Java Enterprise Edition

ANUDIP FOUNDATION

Introduction to JDBC

Objective: • Understanding connectivity of JDBC • Design pattern • Understanding Layers in JEE	**Materials Required:** 1. Eclipse IDE/IntelliJ 2. Jdk 1.8 or later 3. JDBC driver 4. SQL
Theory:240mins	**Pratical:120mins**
Total Duration: 360 mins	

Introduction to JDBC

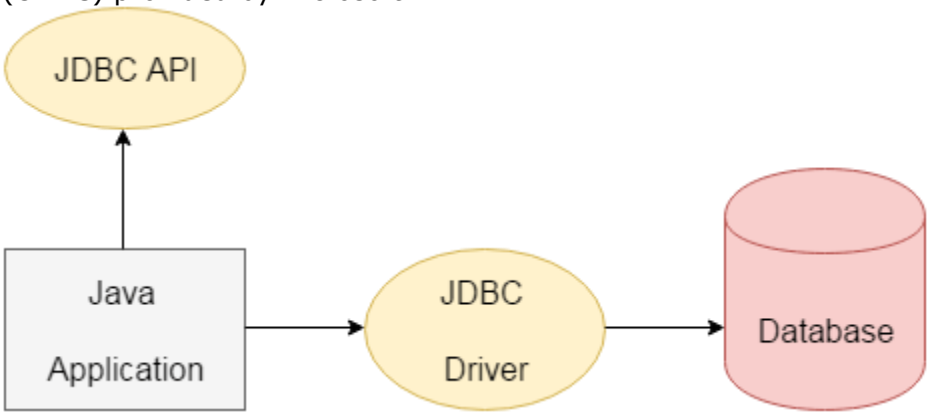
Connection, Statement, PreparedStatement, ResultSet
Self learning with online links and explanation by Trainer with Demos
Creational Design Pattern
Factory Pattern
§ Singleton Pattern
§ Prototype Pattern
o Structural Design Pattern
§ Decorator Pattern
§ Facade Pattern
o Behavioral Design Pattern
§ Chain of Responsibility Pattern
§ Iterator Pattern
o Presentation Layer Design Pattern
o Business Layer Design Pattern
§ Business Delegate Pattern
§ Transfer Object Pattern
o Integration Layer Design Pattern
§ Data Access Object Pattern

Introduction to JDBC

JDBC stands for Java Database Connectivity. JDBC is a Java API to connect and execute the query with the database. It is a part of JavaSE (Java Standard Edition). JDBC API uses JDBC drivers to connect with the database. There are four types of JDBC drivers:

- JDBC-ODBC Bridge Driver,
- Native Driver,
- Network Protocol Driver, and
- Thin Driver

We have discussed the above four drivers in the next chapter.
We can use JDBC API to access tabular data stored in any relational database. By the help of JDBC API, we can save, update, delete and fetch data from the database. It is like Open Database Connectivity (ODBC) provided by Microsoft.



The current version of JDBC is 4.3. It is the stable release since 21st September, 2017. It is based on the X/Open SQL Call Level Interface. The **java.sql** package contains classes and interfaces for JDBC API. A list of popular *interfaces* of JDBC API are given below:

- Driver interface
- Connection interface
- Statement interface
- PreparedStatement interface
- CallableStatement interface
- ResultSet interface
- ResultSetMetaData interface
- DatabaseMetaData interface
- RowSet interface

A list of popular *classes* of JDBC API are given below:

- DriverManager class
- Blob class
- Clob class
- Types class

Why Should We Use JDBC

Before JDBC, ODBC API was the database API to connect and execute the query with the database. But, ODBC API uses ODBC driver which is written in C language (i.e. platform dependent and unsecured). That is why Java has defined its own API (JDBC API) that uses JDBC drivers (written in Java language). We can use JDBC API to handle database using Java program and can perform the following activities:

1. Connect to the database
2. Execute queries and update statements to the database
3. Retrieve the result received from the database.

