https://www.javatpoint.com/solid-principles-java
https://www.edureka.co/blog/solid-principles-in-java/

What is the meaning of S.O.L.I.D?

Solid represents five principles of java which are:

- S: Single responsibility principle
- O: Open-closed principle
- L: Liskov substitution principle
- I: Interface segregation principle
- D: Dependency inversion principle

In this blog, we will discuss all the five SOLID principles of Java in detail.

Single Responsibility Principle in Java

Robert C. Martin describes it as one class should have only one and only responsibility.

According to the single responsibility principle, there should be only one reason due to which a class has to be changed. It means that a class should have one task to do. This principle is often termed as subjective.

The principle can be well understood with an example. Imagine there is a class which performs following operations.

- Connected to a database
- Read some data from database tables
- Finally, write it to a file.

Have you imagined the scenario? Here the class has multiple reasons to change, and few of them are the modification of file output, new data base adoption. When we are talking about single principle responsibility, we would say, there are too many reasons for the class to change; hence, it doesn't fit properly in the single responsibility principle.

Why is that this Principle is Required?

When the Single Responsibility Principle is followed, testing is easier. With a single responsibility, the class will have fewer test cases. Less functionality also means fewer dependencies to other classes. It leads to better code organization since smaller and well-purposed classes are easier to search.

An example to clarify this principle:

Suppose you are asked to implement a UserSetting service wherein the user can change the settings but before that the user has to be authenticated. One way to implement this would be:

```
public class UserSettingService
{
public void changeEmail(User user)
{
if(checkAccess(user))
{
//Grant option to change
}
}
public boolean checkAccess(User user)
{
//Verify if the user is valid.
}
}
```

All looks good until you would want to reuse the checkAccess code at some other place OR you want to make changes to the way checkAccess is being done. In all 2 cases you would end up changing the same class and in the first case you would have to use UserSettingService to check for access as well.

One way to correct this is to decompose the UserSettingService into UserSettingService and SecurityService. And move the checkAccess code into SecurityService.

```
public class UserSettingService
{
   public void changeEmail(User user)
  {
    if(SecurityService.checkAccess(user))
    {
//Grant option to change
    }
 }

}

public class SecurityService
{
public static boolean checkAccess(User user)
{
//check the access.
}
}
```

Open Closed Principle in Java

Robert C. Martin describes it as Software components should be open for extension, but closed for modification.

To be precise, according to this principle, a class should be written in such a manner that it performs its job flawlessly without the assumption that people in the future will simply come and change it. Hence, the class should remain closed for modification, but it should have the option to get extended.

An excellent example of open-closed principle can be understood with the help of browsers. Do you remember installing extensions in your chrome browser?

Basic function of the chrome browser is to surf different sites. Do you want to check grammar when you are writing an email using chrome browser?If yes, you can simply use Grammarly extension, it provides you grammar check on the content.

This mechanism where you are adding things for increasing the functionality of the browser is an extension. Hence, the browser is a perfect example of functionality that is

open for extension but is closed for modification. In simple words, you can enhance the functionality by adding/installing plugins on your browser, but cannot build anything new.

Why is that this principle is required?

OCP is important since classes may come to us through third-party libraries. We should be able to extend those classes without worrying if those base classes can support our extensions. But inheritance may lead to subclasses which depend on base class implementation. To avoid this, use of interfaces is recommended. This additional abstraction leads to loose coupling.

Lets say we need to calculate areas of various shapes. We start with creating a class for our first shape Rectangle which has 2 attributes length & width.

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

Next we create a class to calculate the area of this Rectangle which has a method calculateRectangleArea which takes the Rectangle as an input parameter and calculates its area.

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

So far so good. Now let's say we get our second shape circle. So we promptly create a new class Circle with a single attribute radius.

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

Then we modify Areacalculator class to add circle calculations through a new method calculateCircleaArea()

```
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;
  }
}
```

However, note that there were flaws in the way we designed our solution above.

Lets say we have a new shape pentagon. In that case, we will again end up modifying the AreaCalculator class. As the types of shapes grows this becomes messier as AreaCalculator keeps on changing and any consumers of this class will have to keep on updating their libraries which contain AreaCalculator. As a result, AreaCalculator class will not be baselined(finalized) with surety as every time a new shape comes it will be modified. So, this design is not closed for modification.

AreaCalculator will need to keep on adding their computation logic in newer methods. We are not really expanding the scope of shapes; rather we are simply doing piece-meal(bit-by-bit) solution for every shape that is added.

