Factory Method Pattern

A Factory Pattern or Factory Method Pattern says that just **define an interface or abstract class for creating an object but let the subclasses decide which class to instantiate.** In other words, subclasses are responsible to create the instance of the class.

The Factory Method Pattern is also known as **Virtual Constructor.**

Advantage of Factory Design Pattern

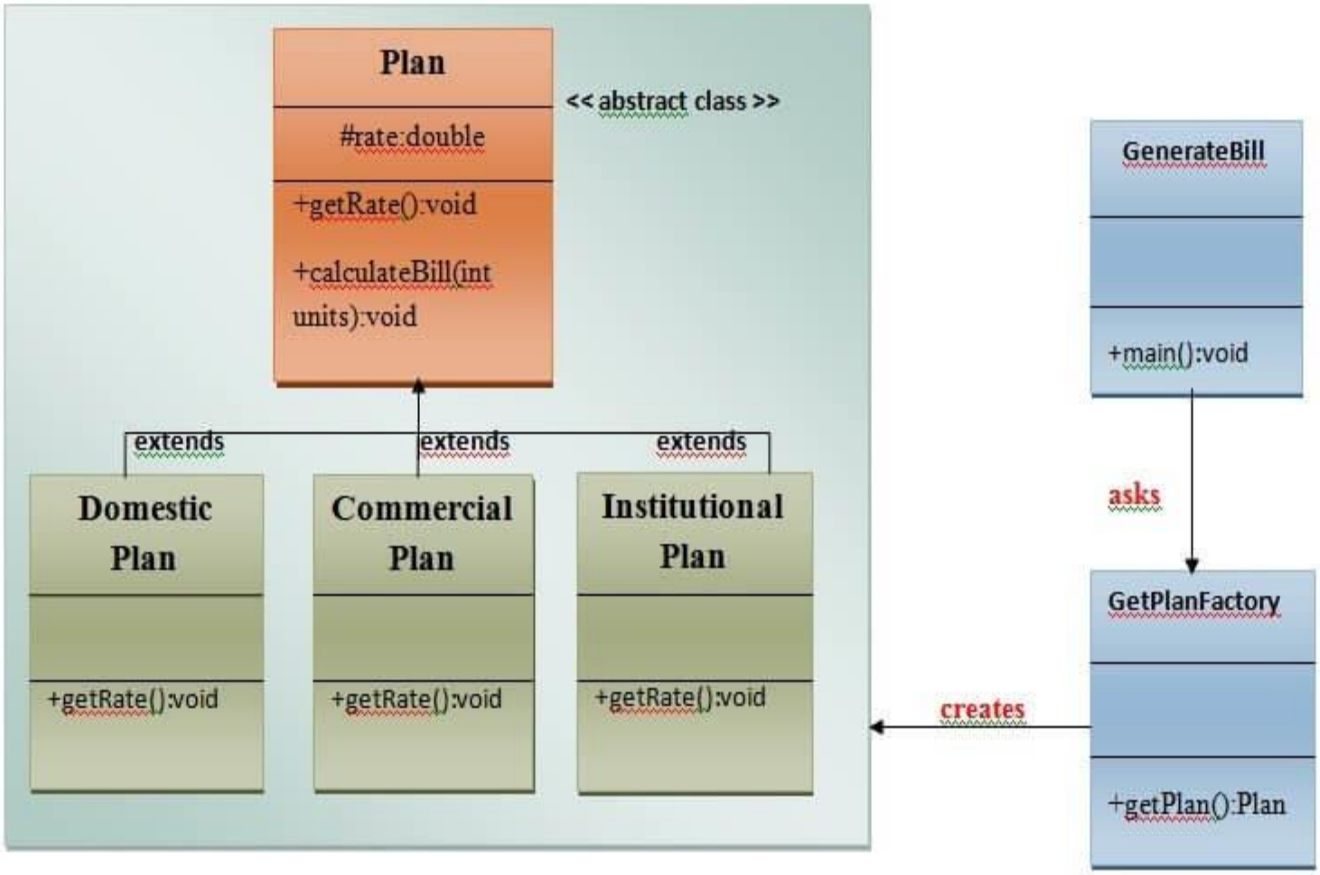
- Factory Method Pattern allows the sub-classes to choose the type of objects to create.
- It promotes the **loose-coupling** by eliminating the need to bind application-specific classes into the code. That means the code interacts solely with the resultant interface or abstract class, so that it will work with any classes that implement that interface or that extends that abstract class.