Connection, Statement, PreparedStatement, ResultSet

Connection interface

A Connection is the session between java application and database. The Connection interface is a factory of Statement, PreparedStatement, and DatabaseMetaData i.e. object of Connection can be used to get the object of Statement and DatabaseMetaData. The Connection interface provide many methods for transaction management like commit(), rollback() etc.

By default, connection commits the changes after executing queries.

Commonly used methods of Connection interface:

1) public Statement createStatement(): creates a statement object that can be used to execute SQL queries.

2) public Statement createStatement(int resultSetType,int resultSetConcurrency): Creates a Statement object that will generate ResultSet objects with the given type and concurrency.

3) public void setAutoCommit(boolean status): is used to set the commit status.By default it is true.

4) public void commit(): saves the changes made since the previous commit/rollback permanent.

5) public void rollback(): Drops all changes made since the previous commit/rollback.

6) public void close(): closes the connection and Releases a JDBC resources immediately.

Statement interface

The **Statement interface** provides methods to execute queries with the database. The statement interface is a factory of ResultSet i.e. it provides factory method to get the object of ResultSet.

Commonly used methods of Statement interface:

The important methods of Statement interface are as follows:

1) public ResultSet executeQuery(String sql): is used to execute SELECT query. It returns the object of ResultSet.

2) public int executeUpdate(String sql): is used to execute specified query, it may be create, drop, insert, update, delete etc.

3) public boolean execute(String sql): is used to execute queries that may return multiple results.

4) public int[] executeBatch(): is used to execute batch of commands.

PreparedStatement interface

The PreparedStatement interface is a subinterface of Statement. It is used to execute parameterized query.

Let's see the example of parameterized query:

```
1. String sql="insert into emp values(?,?,?)";
```

As you can see, we are passing parameter (?) for the values. Its value will be set by calling the setter methods of PreparedStatement.

Why use PreparedStatement?

Improves performance: The performance of the application will be faster if you use PreparedStatement interface because query is compiled only once.

How to get the instance of PreparedStatement?

The prepareStatement() method of Connection interface is used to return the object of PreparedStatement. Syntax:

```
1. public PreparedStatement prepareStatement(String query)throws SQLException{}
```

Methods of PreparedStatement interface

The important methods of PreparedStatement interface are given below:

Method	Description
public void setInt(int paramIndex, int value)	sets the integer value to the given parameter index.
public void setString(int paramIndex, String value)	sets the String value to the given parameter index.
public void setFloat(int paramIndex, float value)	sets the float value to the given parameter index.
public void setDouble(int paramIndex, double value)	sets the double value to the given parameter index.
public int executeUpdate()	executes the query. It is used for create, drop, insert, update, delete etc.
public ResultSet executeQuery()	executes the select query. It returns an instance of ResultSet.

Example of PreparedStatement interface that inserts the record

First of all create table as given below:

```
1. create table emp(id number(10),name varchar2(50));
```

Now insert records in this table by the code given below:

```
1. import java.sql.*;
2. class InsertPrepared{
3. public static void main(String args[]){
4. try{
5. Class.forName("oracle.jdbc.driver.OracleDriver");
6.
7. Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system",
   "oracle");
8.
9. PreparedStatement stmt=con.prepareStatement("insert into Emp values(?,?)");
10. stmt.setInt(1,101);//1 specifies the first parameter in the query
11. stmt.setString(2,"Ratan");
12.
13. int i=stmt.executeUpdate();
14. System.out.println(i+" records inserted");
15.
16. con.close();
```

```
17.
18. }catch(Exception e){ System.out.println(e);}
19.
20. }
21. }
```

Example of PreparedStatement interface that updates the record

```
1. PreparedStatement stmt=con.prepareStatement("update emp set name=? where id=?");
2. stmt.setString(1,"Sonoo");//1 specifies the first parameter in the query i.e. name
3. stmt.setInt(2,101);
4.
5. int i=stmt.executeUpdate();
6. System.out.println(i+" records updated");
```

