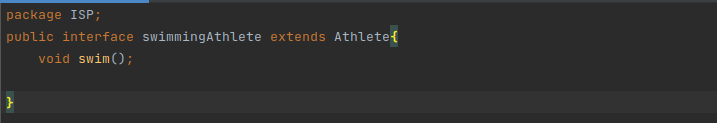
We have added the method compete, but also there some extra methods like swim() and compete()Suppose that sachin is a swimming athlete.

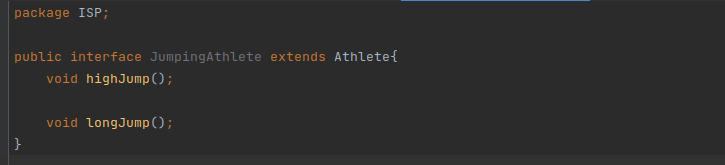


We will follow the interface segregation principle and refactor the original interface:



Then we will create two other interfaces — one for Jumping athletes and one for Swimming athletes.





And therefore Sachin will not have to implement actions that he is not capable of performing:

