SOLID Design Principles In Java

The SOLID principles of Object Oriented Design include following five principles:

- Single Responsibility Principle (SRP)
- Open Closed Design Principle
- Liskov Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Injection or Inversion principle

Single Responsibility Principle (SRP)

It is a SOLID design principle and represent “S” on the SOLID acronym. According to SRP a class should always handle single functionality or there should not be more than one reason for a class to change. It is used to achieve loose coupling between two different functionalities.

Let us understand the single responsibility principle by below example. In the below we have an Employee class which consists of its properties and a method which returns whether the promotion is due for that Employee or not.

```
public class Employee{
  private String empId;
  private String name;
  private string address;

  public boolean isPromotionDueThisYear(){
    //promotion logic
  }

  //Getters & Setters
}
```

The above class is not following single responsibility principle because Employee class should have the single responsibility of maintaining core attributes of an employee. Calculating whether the promotion is due for that Employee or not is not the responsibility of Employee class.

Better Solution:
We can create a new class which will contain the promotion calculation logic separately. This approach will follow the single responsibility principle because now both classes Employee and Promotions have single responsibilities.

```
public class Promotions{
  public boolean isPromotionDueThisYear(Employee emp){
    //promotion logic
  }
}

public class Employee{
  private String empId;
  private String name;
  private string address;
```

```
  //Getters & Setters
}
```

Open Closed Design Principle

The open closed design principle says that software entities like classes, modules, functions, etc. should be open for extension, but closed for modification. It is a SOLID design principle and represent “O” on the SOLID acronym.

A class is considered to be closed if changes are guaranteed to not happen. It can be used as a base class which can be extended by child classes.

A class is considered to be open if its functionality can be enhanced by its subclasses.

Let us understand the open closed design principle with below example. Consider a case when we need to calculate areas of various shapes. We start with creating a class for our first shape Rectangle.

```
public class Rectangle{
 public double length;
 public double width;
}
```

Next create a class to calculate area of this Rectangle which has a method calculateRectangleArea().

```
public class AreaCalculator{
  public double calculateRectangleArea(Rectangle rectangle){
    return rectangle.length *rectangle.width;
  }
}
```

Now we have circle as a new shape. So we create a Circle class.

```
public class Circle{
 public double radius;
}
```

Then we modify AreaCalculator class to add circle calculations through a new method calculateCircleArea().

```
public class AreaCalculator{
  public double calculateRectangleArea(Rectangle rectangle){
    return rectangle.length *rectangle.width;
  }
  public double calculateCircleArea(Circle circle){
    return (22/7)*circle.radius*circle.radius;
  }
}
```

Whenever a new requirement comes we have to modify our class so this design is not closed for modification.

Better Solution:
First we will create a base type shape which will be implemented by all shapes. It has an abstract method calculateArea(). Every sub class override this method and will provide own implementation. This design follow the open closed design principle as Shape class is open for extension and closed for modification.

```
public interface Shape{
  public double calculateArea();
}

public class Rectangle implements Shape{
  double length;
  double width;
  public double calculateArea(){
    return length * width;
  }
}

public class Circle implements Shape{
  public double radius;
  public double calculateArea(){
    return (22/7)*radius*radius;
  }
}
```

Liskov Substitution Principle (LSP)

It is a SOLID design principle and represent “L” on the SOLID acronym. According to LSP subtypes must be substitutable for supertype. The main concept behind LSP SOLID design principle is that derived class or subclass must enhance functionality not reduce.

For example we have Animal class with a MakeNoise() method then any subclass of Animal should implement makeNoise().

Interface Segregation Principle (ISP)

It is a SOLID design principle and represent “I” on the SOLID acronym. According to ISP clients should not be forced to implement unnecessary methods which they will not use. To achieve the ISP SOLID design principle we favor many, smaller, client-specific interfaces over one larger interface.

Let us understand the interface segregation principle by below example. In below example we have one interface which have two methods to generate reports in different formats generateExcel()and generatePdf().

```
public interface GenerateReport{
  public void generateExcel();
  public void generatePDF();
}
```

Now consider a case client Test wants to use this interface but want to use reports only in PDF format and not in excel. With the above design he cannot achieve it because we are forcing him to implement both methods. So this design is not following the interface segregation principle.

Better Solution:
We can break the GenerateReport interface into two small interfaces which contains separate methods and client can use the desire method.

Dependency Injection or Inversion principle

It is a SOLID design principle and represent “D” on the SOLID acronym. According to dependency inversion principle code should depends upon abstractions rather than upon concrete details. We

should design our software in such a way that various modules can be separated from each other using an abstract layer to bind them together. BeanFactory in spring framework represents the classical use of this principle. All spring framework modules are provided as separate components which can work together by simply injected dependencies in other module.

Injection:

Injection is a process of passing the dependency to a dependent object.

Dependency Injection (DI):

Dependency Injection (DI) is a design pattern that implements inversion of control principle for resolving dependencies. It allows a programmer to remove hard coded dependencies so that the application becomes loosely coupled and extendable.

Let us discuss object dependency with below example:

```
public class Student {
   private Address address;

   public Student() {
      address = new Address();
   }
}
```

In above example Student class requires an Address object and it is responsible for initializing and using the Address object. If Address class is changed in future then we have to make changes in Student class also. This approach makes tight coupling between Student and Address objects. We can resolve this problem using dependency injection design pattern. i.e. Address object will be implemented independently and will be provided to Student when Student is instantiated by using constructor-based or setter-based dependency injection.

You can read following topics for dependency injection:

Spring dependency injection tutorial.
Spring constructor based injection tutorial.
Constructor injection type ambiguities tutorial.
Setter based dependency injection tutorial.
Spring dependency injection collections tutorial.