Usage of Factory Design Pattern

- When a class doesn't know what sub-classes will be required to create
- When a class wants that its sub-classes specify the objects to be created.
- When the parent classes choose the creation of objects to its sub-classes.

UML for Factory Method Pattern

- We are going to create a Plan abstract class and concrete classes that extends the Plan abstract class. A factory class GetPlanFactory is defined as a next step.
- GenerateBill class will use GetPlanFactory to get a Plan object. It will pass information (DOMESTICPLAN / COMMERCIALPLAN / INSTITUTIONALPLAN) to GetPalnFactory to get the type of object it needs.



Calculate Electricity Bill : A Real World Example of Factory Method

Step 1: Create a Plan abstract class.

```
import java.io.*;
    abstract class Plan{
    protected double rate;
  abstract void getRate();

      public void calculateBill(int units){
       System.out.println(units*rate);
    }
}
```

Step 2: Create the concrete classes that extends Plan abstract class.

```
class  DomesticPlan extends Plan{
       //@override
        public void getRate(){
            rate=3.50;
        }
  }//end of DomesticPlan class.


class  CommercialPlan extends Plan{
  //@override
        public void getRate(){
       rate=7.50;
  }
/end of CommercialPlan class.


class  InstitutionalPlan extends Plan{
       //@override
   public void getRate(){
       rate=5.50;
  }
/end of InstitutionalPlan class.
```

Step 3: Create a GetPlanFactory to generate object of concrete classes based on given information..

```
    class GetPlanFactory{

  //use getPlan method to get object of type Plan
      public Plan getPlan(String planType){
          if(planType == null){
           return null;
          }
        if(planType.equalsIgnoreCase("DOMESTICPLAN")) {
             return new DomesticPlan();
           }
         else if(planType.equalsIgnoreCase("COMMERCIALPLAN")){
```

```
            return new CommercialPlan();
          }
        else if(planType.equalsIgnoreCase("INSTITUTIONALPLAN")) {
            return new InstitutionalPlan();
        }
     return null;
  }
}//end of GetPlanFactory class.
```

Step 4: Generate Bill by using the GetPlanFactory to get the object of concrete classes by passing an information such as type of plan DOMESTICPLAN or COMMERCIALPLAN or INSTITUTIONALPLAN.

```
import java.io.*;
class GenerateBill{
public static void main(String args[])throws IOException{
GetPlanFactory planFactory = new GetPlanFactory();

System.out.print("Enter the name of plan for which the bill will be generated: ");
BufferedReader br=new BufferedReader(new InputStreamReader(System.in));

String planName=br.readLine();
System.out.print("Enter the number of units for bill will be calculated: ");
int units=Integer.parseInt(br.readLine());
Plan p = planFactory.getPlan(planName);
//call getRate() method and calculateBill()method of DomesticPaln.
System.out.print("Bill amount for "+planName+" of  "+units+" units is: ");
p.getRate();
p.calculateBill(units);
 }
 }//end of GenerateBill class.
```

Abstract Factory Pattern

Abstract Factory Pattern says that just **define an interface or abstract class for creating families of related (or dependent) objects but without specifying their concrete sub-classes.** That means Abstract Factory lets a class returns a factory of classes. So, this is the reason that Abstract Factory Pattern is one level higher than the Factory Pattern.

