NOTE : For all the below assignments
Create the Main method to test out the various objects
Create and throw the appropriate Custom exceptions if necessary

1.

***** SRP *****

Consider below class

```
public class CustomerServiceImpl{

    private CustomerRepository customerRepository; //repository uses array to store the saved customer

    public CustomerServiceImpl(CustomerRepository customerRepository) {
        this.customerRepository = customerRepository;
    }

    public Boolean saveCustomer(CustomerRequest request){
        Customer customer = new Customer();// This will need a Customer class to be created
        // set some value for customer domain class
        customerRepository.save(customer);

        NotificationObject notificationObject = new NotificationObject();
        // set some value for notification
        sendNotification(notificationObject);
        return true;
    }

    public boolean sendNotification(NotificationObject notificationObject){
        // Calling sms gateway for sms, email send, send push notification from here
        return true;
    }
}
```

The problem of this code here, is customer service that
should be responsible for handling customer-related operations.
But here, we are also handling notification-related tasks.
For new developers/contributors, it will be difficult to understand the codebase.
If your notifications-related tasks require modification
then you have to change customer service and it is a risky process.

During unit testing, it will difficult to test this class because of mixed logic here.

Modify the above use case to follow the SRP.
(HINT : Try creating a separate service for Notification)

2.
***** OCP *****
Consider the below class structures

```
public class Rectangle {

    private int width;
    private int height;

    // getter and setter methods...
}

public class Square {

    private int side;

    // getter and setter methods...
}

public class Circle {

    private int radious;

    // getter and setter methods...
}
```

Supposing the developer who is novice at understanding SOLID principles writes below code for drawing the shape.

```
public class ShapePrinterService {

    public void drawShape(Object shape) {

        if (shape instanceof Rectangle) {
            // Draw Rectangle here...
        } else if (shape instanceof Square) {
            // Draw Square here...
        } else if (shape instanceof Circle) {
            // Draw Circle here...
        }
    }
}
```

from the main class after building the object
Here it is drawing the shape.

```
Circle circle = new Circle();
circle.setRadius(5);
sharePrinterService.drawShape(circle);
```

This code has below problems
For a new shape like Polygon, you have to add a new if clause in drawShape method.
These new changes/implementations may create a bug
This type of code is very hard to debug for solving errors.

Modify the above class structures so that it conforms to OCP.
(HINT : Think in terms of using some Abstract super class)

3.
***** LSP *****
Consider the below class structures

```
public class Bird{
    public void fly(){
      System.out.println("Bird Flies High in the sky");
    }
}

public class Duck extends Bird{
  System.out.println("Duck Flies Not So High in the sky");
}

public class Crow extends Bird{
  System.out.println("Crow Flies Moderately High in the sky");
}
```

```java
public class Ostrich extends Bird{
  // For this class, if one calls fly() method then an exception would be thrown

}
```

This clearly violates the LSP
Modify the above code so that it conforms to LSP
(HINT : Try using multilevel inheritance so that the flying and non flying
birds follow different Hierarchy)

4.
***** ISP *****

```java
public interface IWorker {
    public void work();
    public void eat();
}

public class Human implements IWorker {

     @Override
     public void work() {
          // TODO Auto-generated method stub
          System.out.println("Human working");

     }

     @Override
     public void eat() {
          // TODO Auto-generated method stub
          System.out.println("Human eating");

     }

}

public class Robot implements IWorker {

     @Override
     public void work() {
            // TODO Auto-generated method stub
            System.out.println("Robot working");

      }

      @Override
      public void eat() {
            // TODO Auto-generated method stub
            throw new UnsupportedOperationException("cannot eat");

      }

}
```

This violates the ISP
Robot will throw an exception if eat method is called.

Modify the above code such that it conforms to ISP
(HINT : Try separating the work and eat functionalities)

5.
***** DIP *****
Consider the below classes

```
public class BackEndDeveloper {

    public void writeJava() {
        System.out.println("Excellent Java Coding...");
    }
}
public class FrontEndDeveloper {

    public void writeJavascript() {
        System.out.println("Excellent JavaScript Coding...");
    }
 }
 public class Project {

    private BackEndDeveloper backEndDeveloper = new BackEndDeveloper();
    private FrontEndDeveloper frontEndDeveloper = new FrontEndDeveloper();
    public void implement() {

        backEndDeveloper.writeJava();
        frontEndDeveloper.writeJavascript();
    }
 }
```

The Project class is a high level module and
it depends on low level modules such as BackEndDeveloper and FrontEndDeveloper.
So actually violating the first part of the dependency inversion principle.
Also by the inspecting the implement function of the Project.class
we realise that the methods writeJava and writeJavascript are methods
bound to the corresponding classes.
Regarding the project scope those are details since
in both cases they are forms of development.
Thus the second part of the dependency inversion principle is violated.

Modify the above code so that it conforms to DIP
(HINT : Think in terms of using the additional interface Developer
with a method develop(), The Project class will use this abstraction.)