Modification of above design to comply with opened/closed principle:

Let us now see a more elegant design which solves the flaws in the above design by adhering to the Open/Closed Principle. We will first of all make the design extensible. For this we need to first define a base type Shape and have Circle & Rectangle implement Shape interface.

```
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;
}
}
```

There is a base interface Shape. All shapes now implement the base interface Shape. Shape interface has an abstract method calculateArea(). Both circle & rectangle provide their own overridden implementation of calculateArea() method using their own attributes.
We have brought in a degree of extensibility as shapes are now an instance of Shape interfaces. This allows us to use Shape instead of individual classes
The last point above mentioned consumer of these shapes. In our case, the consumer will be the AreaCalculator class which would now look like this.

```
public class AreaCalculator
{
public double calculateShapeArea(Shape shape)
{
return shape.calculateArea();
}
}
```

This AreaCalculator class now fully removes our design flaws noted above and gives a clean solution which adheres to the Open-Closed Principle. Let's move on with other SOLID Principles in Java

Liskov Substitution Principle in Java

The Liskov Substitution Principle (LSP) was introduced by **Barbara Liskov**. It applies to inheritance in such a way that the **derived classes must be completely substitutable for**

their base classes. In other words, if class A is a subtype of class B, then we should be able to replace B with A without interrupting the behavior of the program.

Why is that this principle required?

This avoids misusing inheritance. It helps us conform to the “is-a” relationship.We can also say that subclasses must fulfill a contract defined by the base class. In this sense, it’s related to Design by Contract that was first described by Bertrand Meyer. For example, it’s tempting to say that a circle is a type of ellipse but circles don’t have two foci or major/minor axes.

The LSP is popularly explained using the square and rectangle example. if we assume an ISA relationship between Square and Rectangle. Thus, we call “Square is a Rectangle.” The code below represents the relationship.

```
public class Rectangle
{
private int length;
private int breadth;
public int getLength()
{
return length;
}
public void setLength(int length)
{
this.length = length;
}
public int getBreadth()
{
return breadth;
}
public void setBreadth(int breadth)
{
this.breadth = breadth;
}
public int getArea()
{
return this.length * this.breadth;
}
}
```

Below is the code for Square. Note that Square extends Rectangle.

```
public class Square extends Rectangle
{

public void setBreadth(int breadth)
{
super.setBreadth(breadth);
super.setLength(breadth);
}

public void setLength(int length)
{
super.setLength(length);
super.setBreadth(length);
}
}
```

In this case, we try to establish an ISA relationship between Square and Rectangle such that calling “Square is a Rectangle” in the below code would start behaving unexpectedly if an instance of Square is passed. An assertion error will be thrown in the case of checking for “Area” and checking for “Breadth,” although the program will terminate as the assertion error is thrown due to the failure of the Area check.

```
public class LSPDemo
{
public void calculateArea(Rectangle r)
{
r.setBreadth(2);
r.setLength(3);
assert r.getArea() == 6 : printError("area", r);
assert r.getLength() == 3 : printError("length", r);
assert r.getBreadth() == 2 : printError("breadth", r);
}
private String printError(String errorIdentifer, Rectangle r)
{
return "Unexpected value of " + errorIdentifer + " for instance of " +
}
public static void main(String[] args)
{
LSPDemo lsp = new LSPDemo();
// An instance of Rectangle is passed
lsp.calculateArea(new Rectangle());
// An instance of Square is passed
lsp.calculateArea(new Square());
}
}
```

The class demonstrates the Liskov Substitution Principle (LSP) As per the principle, the functions that use references to the base classes must be able to use objects of derived class without knowing it.

Thus, in the example shown below, the function calculateArea which uses the reference of “Rectangle” should be able to use the objects of derived class such as Square and fulfill the requirement posed by Rectangle definition. One should note that as per the definition of Rectangle, following must always hold true given the data below:

1. Length must always be equal to the length passed as the input to method, setLength
2. Breadth must always be equal to the breadth passed as input to method, setBreadth
3. Area must always be equal to product of length and breadth

In case, we try to establish ISA relationship between Square and Rectangle such that we call “Square is a Rectangle”, above code would start behaving unexpectedly if an

instance of Square is passed Assertion error will be thrown in case of check for area and check for breadth, although the program will terminate as the assertion error is thrown due to failure of Area check.

The Square class does not need methods like setBreadth or setLength. The LSPDemo class would need to know the details of derived classes of Rectangle (such as Square) to code appropriately to avoid throwing error. The change in the existing code breaks the open-closed principle in the first place.

The above classes violated the Liskov substitution principle because the Square class has extra constraints i.e. length and breadth that must be the same. Therefore, the Rectangle class (base class) cannot be replaced by Square class (derived class).

Hence, substituting the class Rectangle with Square class may result in unexpected behavior.

