NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES

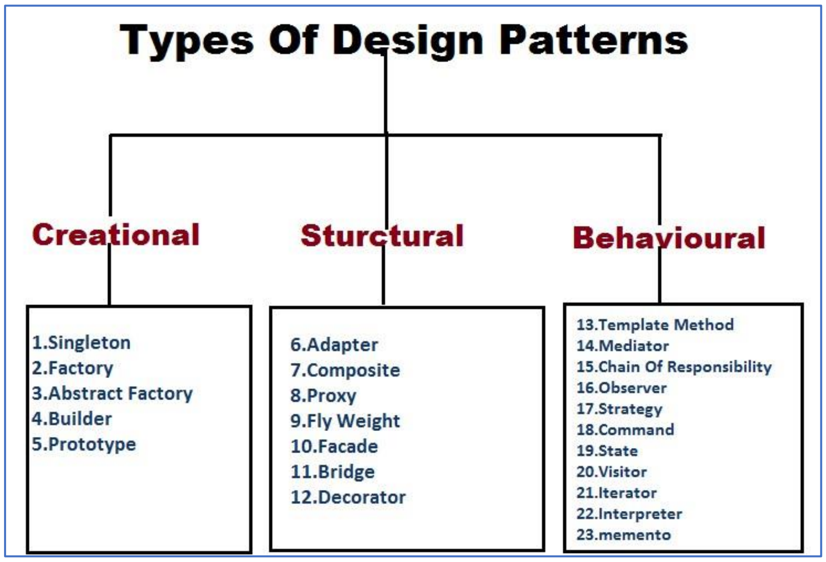
SE-220 – Software Design & Architecture Lab

LAB Instructors: Sobia Iftikhar “Sobia.iftikhar@nu.edu.pk”

Lab 12

- Singleton pattern
- Faced Pattern
- Example
- Exercise

Design patterns in java are best practices which are used to resolve some known issues. Design patterns can be divided into 3 different types. Here we have listed down some of the widely used design patterns in Java.



Singleton Design Pattern

Intent

- Ensure a class has only one instance, and provide a global point of access to it.
- Encapsulated "just-in-time initialization" or "initialization on first use".

Problem

- Application needs one, and only one, instance of an object. Additionally, lazy initialization and global access are necessary.

Initialization Types:

1. Lazy Initialization
2. Early Initialization

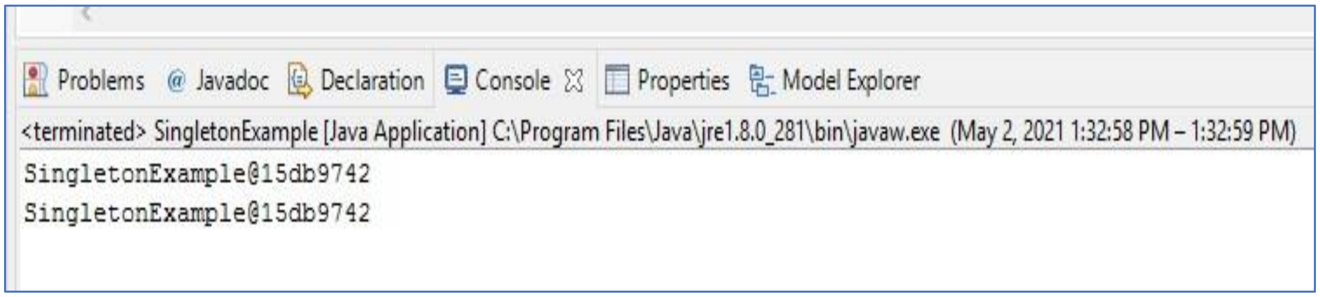
Example-01: Early Initialization

This is the simplest method of creating a singleton class. In this, object of class is created when it is loaded to the memory by JVM. It is done by assigning the reference an instance directly.