Example of PreparedStatement interface that deletes the record

```
1. PreparedStatement stmt=con.prepareStatement("delete from emp where id=?");
2. stmt.setInt(1,101);
3.
4. int i=stmt.executeUpdate();
5. System.out.println(i+" records deleted");
```

Example of PreparedStatement interface that retrieve the records of a table

```
1. PreparedStatement stmt=con.prepareStatement("select * from emp");
2. ResultSet rs=stmt.executeQuery();
3. while(rs.next()){
4. System.out.println(rs.getInt(1)+" "+rs.getString(2));
5. }
```

Example of PreparedStatement to insert records until user press n

```
1. import java.sql.*;
2. import java.io.*;
3. class RS{
4. public static void main(String args[])throws Exception{
5. Class.forName("oracle.jdbc.driver.OracleDriver");
6. Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system",
   "oracle");
7.
8. PreparedStatement ps=con.prepareStatement("insert into emp130 values(?,?,?)");
9.
10. BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
11.
12. do{
13. System.out.println("enter id:");
14. int id=Integer.parseInt(br.readLine());
15. System.out.println("enter name:");
16. String name=br.readLine();
17. System.out.println("enter salary:");
```

```
18. float salary=Float.parseFloat(br.readLine());
19.
20. ps.setInt(1,id);
21. ps.setString(2,name);
22. ps.setFloat(3,salary);
23. int i=ps.executeUpdate();
24. System.out.println(i+" records affected");
25.
26. System.out.println("Do you want to continue: y/n");
27. String s=br.readLine();
28. if(s.startsWith("n")){
29. break;
30. }
31. }while(true);
32.
33. con.close();
34. }}
```

ResultSet interface
The object of ResultSet maintains a cursor pointing to a row of a table. Initially, cursor points to before the first row.

By default, ResultSet object can be moved forward only and it is not updatable.

But we can make this object to move forward and backward direction by passing either TYPE_SCROLL_INSENSITIVE or TYPE_SCROLL_SENSITIVE in createStatement(int,int) method as well as we can make this object as updatable by:

```
1. Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
2.                     ResultSet.CONCUR_UPDATABLE);
```

Commonly used methods of ResultSet interface

1) public boolean next():	is used to move the cursor to the one row next from the current position.
2) public boolean previous():	is used to move the cursor to the one row previous from the current position.
3) public boolean first():	is used to move the cursor to the first row in result set object.
4) public boolean last():	is used to move the cursor to the last row in result set object.
5) public boolean absolute(int row):	is used to move the cursor to the specified row number in the ResultSet object.
6) public boolean relative(int row):	is used to move the cursor to the relative row number in the ResultSet object, it may be positive or negative.

7) public int getInt(int columnIndex):	is used to return the data of specified column index of the current row as int.
8) public int getInt(String columnName):	is used to return the data of specified column name of the current row as int.
9) public String getString(int columnIndex):	is used to return the data of specified column index of the current row as String.
10) public String getString(String columnName):	is used to return the data of specified column name of the current row as String.

Example of Scrollable ResultSet

Let's see the simple example of ResultSet interface to retrieve the data of 3rd row.

```
1. import java.sql.*;
2. class FetchRecord{
3. public static void main(String args[])throws Exception{
4.
5. Class.forName("oracle.jdbc.driver.OracleDriver");
6. Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system",
   "oracle");
7. Statement stmt=con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,ResultSet.CONCUR_
   UPDATABLE);
8. ResultSet rs=stmt.executeQuery("select * from emp765");
9.
10. //getting the record of 3rd row
11. rs.absolute(3);
12. System.out.println(rs.getString(1)+" "+rs.getString(2)+" "+rs.getString(3));
13.
14. con.close();
15. }}
```

Self learning with online links and explanation by Trainer with Demos

1. **https://docs.oracle.com/javase/tutorial/jdbc/overview/index.html**
2. **https://www.javatpoint.com/ResultSet-interface**
3. **https://www.geeksforgeeks.org/introduction-to-jdbc/**
4. **https://www.tutorialspoint.com/jdbc/jdbc-introduction.htm**
5. J2EE Design Patterns
6. Core J2EE Patterns: Best Practices and Design Strategies
7. http://www.corej2eepatterns.com