An Abstract Factory Pattern is also known as **Kit.**

Advantage of Abstract Factory Pattern

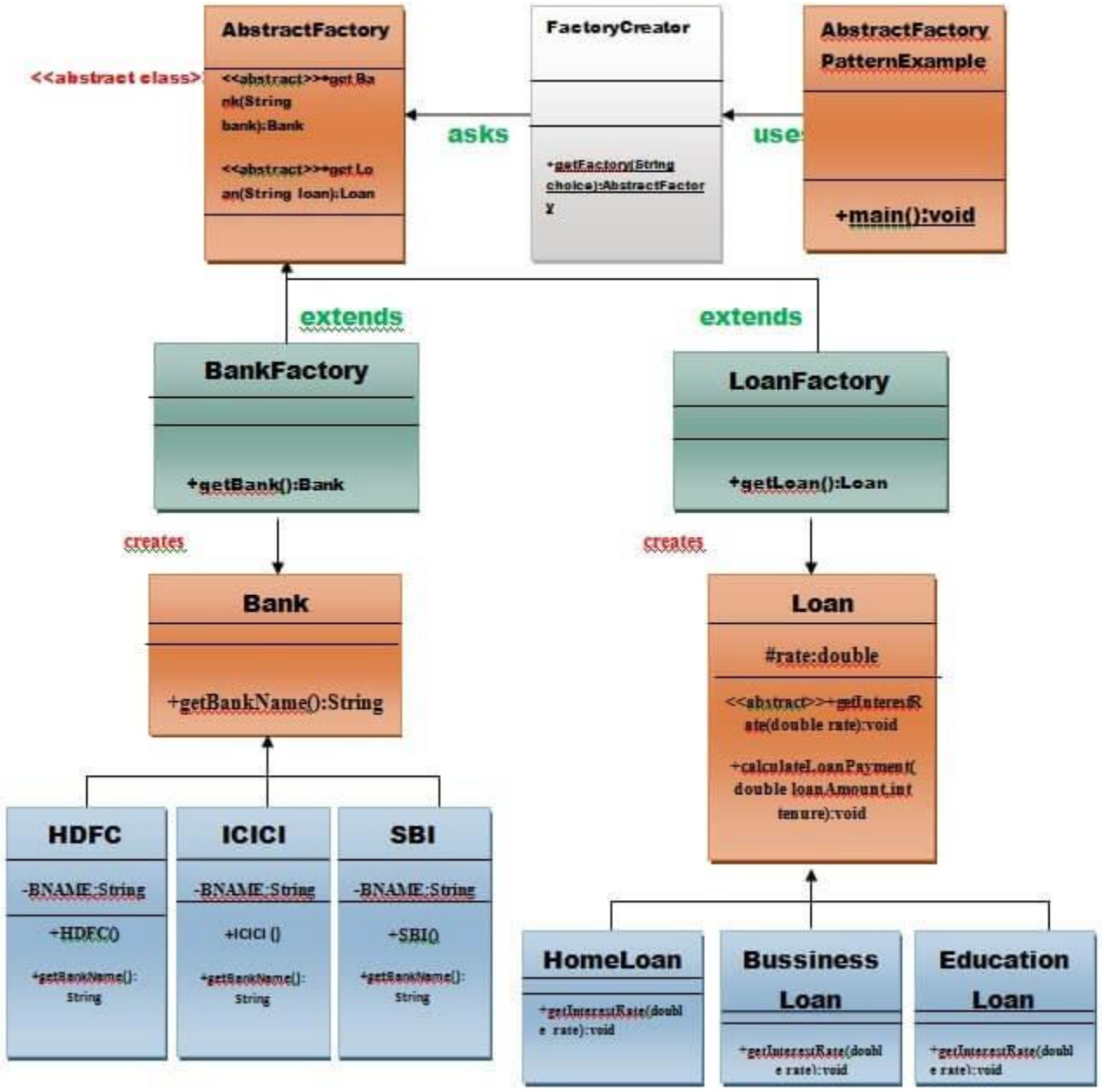
- Abstract Factory Pattern isolates the client code from concrete (implementation) classes.
- It eases the exchanging of object families.
- It promotes consistency among objects.

Usage of Abstract Factory Pattern

- When the system needs to be independent of how its object are created, composed, and represented.
- When the family of related objects has to be used together, then this constraint needs to be enforced.
- When you want to provide a library of objects that does not show implementations and only reveals interfaces.
- When the system needs to be configured with one of a multiple family of objects.

UML for Abstract Factory Pattern

- We are going to create a **Bank interface** and a **Loan abstract class** as well as their sub-classes.
- Then we will create **AbstractFactory** class as next step.
- Then after we will create concrete classes, **BankFactory,** and **LoanFactory** that will extends **AbstractFactory class**
- After that, **AbstractFactoryPatternExample** class uses the **FactoryCreator** to get an object of **AbstractFactory** class.
- See the diagram carefully which is given below:



Example of Abstract Factory Pattern

Here, we are calculating the loan payment for different banks like HDFC, ICICI, SBI etc.