It can be used when program will always use instance of this class, or the cost of creating the instance is not too large in terms of resources and time.

```java
public class SingletonExample {

      private static SingletonExample instance = new SingletonExample();

        private SingletonExample(){}

        public static SingletonExample getInstance() {
          return instance;
        }

      public static void main(String[] args) {
            // TODO Auto-generated method stub
            SingletonExample instance = SingletonExample.getInstance();

                System.out.println(instance);
            SingletonExample instance1 = SingletonExample.getInstance();

                System.out.println(instance1);

      }
}
```

Output:



```
<terminated> SingletonExample [Java Application] C:\Program Files\Java\jre1.8.0_281\bin\javaw.exe (May 2, 2021 1:32:58 PM – 1:32:59 PM)
SingletonExample@15db9742
SingletonExample@15db9742
```

Example-02: Lazy Initialization

In this method, object is created only if it is needed. This may prevent resource wastage. An implementation of getInstance() method is required which return the instance. There is a null check that if object is not created then create, otherwise return previously created. To make sure that class cannot be instantiated in any other way, constructor is made final. As object is created with in a method, it ensures that object will not be created until and unless it is required.

Instance is kept private so that no one can access it directly.

It can be used in a single threaded environment because multiple threads can break singleton property because they can access get instance method simultaneously and create multiple objects.

```
//Java Code to create singleton class
// With Lazy initialization
      public class GFG
      {
  // private instance, so that it can be
  // accessed by only by getInstance() method
  private static GFG instance;

  private GFG()
  {
    // private constructor
  }

  //method to return instance of class
  public static GFG getInstance()
  {
    if (instance == null)
    {
      // if instance is null, initialize
      instance = new GFG();
    }
    return instance;
  }
}
a
```

Example-03

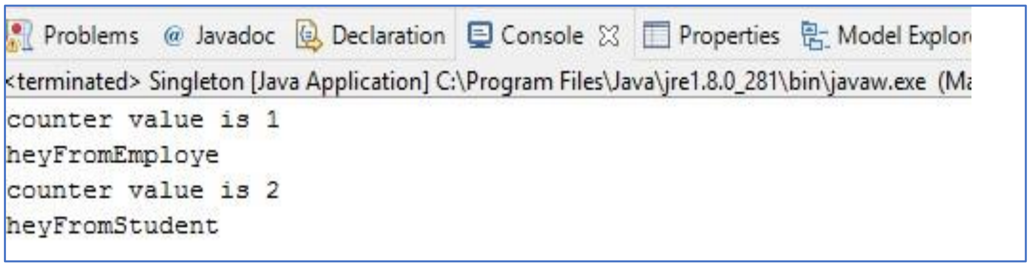
Following code create two separate instance for same methods. This is not technically right to run a same method with two different object. To reduce a memory overhead we have to work with one instance to call same method at different time. Using Singleton.

The Singleton's purpose is to control object creation, limiting the number to one but allowing the flexibility to create more objects if the situation changes. Since there is only one Singleton instance, any instance fields of a Singleton will occur only once per class, just like static fields.

```
public class Singleton {
//
      private static int counter=0;
        private Singleton(){
              counter++;
              System.out.println("counter value is "+counter);

        }
        public static void getInstance(String Message) {
              System.out.println(Message);
}
      public static void main(String[] args) {
            // TODO Auto-generated method stub
            Singleton instance = new Singleton ();
            instance.getInstance("heyFromEmploye");
```

```
                Singleton instance1= new Singleton  ();
                instance1.getInstance("heyFromStudent");

          }
}
```

Here you can see two separate counter value occurred because at the backhand instance are created more then one time. And this is not right to create more then one instance.



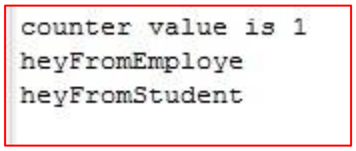
```
<terminated> Singleton [Java Application] C:\Program Files\Java\jre1.8.0_281\bin\javaw.exe (Ma
counter value is 1
heyFromEmploye
counter value is 2
heyFromStudent
```

Following code are singleton based solution, that restrict to create more then one object. You can see counter value is one due to one instance using lazy Initialization.

```
      public class Singleton {
//
      private static int counter=0;
      private static Singleton instance=null ;

        public static Singleton getInstance() {
              if(instance == null) {
                  instance = new Singleton(); }
          return instance; }
        private Singleton(){
              counter++;
              System.out.println("counter value is "+counter);
        }
```

```
public static void getInstance(String Message) {
             System.out.println(Message);
        }
      public static void main(String[] args) {
            // TODO Auto-generated method stub
            Singleton instance = Singleton.getInstance();
            instance.getInstance("heyFromEmploye");

             Singleton instance1 = Singleton.getInstance();

             instance1.getInstance("heyFromStudent");
    }}
```