Creational design patterns
Creational design patterns are concerned with **the way of creating objects.** These design patterns are used when a decision must be made at the time of instantiation of a class (i.e. creating an object of a class).
But everyone knows an object is created by using new keyword in java. For example:

1. StudentRecord s1=new StudentRecord();

Hard-Coded code is not the good programming approach. Here, we are creating the instance by using the new keyword. Sometimes, the nature of the object must be changed according to the nature of the program. In such cases, we must get the help of creational design patterns to provide more general and flexible approach.

Types of creational design patterns

There are following 6 types of creational design patterns.

1. Factory Method Pattern
2. Abstract Factory Pattern
3. Singleton Pattern
4. Prototype Pattern
5. Builder Pattern
6. Object Pool Pattern

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

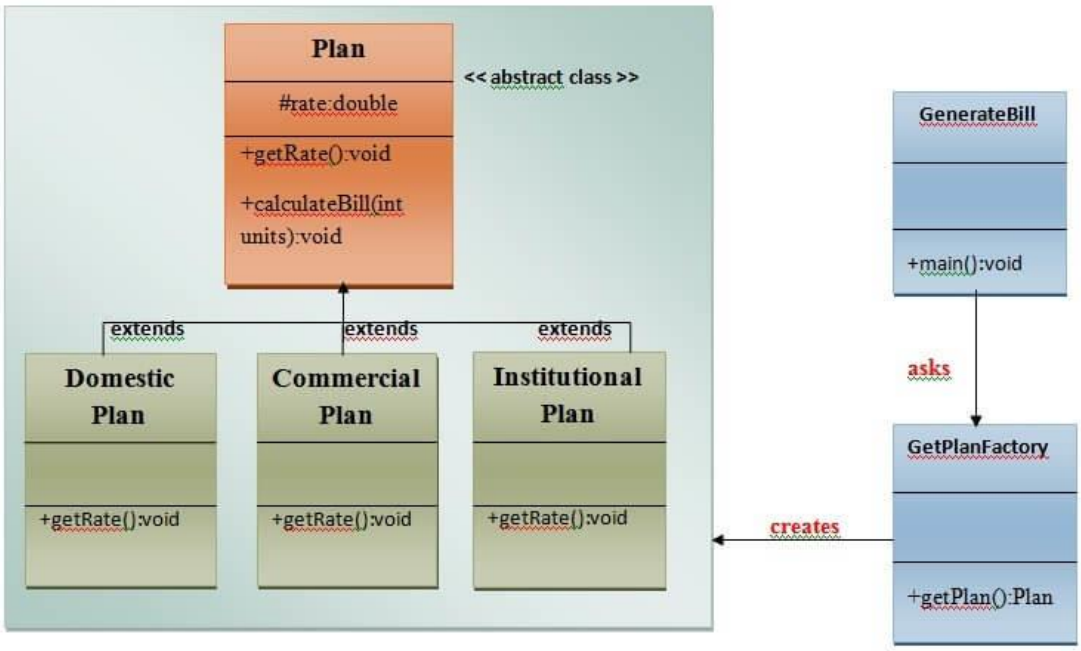
- Factory Method Pattern allows the sub-classes to choose the type of objects to create.
- It promotes the **loose-coupling** by eliminating the need to bind application-specific classes into the code. That means the code interacts solely with the resultant interface or abstract class, so that it will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

```
1. import java.io.*;
2. abstract class Plan{
3.         protected double rate;
4.         abstract void getRate();
5.
6.         public void calculateBill(int units){
7.             System.out.println(units*rate);
8.          }
9. }//end of Plan class.
```

