**1. Single Responsibility Principle (SRP)**

As the name indicates, it states that all classes and modules should have only 1 well-defined responsibility. As per **Robert C Martin**,

A class should have one, and only one reason to change.

This means when we design our classes, we need to ensure that our class is responsible only for 1 task or functionality and when there is a change in that task/functionality, only then, that class should change.

**Benefits of Single Responsibility Principle**

* When an application has multiple classes, each of them following this principle, then the application becomes more maintainable, easier to understand.
* The code quality of the application is better, thereby having fewer defects.
* Onboarding new members is easy, and they can start contributing much faster.
* Testing and writing test cases is much simpler

Let’s look at one example to understand this concept better. Consider a food delivery application that takes food orders, calculates the bill, and delivers it to customers. We can have 1 separate class for each of the tasks to be performed, and then the main class can just invoke those classes to get these actions done one after the other.

**Code:**

**import java.io.\*;**

**import java.util.\*;**

**public class main {**

**public static void main(String[] args)**

**{**

**Customer customer1 = new Customer();**

**customer1.setName("Yamika");**

**customer1.setAddress("Pune");**

**Order order1 = new Order();**

**order1.setItemName("Pizza");**

**order1.setQuantity(2);**

**order1.setCustomer(customer1);**

**order1.prepareOrder();**

**Bill Calculation bill Calculation**

**= new BillCalculation(order1);**

**bill Calculation.calculate Bill();**

**DeliveryApp deliveryApp = new DeliveryApp(order1);**

**deliveryApp.delivery();**

**}**

**}**

**public class Customer {**

**private String name;**

**private String address;**

**public String getName() { return name; }**

**public void setName(String name) { this.name = name; }**

**public String getAddress() { return address; }**

**public void setAddress(String address)**

**{**

**this.address = address;**

**}**

**}**

**import java.util.Random;**

**public class Order {**

**private Customer customer;**

**private String orderId;**

**private String itemName;**

**private int quantity;**

**private int totalBillAmt;**

**public Customer getCustomer() { return customer; }**

**public void setCustomer(Customer customer)**

**{**

**this.customer = customer;**

**}**

**public String getOrderId() { return orderId; }**

**public void setOrderId(String orderId)**

**{**

**Random random = new Random();**

**this.orderId = orderId + "-" + random.nextInt(500);**

**}**

**public String getItemName() { return itemName; }**

**public void setItemName(String itemName)**

**{**

**this.itemName = itemName;**

**setOrderId(itemName);**

**}**

**public int getQuantity() { return quantity; }**

**public void setQuantity(int quantity)**

**{**

**this.quantity = quantity;**

**}**

**public int getTotal BellAmy() { return totalBiscuit; }**

**public void setTotalBillAmt(int totalBillAmt)**

**{**

**this.totalBillAmt = totalBillAmt;**

**}**

**public void prepareOrder()**

**{**

**System.*out*.println("Preparing order for customer -"**

**+ this.getCustomer().getName()**

**+ " who has ordered "**

**+ this.getItemName());**

**}**

**}**

**import java.util.Random;**

**public class BillCalculation {**

**private Order order;**

**public BillCalculation(Order order)**

**{**

**this.order = order;**

**}**

**public void calculateBill()**

**{**

**Random rand = new Random();**

**int totalAmt**

**= rand.nextInt(200) \* this.order.getQuantity();**

**this.order.setTotalBillAmt(totalAmt);**

**System.*out*.println("Order with order id "**

**+ this.order.getOrderId()**

**+ " has a total bill amount of "**

**+ this.order.getTotalBillAmt());**

**}**

**}**

**public class DeliveryApp {**

**private Order order;**

**publix DeliveryApp(Order order) { this.order = order; }**

**public void delivery()**

**{**

**System.*out*.println("Delivering the order");**

**System.*out*.println(**

**"Order with order id as "**

**+ this.order.getOrderId()**

**+ " being delivered to "**

**+ this.order.getCustomer().getName());**

**System.*out*.println(**

**"Order is to be delivered to: "**

**+ this.order.getCustomer().getAddress());**

**}**

**}**

Here is the output :