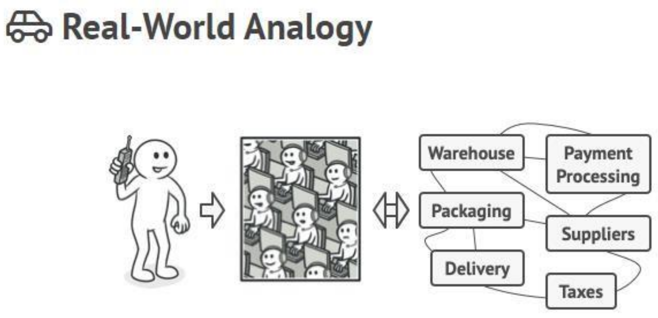
Result



```
counter value is 1
heyFromEmploye
heyFromStudent
```

Faced Design Pattern

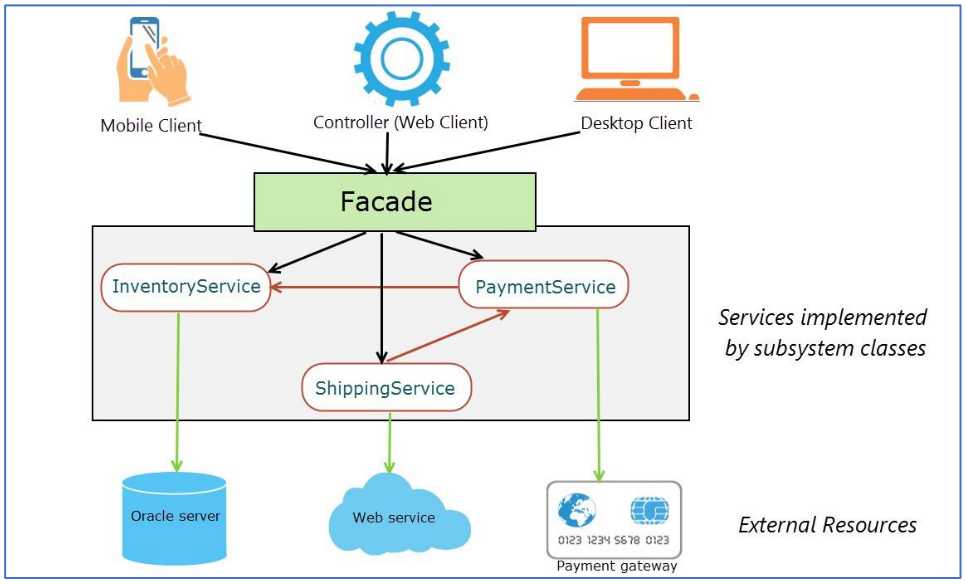
- Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
- Wrap a complicated subsystem with a simpler interface.



Real world facade examples

- To understand the facade, let's take a very simple example of a desktop computer machine. When we have to start a computer, all we have to do is press the start button. We really do not care what all things go inside the computer hardware and software. It is an example of Facade pattern.

- In Java programming, we must have connected to a database to fetch some data. We simply call the method dataSource.getConnection() to get the connection but internally a lot of things happen such as loading the driver, creating connection or fetching connection from pool, update stats and then return the connection reference to caller method. It is another example of Facade pattern in the programming world.