Step 1: Create a Bank interface

```
import java.io.*;
interface Bank{
	String getBankName();
}
```

Step 2: Create concrete classes that implement the Bank interface.

```
class HDFC implements Bank{
        private final String BNAME;
        public HDFC(){
              BNAME="HDFC BANK";
       }
       public String getBankName() {
               return BNAME;
       }
}


class ICICI implements Bank{
      private final String BNAME;
      ICICI(){
              BNAME="ICICI BANK";
       }
       public String getBankName() {
               return BNAME;
      }
}


class SBI implements Bank{
     private final String BNAME;
     public SBI(){
             BNAME="SBI BANK";
      }
      public String getBankName(){
               return BNAME;
      }
}
```

Step 3: Create the Loan abstract class.

```
abstract class Loan{
  protected double rate;
  abstract void getInterestRate(double rate);
  public void calculateLoanPayment(double loanamount, int years)
  {

     /*
```

```
        to calculate the monthly loan payment i.e. EMI

        rate=annual interest rate/12*100;
        n=number of monthly installments;
        1year=12 months.
        so, n=years*12;

       */

      double EMI;
      int n;

      n=years*12;
      rate=rate/1200;
      EMI=((rate*Math.pow((1+rate),n))/((Math.pow((1+rate),n))-1))*loanamount;

System.out.println("your monthly EMI is "+ EMI +" for the amount"+loanamount+" you hav
e borrowed");
 }
}// end of the Loan abstract class.
```

Step 4: Create concrete classes that extend the Loan abstract class..

```
class HomeLoan extends Loan{
    public void getInterestRate(double r){
       rate=r;
   }
}//End of the HomeLoan class.

class BussinessLoan extends Loan{
   public void getInterestRate(double r){
        rate=r;
    }

}//End of the BusssinessLoan class.

class EducationLoan extends Loan{
    public void getInterestRate(double r){
      rate=r;
 }
}//End of the EducationLoan class.
```

Step 5: Create an abstract class (i.e AbstractFactory) to get the factories for Bank and Loan Objects.

```
abstract class AbstractFactory{
  public abstract Bank getBank(String bank);
  public abstract Loan getLoan(String loan);
}
```

Step 6: Create the factory classes that inherit AbstractFactory class to generate the object of concrete class based on given information.

```
    class BankFactory extends AbstractFactory{
public Bank getBank(String bank){
     if(bank == null){
       return null;
     }
     if(bank.equalsIgnoreCase("HDFC")){
       return new HDFC();
     } else if(bank.equalsIgnoreCase("ICICI")){
       return new ICICI();
     } else if(bank.equalsIgnoreCase("SBI")){
       return new SBI();
     }
     return null;
  }
  public Loan getLoan(String loan) {
     return null;
  }
}//End of the BankFactory class.

class LoanFactory extends AbstractFactory{
          public Bank getBank(String bank){
              return null;
          }

     public Loan getLoan(String loan){
     if(loan == null){
       return null;
     }
     if(loan.equalsIgnoreCase("Home")){
       return new HomeLoan();
     } else if(loan.equalsIgnoreCase("Business")){
       return new BussinessLoan();
```

```
    } else if(loan.equalsIgnoreCase("Education")){
       return new EducationLoan();
    }
    return null;
  }

}
```

Step 7: Create a FactoryCreator class to get the factories by passing an information such as Bank or Loan.

```
class FactoryCreator {
   public static AbstractFactory getFactory(String choice){
    if(choice.equalsIgnoreCase("Bank")){
       return new BankFactory();
    } else if(choice.equalsIgnoreCase("Loan")){
       return new LoanFactory();
    }
    return null;
  }
}//End of the FactoryCreator.
```

Step 8: Use the FactoryCreator to get AbstractFactory in order to get factories of concrete classes by passing an information such as type.

```
import java.io.*;
class AbstractFactoryPatternExample {
    public static void main(String args[])throws IOException {

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

    System.out.print("Enter the name of Bank from where you want to take loan amount: "
);
    String bankName=br.readLine();

System.out.print("\n");
System.out.print("Enter the type of loan e.g. home loan or business loan or education loan :
 ");

String loanName=br.readLine();
AbstractFactory bankFactory = FactoryCreator.getFactory("Bank");
Bank b=bankFactory.getBank(bankName);
```

```
System.out.print("\n");
System.out.print("Enter the interest rate for "+b.getBankName()+ ": ");

double rate=Double.parseDouble(br.readLine());
System.out.print("\n");
System.out.print("Enter the loan amount you want to take: ");

double loanAmount=Double.parseDouble(br.readLine());
System.out.print("\n");
System.out.print("Enter the number of years to pay your entire loan amount: ");
int years=Integer.parseInt(br.readLine());

System.out.print("\n");
System.out.println("you are taking the loan from "+ b.getBankName());

AbstractFactory loanFactory = FactoryCreator.getFactory("Loan");
        Loan l=loanFactory.getLoan(loanName);
        l.getInterestRate(rate);
        l.calculateLoanPayment(loanAmount,years);
  }
}//End of the  AbstractFactoryPatternExample
```

Singleton design pattern in Java

Singleton Pattern says that just**"define a class that has only one instance and provides a global point of access to it".**

In other words, a class must ensure that only single instance should be created and single object can be used by all other classes.