Interface Segregation Principle

Robert C. Martin describes it as clients should not be forced to implement unnecessary methods which they will not use.

According to Interface segregation principle a client, no matter what should never be forced to implement an interface that it does not use or the client should never be obliged to depend on any method, which is not used by them.So basically, the interface segregation principles as you prefer the interfaces, which are small but client specific instead of monolithic and bigger interface.In short, it would be bad for you to force the client to depend on a certain thing, which they don’t need.

For example, a single logging interface for writing and reading logs is useful for a database but not for a console. Reading logs make no sense for a console logger. Moving on with this SOLID Principles in Java article.

Why is that this principle required?

Let us say that there is a Restaurant interface which contains methods for accepting orders from online customers, dial-in or telephone customers and walk-in customers. It also contains methods for handling online payments (for online customers) and in-person payments (for walk-in customers as well as telephone customers when their order is delivered at home).

Now let us create a Java Interface for Restaurant and name it as RestaurantInterface.java.

```
public interface RestaurantInterface
{
  public void acceptOnlineOrder();
  public void takeTelephoneOrder();
  public void payOnline();
  public void walkInCustomerOrder();
  public void payInPerson();
}
```

There are 5 methods defined in RestaurantInterface which are for accepting online order, taking telephonic order, accepting orders from a walk-in customer, accepting online payment and accepting payment in person.

Let us start by implementing the RestaurantInterface for online customers as OnlineClientImpl.java

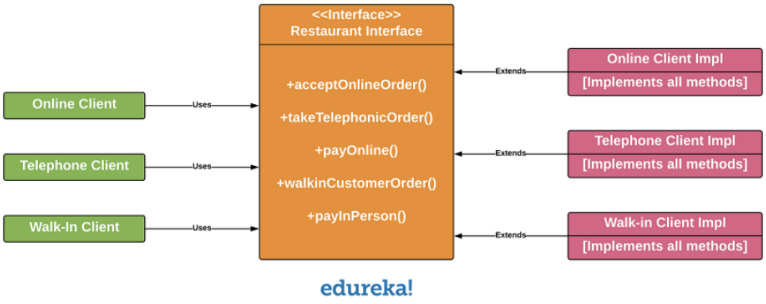
```
public class OnlineClientImpl implements RestaurantInterface
{
public void acceptOnlineOrder()
{
//logic for placing online order
}
public void takeTelephoneOrder()
{
//Not Applicable for Online Order
throw new UnsupportedOperationException();
}
public void payOnline()
{
//logic for paying online
}
public void walkInCustomerOrder()
{
//Not Applicable for Online Order
throw new UnsupportedOperationException();
}
public void payInPerson() {
//Not Applicable for Online Order
throw new UnsupportedOperationException();
}
}
```

- Since the above code(OnlineClientImpl.java) is for online orders, throw UnsupportedOperationException.
- Online, telephonic and walk-in clients use the RestaurantInterface implementation specific to each of them.
- The implementation classes for Telephonic client and Walk-in client will have unsupported methods.
- Since the 5 methods are part of the RestaurantInterface, the implementation classes have to implement all 5 of them.

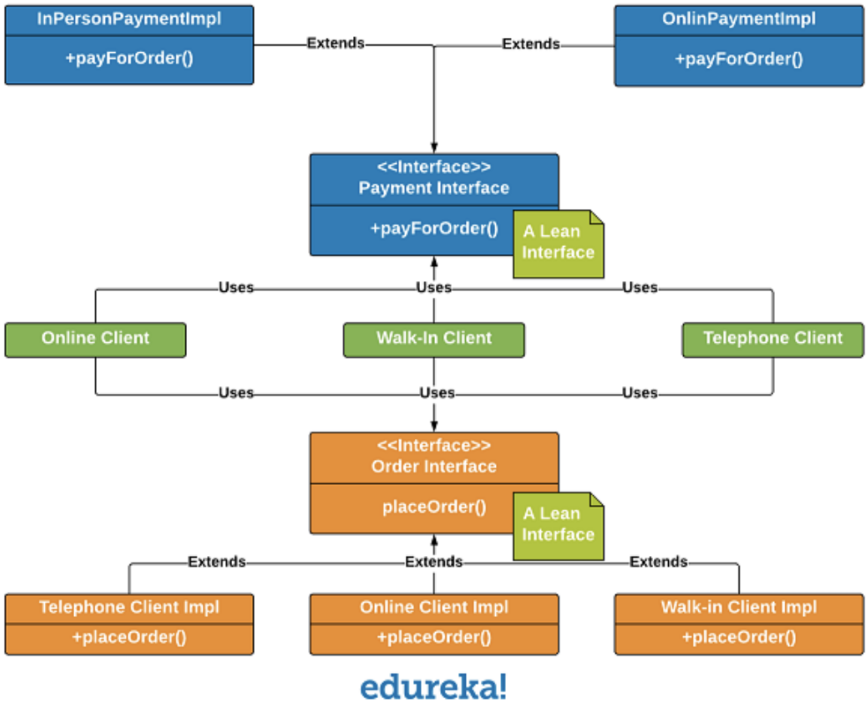
- The methods that each of the implementation classes throw UnsupportedOperationException. As you can clearly see – implementing all methods is inefficient.
- Any change in any of the methods of the RestaurantInterface will be propagated to all implementation classes. Maintenance of code then starts becoming really cumbersome and regression effects of changes will keep increasing.
- RestaurantInterface.java breaks Single Responsibility Principle because the logic for payments as well as that for order placement is grouped together in a single interface.