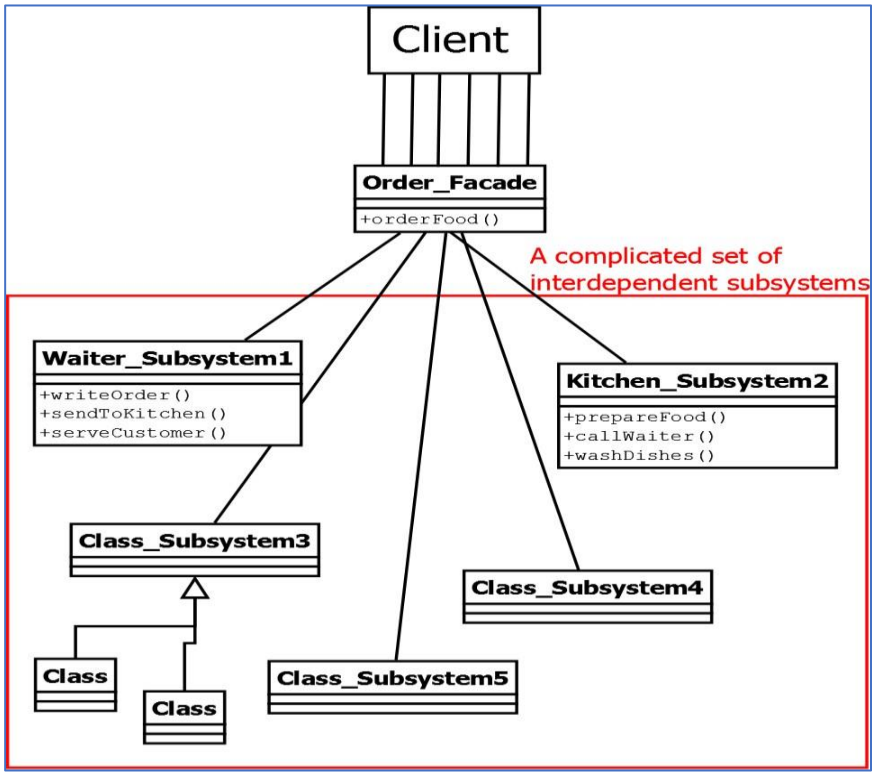
Example-01-Food Order

1. **Client:** The client in this example is the customer of a restaurant that wants to order food.

2. **Facade:** Its job is to be able to provide to the client more simplified access towards numerous interdependent subsystems that are considered complicated. In this example, a client's food-order would require a series of carefully sequenced method calls of two different subsystems (Waiter and Kitchen).

3. **Subsystems:** The subsystems are hidden from the client. They might also be not accessible to the client. The client cannot fiddle with any of the subsystems where a simple code-change may prove to be fatal or even break other unknown parts of the system itself. In this scenario, the waiter and the kitchen have to do a series of tasks. A subsystem' s task is sometimes dependent on the another's task.

4. For example, the kitchen cannot prepare the food if the waiter doesn't bring the order to the kitchen. The waiter cannot serve the customer if the food is not cooked.



Code

Waiter_Subsystem.java

```
public class Waiter_Subsystem1 {
      public  void writeOrder()    {
            System.out.println(" Waiter writes client's order\n");
      }
      public void sendToKitchen(){
            System.out.println(" Send order to kitchen\n");
      }
      public void serveCustomer(){
            System.out.println(" Yeeei customer is served!!!\n");
}}
```

Kitchen_Subsystem2.java

```
public class Kitchen_Subsystem2 {

         public void prepareFood(){ System.out.println("  Cook food\n");}
         public void callWaiter() { System.out.println("  Call Waiter\n");}
         public void washDishes() { System.out.println("  Wash the dishes\n");}
}
```

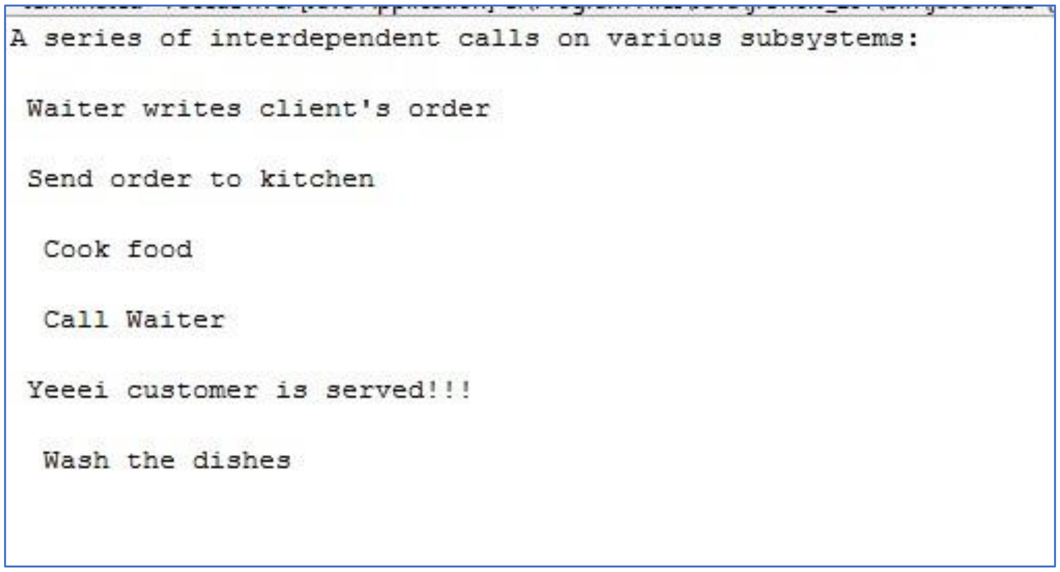
Order_Facade.java

```
public class Order_Facade {
       Waiter_Subsystem1 waiter=new Waiter_Subsystem1();
       Kitchen_Subsystem2 kitchen=new Kitchen_Subsystem2();
      public
          void orderFood()
          {
                  System.out.println("A series of interdependent calls on various
subsystems:\n");

              waiter.writeOrder();
              waiter.sendToKitchen();
              kitchen.prepareFood();
              kitchen.callWaiter();
              waiter.serveCustomer();
              kitchen.washDishes();
          }
}
```