There are two forms of singleton design pattern

- **Early Instantiation:** creation of instance at load time.
- **Lazy Instantiation:** creation of instance when required.

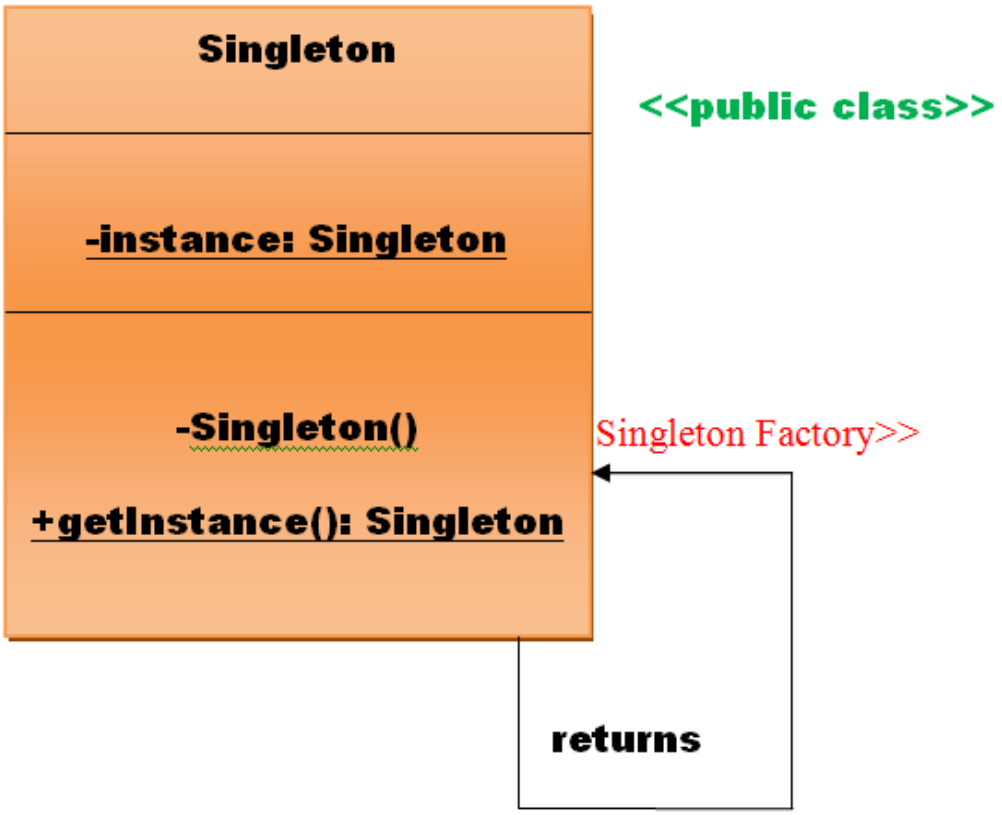
Advantage of Singleton design pattern

- Saves memory because object is not created at each request. Only single instance is reused again and again.

Usage of Singleton design pattern

- Singleton pattern is mostly used in multi-threaded and database applications. It is used in logging, caching, thread pools, configuration settings etc.

Uml of Singleton design pattern



How to create Singleton design pattern?

To create the singleton class, we need to have static member of class, private constructor and static factory method.

- **Static member:** It gets memory only once because of static, itcontains the instance of the Singleton class.
- **Private constructor:** It will prevent to instantiate the Singleton class from outside the class.
- **Static factory method:** This provides the global point of access to the Singleton object and returns the instance to the caller.

Understanding early Instantiation of Singleton Pattern

In such case, we create the instance of the class at the time of declaring the static data member, so instance of the class is created at the time of classloading.

Let's see the example of singleton design pattern using early instantiation.