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

```
1. class  DomesticPlan extends Plan{
2.        //@override
3.         public void getRate(){
4.            rate=3.50;
5.        }
6.    }//end of DomesticPlan class.
1. class  CommercialPlan extends Plan{
2.    //@override
3.     public void getRate(){
4.         rate=7.50;
5.    }
6. /end of CommercialPlan class.
1. class  InstitutionalPlan extends Plan{
2.    //@override
3.     public void getRate(){
4.         rate=5.50;
5.    }
6. /end of InstitutionalPlan class.
```

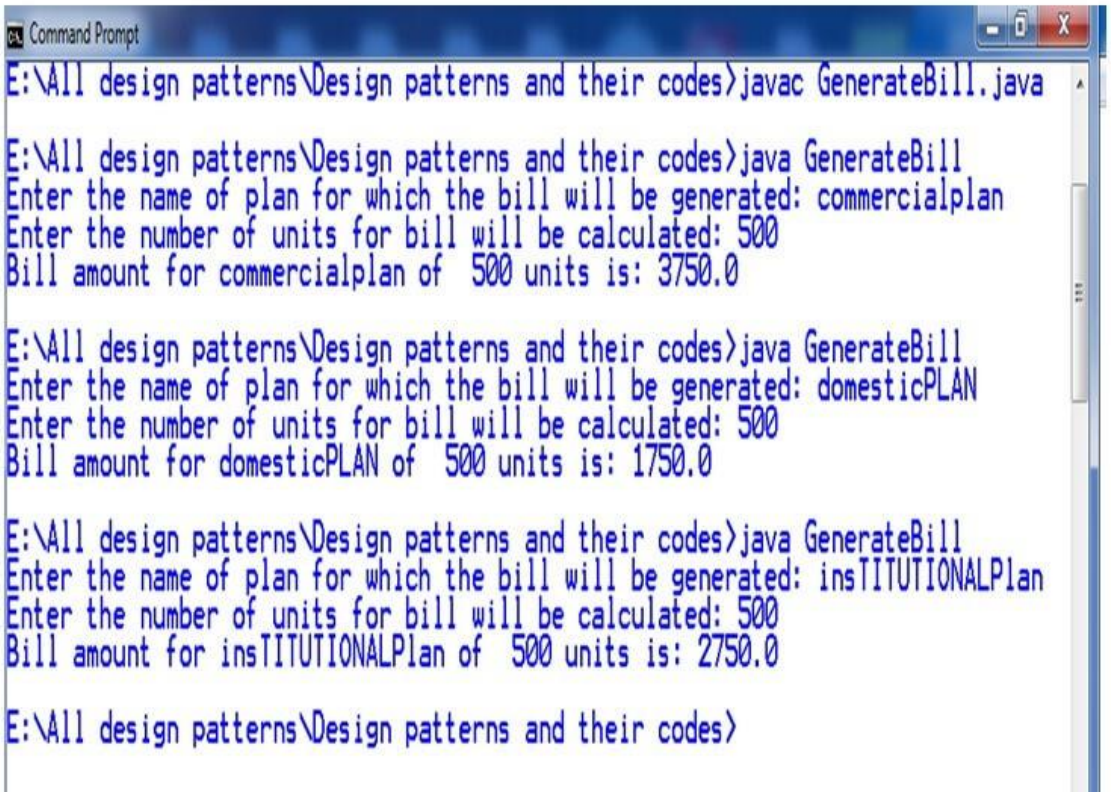
Step 3: Create a GetPlanFactory to generate object of concrete classes based on given information..

```
1.  class GetPlanFactory{
2.
3.     //use getPlan method to get object of type Plan
4.        public Plan getPlan(String planType){
5.            if(planType == null){
6.             return null;
7.            }
8.           if(planType.equalsIgnoreCase("DOMESTICPLAN")) {
9.                 return new DomesticPlan();
10.              }
11.          else if(planType.equalsIgnoreCase("COMMERCIALPLAN")){
12.              return new CommercialPlan();
13.           }
14.          else if(planType.equalsIgnoreCase("INSTITUTIONALPLAN")) {
15.              return new InstitutionalPlan();
16.          }
17.     return null;
18.  }
19. }//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.

