**SOLID Principles:**

These Principles are used in OOD(Object Oriented Design) which helps us to keep the code

* modular
* maintainable
* less complex
* easy to understand
* avoid duplication

**Acronym of SOLID**

S - Single Responsibility Principle

O - Open/Closed Principle

L - Liskov Substitution Principle

I - Interface Segregation Principle

D – Dependency Inversion Principle

**Single Responsibility Principle:**

* This principle states that **each class should have** **one responsibility, one single purpose**. This means that a class will do only one job, which leads us to conclude it should have **only one reason to change**
* Any Method/Function/Class which we create should have one responsibility and should only have one reason to change

Advantages:

* Adding multiple functionalities to single class can lead to complexity and hard to maintain
* Changing one functionality can effect other functionalities because they exist in a single class

**Examples:**

**Example 1:** We need to develop a Feature where we can manipulate a given text in different ways and Print the text in different ways

**Solutions:**  This feature can be developed in many ways, mentioning some below:

Approach 1: Developing both the features in a single place

Approach 2: Break Down the Feature into multiple parts where each part does one job

* One Class does the job of Text Manipulating
* One Cass does the job of Print the text

***Approach 2 will follow Single Responsibility principle***

**Example 2:** We need to develop a Feature where we need to add some Invoice and delete the invoices, generate reports and email reports

**Solutions:** This feature can be developed in many ways, mentioning some below:

Approach 1: Developing both the features in a single place

Approach 2: Break Down the Feature into multiple parts where each part does one job

* One Class for add and remove invoice
* One Class for generating Report
* Once Class for Sending email of report

***Approach 2 will follow Single Responsibility principle***

**Example 3: Feature to develop to compile a text and print the text**

**Solution:**

consider a module that compiles and prints a report. Imagine such a module can be changed for two reasons. First, the content of the report could change. Second, the format of the report could change. These two things change for different causes. The single responsibility principle says that these two aspects of the problem are really two separate [responsibilities](https://en.wikipedia.org/wiki/Interface_(computing)), and should, therefore, be in separate classes or modules. It would be a bad design to [couple](https://en.wikipedia.org/wiki/Coupling_(computer_programming)) two things that change for different reasons at different times.

The reason it is important to keep a class focused on a single concern is that it makes the class more robust. Continuing with the foregoing example, if there is a change to the report compilation process, there is a greater danger that the printing code will break if it is part of the same class.

**Example 4:**

**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.**

import java.io.\*;

import java.util.\*;

class GFG {

public static void main(String[] args)

{

Customer customer1 = new Customer();

customer1.setName("John");

customer1.setAddress("Pune");

Order order1 = new Order();

order1.setItemName("Pizza");

order1.setQuantity(2);

order1.setCustomer(customer1);

order1.prepareOrder();

BillCalculation billCalculation

= new BillCalculation(order1);

billCalculation.calculateBill();

DeliveryApp deliveryApp = new DeliveryApp(order1);

deliveryApp.delivery();

}

}

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;

}

}

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 getTotalBillAmt() { return totalBillAmt; }

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());

}

}

class BillCalculation {

private Order order;

public BillCalculation(Order order)

{

this.order = order;

}

public void calculateBill()

{

/\* In the real world, we would want a kind of lookup

functionality implemented here where we look for

the price of each item included in the order, add

them up and add taxes, delivery charges, etc on

top to reach the total price. We will simulate

this behaviour here, by generating a random number

for total price.

\*/

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());

}

}

class DeliveryApp {

private Order order;

public DeliveryApp(Order order) { this.order = order; }

public void delivery()

{

// Here, we would want to interface with another

// system which actually assigns the task of

// delivery to different persons

// based on location, etc.

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());

}

}