File: A.java

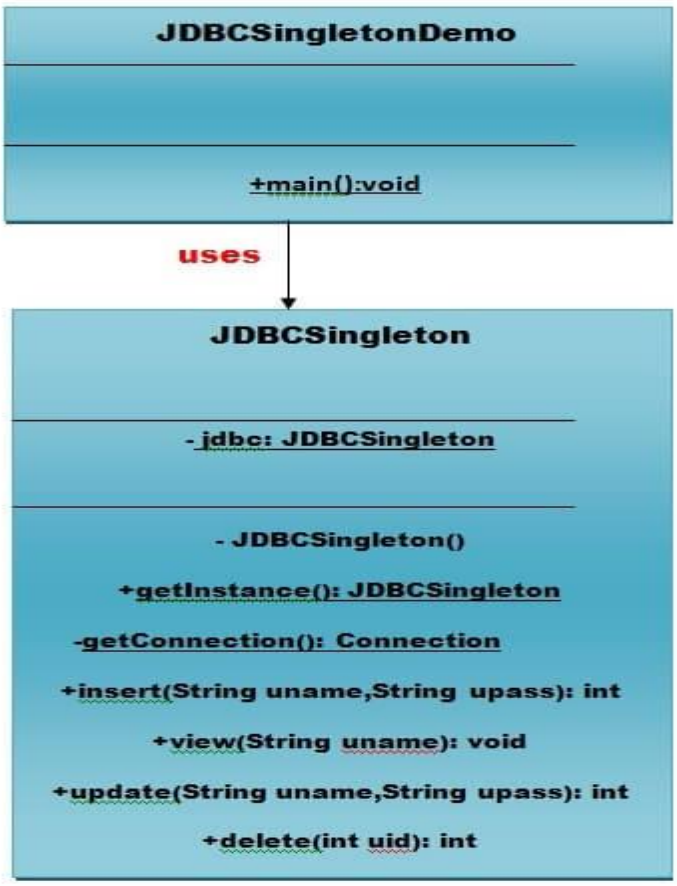
```
    class A{
private static A obj=new A();//Early, instance will be created at load time
private A(){}

public static A getA(){
 return obj;
}

public void doSomething(){
//write your code
 }
}
```

Understanding Real Example of Singleton Pattern

- We are going to create a JDBCSingleton class. This JDBCSingleton class contains its constructor as private and a private static instance jdbc of itself.
- JDBCSingleton class provides a static method to get its static instance to the outside world. Now, JDBCSingletonDemo class will use JDBCSingleton class to get the JDBCSingleton object.



Assumption: you have created a table userdata that has three fields uid, uname and upassword in mysql database. Database name is ashwinirajput, username is root, password is ashwini.

File: JDBCSingleton.java

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

class JDBCSingleton {
    //Step 1
     // create a JDBCSingleton class.
```

```
    //static member holds only one instance of the JDBCSingleton class.

        private static JDBCSingleton jdbc;

    //JDBCSingleton prevents the instantiation from any other class.
      private JDBCSingleton() {  }

   //Now we are providing gloabal point of access.
        public static JDBCSingleton getInstance() {
                              if (jdbc==null)
                             {
                                    jdbc=new  JDBCSingleton();
                             }
                    return jdbc;
          }

  // to get the connection from methods like insert, view etc.
        private static Connection getConnection()throws ClassNotFoundException,
 SQLException
        {

           Connection con=null;
           Class.forName("com.mysql.jdbc.Driver");
           con= DriverManager.getConnection("jdbc:mysql://localhost:3306/ashwan
irajput", "root", "ashwani");
           return con;

        }

 //to insert the record into the database
        public int insert(String name, String pass) throws SQLException
        {
           Connection c=null;

           PreparedStatement ps=null;

           int recordCounter=0;

           try {
```

```
                c=this.getConnection();
                ps=c.prepareStatement("insert into userdata(uname,upassword)values
(?,?)");
                ps.setString(1, name);
                ps.setString(2, pass);
                recordCounter=ps.executeUpdate();

            } catch (Exception e) { e.printStackTrace(); } finally{
                if (ps!=null){
                  ps.close();
                }if(c!=null){
                  c.close();
                }
              }
             return recordCounter;
          }

//to view the data from the database
     public  void view(String name) throws SQLException
     {
            Connection con = null;
       PreparedStatement ps = null;
       ResultSet rs = null;

            try {

                con=this.getConnection();
                ps=con.prepareStatement("select * from userdata where uname=?");

                ps.setString(1, name);
                rs=ps.executeQuery();
                while (rs.next()) {
                      System.out.println("Name= "+rs.getString(2)+"\t"+"Paasword
= "+rs.getString(3));