```
1.  import java.io.*;
2.  class GenerateBill{
3.      public static void main(String args[])throws IOException{
4.        GetPlanFactory planFactory = new GetPlanFactory();
5.
6.        System.out.print("Enter the name of plan for which the bill will be generated: ");
7.        BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
8.
9.        String planName=br.readLine();
10.       System.out.print("Enter the number of units for bill will be calculated: ");
11.       int units=Integer.parseInt(br.readLine());
12.
13.       Plan p = planFactory.getPlan(planName);
14.       //call getRate() method and calculateBill()method of DomesticPaln.
15.
16.        System.out.print("Bill amount for "+planName+" of  "+units+" units is: ");
17.           p.getRate();
18.           p.calculateBill(units);
19.            }
20.    }//end of GenerateBill class.
```

Output



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

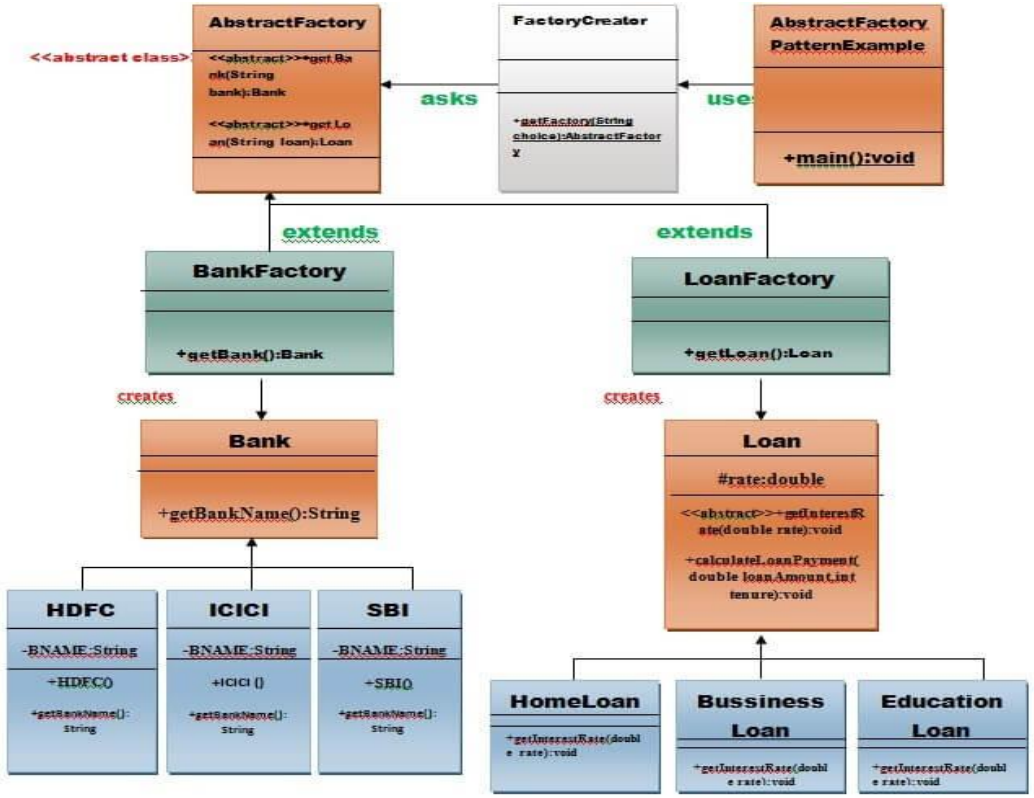
- Abstract Factory Pattern isolates the client code from concrete (implementation) classes.
- It eases the exchanging of object families.
- It promotes consistency among objects.

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

```
1. import java.io.*;
2. interface Bank{
3.         String getBankName();
4. }
```

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

```
1. class HDFC implements Bank{
2.          private final String BNAME;
3.          public HDFC(){
4.                 BNAME="HDFC BANK";
5.          }
6.          public String getBankName() {
7.                  return BNAME;
8.          }
```