FacedDriver.java

```
public class FacedDriver {

      public static void main(String[] args) {
            // TODO Auto-generated method stub
            Order_Facade facade = new Order_Facade();
          facade.orderFood();
      }}
```

Result



```
A series of interdependent calls on various subsystems:

 Waiter writes client's order

 Send order to kitchen

  Cook food

  Call Waiter

 Yeeei customer is served!!!

  Wash the dishes
```

When Should this pattern be used?

The facade pattern is appropriate when you have a **complex system** that you want to expose to clients in a simplified way, or you want to make an external communication layer over an existing system which is incompatible with the system. Facade deals with interfaces, not implementation. Its purpose is to hide internal complexity behind a single interface that appears simple on the outside.

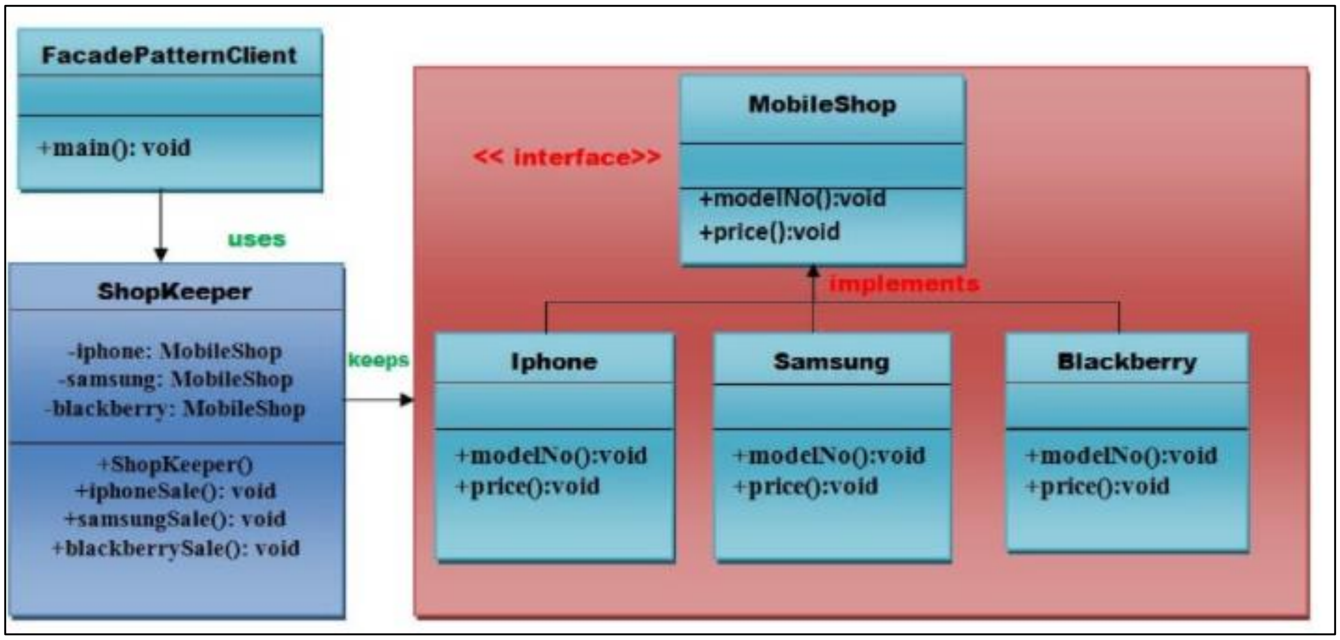
Example-02-Mobile Shop

There are three different type of mobile products that details are save in mobile shop , all the details are not display directly to the customer, only customer will able to buy the product from shopkeeper, shopkeeper will entertain the customer according to the customer requirement and budget

Let's understand the example of facade design pattern by the above UML diagram.