Preparing order for customer -Yamika who has ordered Pizza

Order with order id Pizza-292 has a total bill amount of 100

Delivering the order

Order with order id as Pizza-292 being delivered to Yakima

Order is to be delivered to: Pune

# **2.Open Closed Principle**

The principle states that software entities like class, modules, functions, etc.; should be able to extend a class behavior without modifying it. This principle separates the existing code from modified mode to provide better stability, maintainability and minimizes the changes in the code. In a nutshell, the developer must need to change only a specific part of the code (a class or a function) every time a requirement changes.

**Implementation:**

Program to calculate the volume in which consider the task of building an application that calculates the volumes of all the geometric objects.

* The Cuboid class stores dimensions of the cuboid
* Later on, the Application class calculates the total volume of the geometric objects–which are only cuboids currently.
* Run class helps to run the whole program.

CODE:

public class Cuboid {

public double length;

public double breadth;

public double height;

}

public class Application {

public double get\_total\_volume(Cuboid[] geo\_objects)

{

double vol\_sum = 0;

for (Cuboid geo\_obj : geo\_objects) {

vol\_sum += geo\_obj.length \* geo\_obj.breadth

\* geo\_obj.height;

}

return vol\_sum;

}

}

public class main {

public static void main(String args[])

{

Cuboid cb1 = new Cuboid();

cb1.length = 5;

cb1.breadth = 10;

cb1.height = 15;

Cuboid cb2 = new Cuboid();

cb2.length = 2;

cb2.breadth = 4;

cb2.height = 6;

Cuboid cb3 = new Cuboid();

cb3.length = 3;

cb3.breadth = 12;

cb3.height = 15;

Cuboid[] c\_arr = new Cuboid[3];

c\_arr[0] = cb1;

c\_arr[1] = cb2;

c\_arr[2] = cb3;

Application app = new Application();

double vol = app.get\_total\_volume(c\_arr);

System.*out*.println("The total volume is " + vol);

}

}

**Output**

The total volume is 1338.0

# **3.The Liskov Substitution Principle**

As per the LSP, *functions that use references to base classes must be able to use objects of the derived class without knowing it*. In simple words, derived classes must be substitutable for the base class. To illustrate the LSP, let’s take an example of rectangles and squares. One tends to establish the ISA relationship, thus, you can say that a square is a rectangle.

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

**CODE:**

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;

}

}

public class Square extends Rectangle {

@Override

public void setBreadth(int breadth) {

super.setBreadth(breadth);

super.setLength(breadth);

}

@Override

public void setLength(int length) {

super.setLength(length);

super.setBreadth(length);

}

}

The code below represents the function which has Rectangle as an argument. 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.

public class main {

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 error Identifier, Rectangle r) {

return "Unexpected value of " + error Identifier + " for instance of " + r.getClass().getName();

}

public static void main(String[] args) {

main lsp = new main();

lsp.calculateArea(new Rectangle());

lsp.calculateArea(new Square());

}

}

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.

Given the above code, what is the problem with the Square-Rectangle ISA relationship?

* The Square class does not need methods like setBreadth or setLength, as the sides of a square are equal. This is wasteful. Imagine hundreds of thousands of Square objects.
* The LSPDemo class would need to know the details of derived classes of Rectangle (such as Square) to code appropriately to avoid throwing errors. The change in the existing code to take care of the derived class breaks the open-closed principle in the first place.

## What Code Quality Characteristics Is Represented by LSP?

The following are some of the code quality characteristics that are represented by the Liskov Substitution Principle.

* It is only when derived types are completely substitutable for their base types that functions that use those base types can be reused with impunity, and the derived types can be changed with impunity.
* The LSP is also, at times, termed as “Design by Contract.” Using this scheme, methods of classes declare preconditions and postconditions. The pre-conditions must be true in order for the method to execute. Upon completion, the method guarantees that the post-condition will be true.
* Design by Contract does influence the declaration of “throws” exceptions, as well as the throwing of runtime exceptions and try/catch in general.