SOLID PRINCIPLES ASSIGNMENT

1. 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.

```java
package com.ilp.entity;

public class Products {
    private String productId;
    private String productName;
    private double price;
    public String getProductId() {
        return productId;
    }
    public Products(String productId, String productName, double price) {
        super();
        this.productId = productId;
        this.productName = productName;
        this.price = price;
    }
```

G

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

```java
package com.ilp.entity;

import java.util.ArrayList;

public abstract class Stores{
    private int storeId;
    private String storeName;
    private String address;
    ArrayList<Products> products = new ArrayList<Products>();

    public ArrayList<Products> getProducts() {
        return products;
    }
```

2. 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.

```
    public String getCountry() {
        return country;
    }
```

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.

3. 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.

```java
import com.ilp.entity.Products;

public class ShippingService implements StoreDisplay, Shipping {

    private Stores stores;

    public ShippingService(Stores stores) {
        this.stores = stores;
    }
```

```java
StoresByCity storesByCity = new StoresByCity(1, "Kirk Freeport Plaza Ltd", "Bayshore Mall, Harbour Drive George Town Grand Cayma
storesByCity.getProducts().add(product1);
storesByCity.getProducts().add(product2);
StoresByCountry storesByCountry = new StoresByCountry(2, "Pedrart Exclusive Rolex Store", "Park Shopping SAI/SO Área 6580 Loja 1
storesByCountry.getProducts().add(product3);
storesByCountry.getProducts().add(product4);

Scanner scanner = new Scanner(System.in);
scanner.nextLine();
ShippingService shippingService1 = new ShippingService(storesByCity);
```

4. 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.

```java
package com.ilp.interfaces;

public interface StoreLocatorByCity {
    public void findStoreByCity(String city);
    public String getCity(); // new method to retrieve the city information
}
```

```java
package com.ilp.interfaces;

public interface StoreLocatorByCountry {
    public void findStoreByCountry(String country);
    public String getCountry(); // new method to retrieve the country information
}
```

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.

```java
import com.ilp.entity.Products;

public class ShippingService implements StoreDisplay, Shipping {

    private Stores stores;

    public ShippingService(Stores stores) {
        this.stores = stores;
    }
```

```java
StoresByCity storesByCity = new StoresByCity(1, "Kirk Freeport Plaza Ltd",
storesByCity.getProducts().add(product1);
storesByCity.getProducts().add(product2);
StoresByCountry storesByCountry = new StoresByCountry(2, "Pedrart Exclusive
storesByCountry.getProducts().add(product3);
storesByCountry.getProducts().add(product4);

Scanner scanner = new Scanner(System.in);
scanner.nextLine();
ShippingService shippingService1 = new ShippingService(storesByCity);
shippingService1.displayStores();
scanner.nextLine();
storesByCity.findStoreByCity("New York");
scanner.nextLine();
shippingService1.localShipping();
scanner.nextLine();
ShippingService shippingService2 = new ShippingService(storesByCountry);
shippingService2.displayStores();
scanner.nextLine();
storesByCountry.findStoreByCountry("Canada");
scanner.nextLine();
shippingService2.internationalShipping();
scanner.nextLine();
```