UML for Facade Pattern:

Let us look into the Class Diagram and the Java code.

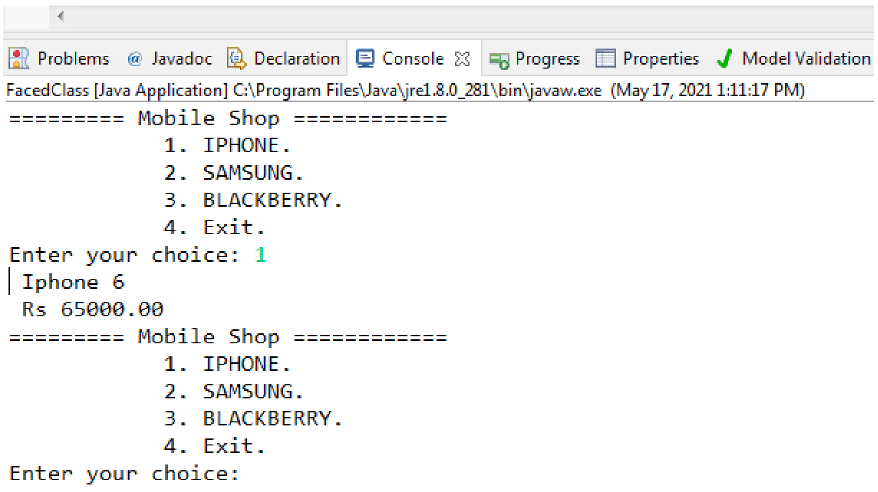


Code:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

interface MobileShop {
    public void modelNo();
    public void price();
}
class Iphone implements MobileShop {
    @Override
    public void modelNo() {
        System.out.println(" Iphone 6
");
    }
    @Override
    public void price() {
    System.out.println(" Rs 65000.00
");
    }
}
 class Samsung implements MobileShop {
    @Override
    public void modelNo() {
    System.out.println(" Samsung
galaxy tab 3 ");
    }
    @Override
    public void price() {
        System.out.println(" Rs
45000.00 ");
    }
}
 class Blackberry implements
MobileShop {
    @Override
    public void modelNo() {
    System.out.println(" Blackberry
Z10 ");
    }
    @Override
    public void price() {
        System.out.println(" Rs
55000.00 ");
    }
}
 class ShopKeeper {
    private MobileShop iphone;
    private MobileShop samsung;
    private MobileShop blackberry;

    public ShopKeeper(){
        iphone= new Iphone();
        samsung=new Samsung();
        blackberry=new Blackberry();
    }
    public void iphoneSale(){
        iphone.modelNo();
        iphone.price();
    }
        public void samsungSale(){
        samsung.modelNo();
        samsung.price();
    }
   public void blackberrySale(){
    blackberry.modelNo();
    blackberry.price();
        }
}
 public class FacedClass {

      private static int  choice;
     public static void main(String
args[]) throws NumberFormatException,
IOException{
        do{

System.out.print("========= Mobile
Shop ============ \n");
            System.out.print("
1. IPHONE.              \n");
            System.out.print("
2. SAMSUNG.              \n");
            System.out.print("
3. BLACKBERRY.            \n");
            System.out.print("
4. Exit.                      \n");
            System.out.print("Enter
your choice: ");

            BufferedReader br=new
BufferedReader(new
InputStreamReader(System.in));
```

output

```
choice=Integer.parseInt(br.readLine())
;
            ShopKeeper sk=new
ShopKeeper();

            switch (choice) {
            case 1:
                {
                  sk.iphoneSale();
                    }
                break;
       case 2:
                {
                  sk.samsungSale();
                    }
                break;
       case 3:
                            {

sk.blackberrySale();
                            }
                   break;
            default:
            {

System.out.println("Nothing You
purchased");
            }
                return;
            }

      }while(choice!=4);
   }

}
```



FacedClass [Java Application] C:\Program Files\Java\jre1.8.0_281\bin\javaw.exe (May 17, 2021 1:11:17 PM)

```
========= Mobile Shop ============
            1. IPHONE.
            2. SAMSUNG.
            3. BLACKBERRY.
            4. Exit.
Enter your choice: 1
 Iphone 6
 Rs 65000.00
========= Mobile Shop ============
            1. IPHONE.
            2. SAMSUNG.
            3. BLACKBERRY.
            4. Exit.
Enter your choice:
```