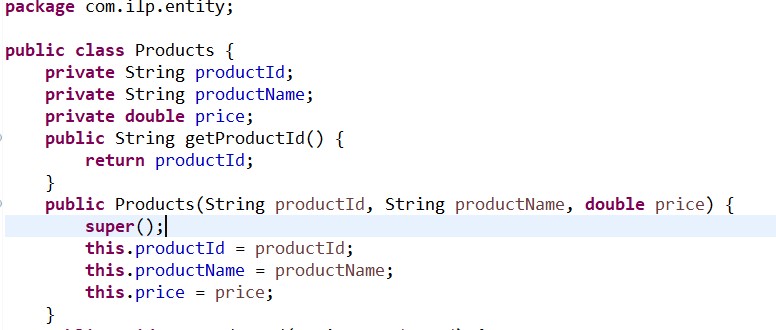
SOLID PRINCIPLES ASSIGNMENT

## **S — Single Responsibility**

A class should have a single responsibility

If a Class has many responsibilities, it increases the possibility of bugs because making changes to one of its responsibilities, could affect the other ones without you knowing.

* The **Products** class has a single responsibility, which is to represent product information.



* The **Stores** class represents a store and manages its basic details along with a list of products. It has a single responsibility.

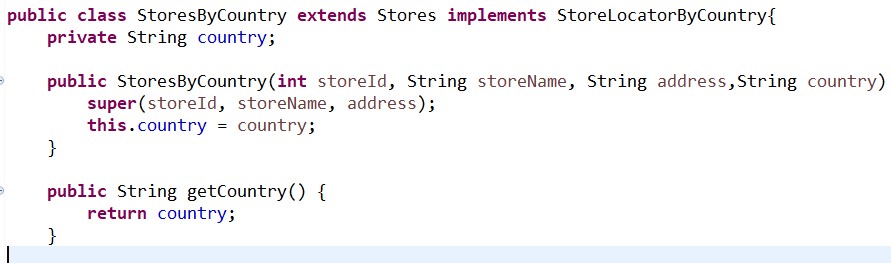
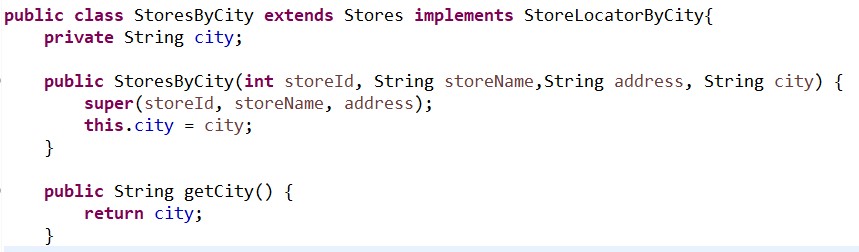


1. **O — Open-Closed**

Classes should be open for extension but closed for modification.

If you want the Class to perform more functions, the ideal approach is to add to the functions that already exist NOT change them.

* New types of stores can be added by extending existing abstract class stores.

Here we can say that instead of creating child classes of stores and implementing interfaces, if we are writing each type of stores using ”if” condition inside the Stores class and implementing the functions, then if we need to add another type of store and its respective function, we need to modify the Stores class, which can cause bugs in places and functions where Stores class is used.

1. **L** — **Liskov Substitution**

If S is a subtype of T, then objects of type T in a program may be replaced with objects of type S without altering any of the desirable properties of that program.

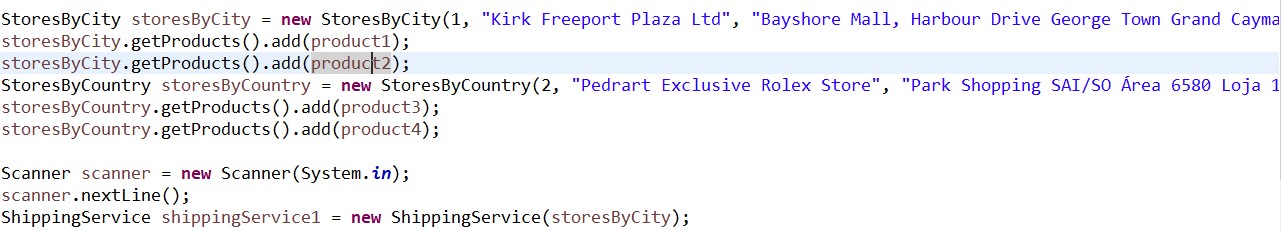
This principle aims to enforce consistency so that the parent Class or its child Class can be used in the same way without any errors.

* The StoresByCity and StoresByCountry classes extend the Stores class without breaking the functionality. This indicates adherence to the Liskov Substitution Principle.

Here in the ShippingService a Stores instance is given as the constructor, but in the utility class, a StoresByCity and StoresByCountry object is given as constructor when creating the ShippingService object.

So, we can use StoresByCity and StoresByCountry which are the child classes as the same way we use Stores class in ShippingService.

If the StoresByCity and StoresByCountry didn't extend the Stores class, then we would not have been able to replace the Stores class instances with objects of StoresByCity and StoresByCountry.



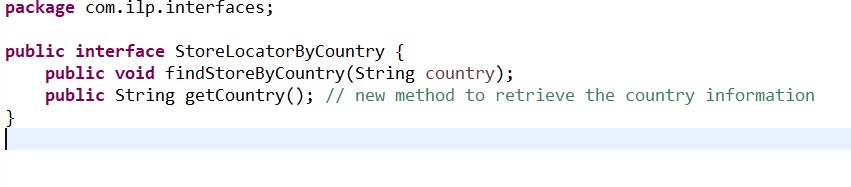
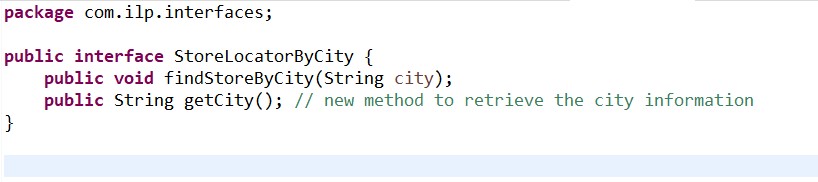
1. **I** — **Interface Segregation**

Clients should not be forced to depend on methods that they do not use.

This principle aims at splitting a set of actions into smaller sets so that a Class executes ONLY the set of actions it requires.

* The interfaces (Shipping, StoreDisplay, StoreLocatorByCity, StoreLocatorByCountry) seem to have specific and focused methods, adhering to the Interface Segregation Principle.

If interfaces were not correctly segregated and call methods were clubbed in a single interface then, the classes which don’t use certain functions would also have to implement the methods.



5. **D** — **Dependency Inversion**

High-level modules should not depend on low-level modules.

This principle aims at reducing the dependency of a high-level Class on the low-level Class by introducing an interface.

* The ShippingService class depends on the abstract class Stores rather than a concrete implementation of StoresByCity and StoresByCountry.This is in line with the Dependency Inversion Principle.

When the Shipping Service is called with any of the type of stores, the code works as we have created the instance of abstract class Stores in ShippingService.

If instead of creating the instance of Stores, we created an instance of StoresByCity it would have been difficult to call ShippingService with the instance of StoresByCountry, and vice versa.