To overcome the above mentioned problems , we apply Interface Segregation Principle to refactor the above design.

1. Separate out payment and order placement functionalities into two separate lean interfaces,PaymentInterface.java and OrderInterface.java.
2. Each of the clients use one implementation each of PaymentInterface and OrderInterface. For example – OnlineClient.java uses OnlinePaymentImpl and OnlineOrderImpl and so on.
3. Single Responsibility Principle is now attached as Payment interface(PaymentInterface.java) and Ordering interface(OrderInterface).
4. Change in any one of the order or payment interfaces does not affect the other. They are independent now.There will be no need to do any dummy implementation or throw an UnsupportedOperationException as each interface has only methods it will always use.



After applying ISP



Dependency Inversion Principle

The principle states that we must use abstraction (abstract classes and interfaces) instead of concrete implementations. High-level modules should not depend on the low-level module but both should depend on the abstraction. Because the abstraction does not depend on detail but the detail depends on abstraction. It decouples the software. Let's understand the principle through an example.

It is worth, if we have not keyboard and mouse to work on Windows. To solve this problem, we create a constructor of the class and add the instances of the keyboard and monitor. After adding the instances, the class looks like the following:

```
public class WindowsMachine
{
public final keyboard;
public final monitor;
public WindowsMachine()
{
monitor = new monitor();  //instance of monitor class
keyboard = new keyboard(); //instance of keyboard class
}
}
```

Now we can work on the Windows machine with the help of a keyboard and mouse. But we still face the problem. Because we have tightly coupled the three classes together by using the new keyword. It is hard o test the class windows machine.

To make the code loosely coupled, we decouple the WindowsMachine from the keyboard by using the Keyboard interface and this keyword.

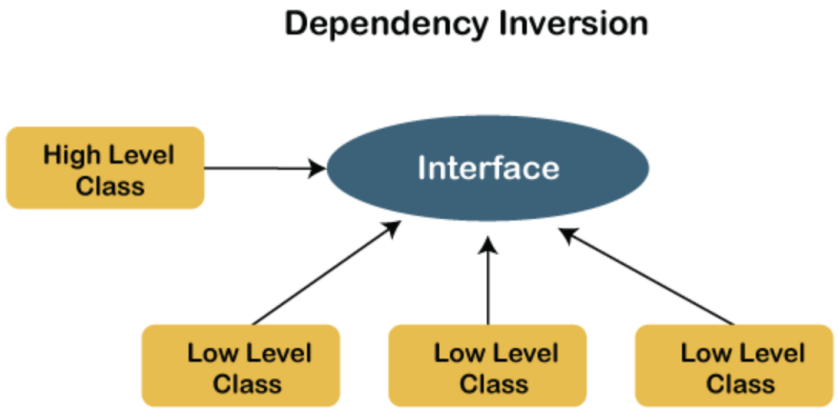
Keyboard.java

```
public interface Keyboard
{
//functionality
}
```

WindowsMachine.java

```
public class WindowsMachine
{
private final Keyboard keyboard;
private final Monitor monitor;
public WindowsMachine(Keyboard keyboard, Monitor monitor)
{
this.keyboard = keyboard;
this.monitor = monitor;
}
}
```

In the above code, we have used the dependency injection to add the keyboard dependency in the WindowsMachine class. Therefore, we have decoupled the classes.



Why should we use SOLID principles?

- It reduces the dependencies so that a block of code can be changed without affecting the other code blocks.
- The principles intended to make design easier, understandable.
- By using the principles, the system is maintainable, testable, scalable, and reusable.
- It avoids the bad design of the software.

Next time when you design software, keeps these five principles in mind. By applying these principles, the code will be much more clear, testable, and expendable.