                }

        } catch (Exception e) { System.out.println(e);}
        finally{
              if(rs!=null){
```

```
                rs.close();
              }if (ps!=null){
                ps.close();
             }if(con!=null){
                con.close();
             }
           }
     }

    // to update the password for the given username
     public int update(String name, String password) throws SQLException  {
           Connection c=null;
           PreparedStatement ps=null;

           int recordCounter=0;
           try {
                 c=this.getConnection();
                 ps=c.prepareStatement(" update userdata set upassword=? where una
me='"+name+"' ");
                 ps.setString(1, password);
                 recordCounter=ps.executeUpdate();
           } catch (Exception e) {  e.printStackTrace(); } finally{

              if (ps!=null){
                 ps.close();
              }if(c!=null){
                 c.close();
              }
            }
          return recordCounter;
        }

// to delete the data from the database
       public int delete(int userid) throws SQLException{
           Connection c=null;
           PreparedStatement ps=null;
           int recordCounter=0;
           try {
                  c=this.getConnection();
```

```
                ps=c.prepareStatement(" delete from userdata where uid='"+userid+"'
");
                recordCounter=ps.executeUpdate();
            } catch (Exception e) { e.printStackTrace(); }
            finally{
            if (ps!=null){
                ps.close();
            }if(c!=null){
                c.close();
            }
          }
            return recordCounter;
        }
}// End of JDBCSingleton class
```

File: JDBCSingletonDemo.java

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
class JDBCSingletonDemo{
   static int count=1;
   static int  choice;
   public static void main(String[] args) throws IOException {

      JDBCSingleton jdbc= JDBCSingleton.getInstance();


      BufferedReader br=new BufferedReader(new InputStreamReader(System.in)
);
  do{
      System.out.println("DATABASE OPERATIONS");
      System.out.println(" ---------------------- ");
      System.out.println(" 1. Insertion ");
```

```
        System.out.println(" 2. View      ");
        System.out.println(" 3. Delete    ");
        System.out.println(" 4. Update    ");
        System.out.println(" 5. Exit      ");

        System.out.print("\n");
        System.out.print("Please enter the choice what you want to perform in the dat
abase: ");

        choice=Integer.parseInt(br.readLine());
        switch(choice) {

          case 1:{
                System.out.print("Enter the username you want to insert data into the
database: ");
                String username=br.readLine();
                System.out.print("Enter the password you want to insert data into the d
atabase: ");
                String password=br.readLine();

                try {
                     int i= jdbc.insert(username, password);
                     if (i>0) {
                     System.out.println((count++) + " Data has been inserted successf
ully")
                     }else{
                        System.out.println("Data has not been inserted ");
                     }

                   } catch (Exception e) {
                     System.out.println(e);
                   }

                 System.out.println("Press Enter key to continue...");
                 System.in.read();

               }//End of case 1
               break;
           case 2:{
                System.out.print("Enter the username : ");
```

```
                String username=br.readLine();

                try  {
                      jdbc.view(username);
                    } catch (Exception e) {
                     System.out.println(e);
                    }
                 System.out.println("Press Enter key to continue...");
                 System.in.read();

                }//End of case 2
               break;
            case 3:{
                 System.out.print("Enter the userid,  you want to delete: ");
                 int userid=Integer.parseInt(br.readLine());

                 try {
                      int i= jdbc.delete(userid);
                      if (i>0) {
                      System.out.println((count++) + " Data has been deleted successfu
lly");
                      }else{
                         System.out.println("Data has not been deleted");
                      }

                    } catch (Exception e) {
                     System.out.println(e);
                    }
                 System.out.println("Press Enter key to continue...");
                 System.in.read();

                }//End of case 3
               break;
            case 4:{
                System.out.print("Enter the username,  you want to update: ");
                String username=br.readLine();
                System.out.print("Enter the new password ");
                String password=br.readLine();

                try {
```

```
                    int i= jdbc.update(username, password);
                    if (i>0) {
                    System.out.println((count++) + " Data has been updated successf
ully");
                    }

                  } catch (Exception e) {
                    System.out.println(e);
                  }
                System.out.println("Press Enter key to continue...");
                System.in.read();

              }// end of case 4
            break;

          default:
                return;
       }

      } while (choice!=4);
  }
}
```