```
}
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 have bo
   rrowed");
 }
}// 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")){
```

```
15.         return new EducationLoan();
16.       }
17.       return null;
18.    }
19.
20. }
```

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

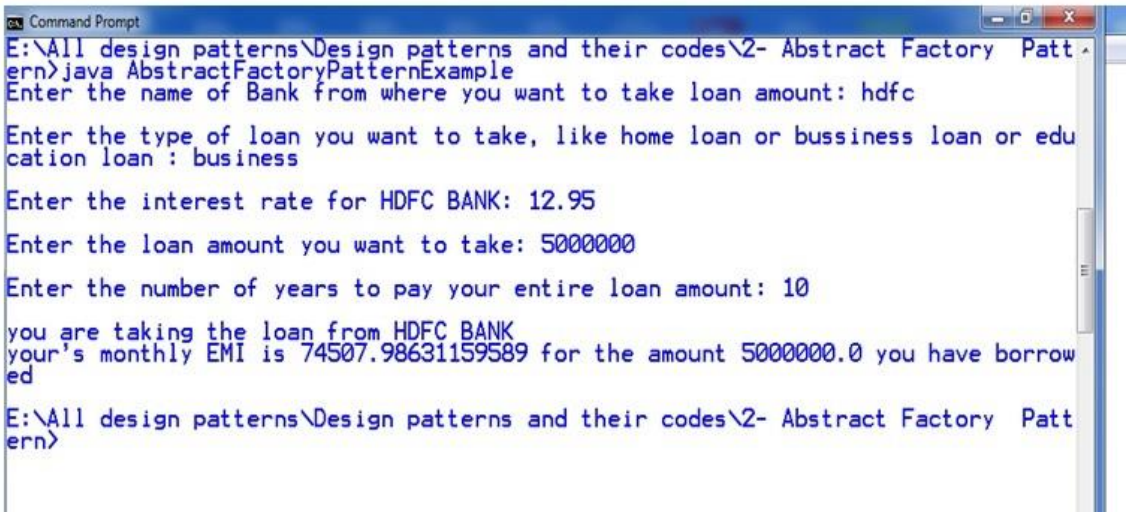
```
1.  class FactoryCreator {
2.        public static AbstractFactory getFactory(String choice){
3.        if(choice.equalsIgnoreCase("Bank")){
4.           return new BankFactory();
5.        } else if(choice.equalsIgnoreCase("Loan")){
6.           return new LoanFactory();
7.        }
8.        return null;
9.     }
10. }//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.

```
1.  import java.io.*;
2.  class AbstractFactoryPatternExample {
3.        public static void main(String args[])throws IOException {
4.
5.        BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
6.
7.        System.out.print("Enter the name of Bank from where you want to take loan amount: ");
8.        String bankName=br.readLine();
9.
10. System.out.print("\n");
11. System.out.print("Enter the type of loan e.g. home loan or business loan or education loan : ");

12.
13. String loanName=br.readLine();
14. AbstractFactory bankFactory = FactoryCreator.getFactory("Bank");
15. Bank b=bankFactory.getBank(bankName);
16.
17. System.out.print("\n");
18. System.out.print("Enter the interest rate for "+b.getBankName()+ ": ");
19.
20. double rate=Double.parseDouble(br.readLine());
21. System.out.print("\n");
22. System.out.print("Enter the loan amount you want to take: ");
23.
24. double loanAmount=Double.parseDouble(br.readLine());
25. System.out.print("\n");
26. System.out.print("Enter the number of years to pay your entire loan amount: ");
27. int years=Integer.parseInt(br.readLine());
28.
29. System.out.print("\n");
30. System.out.println("you are taking the loan from "+ b.getBankName());
```

```
31.
32. AbstractFactory loanFactory = FactoryCreator.getFactory("Loan");
33.          Loan l=loanFactory.getLoan(loanName);
34.          l.getInterestRate(rate);
35.          l.calculateLoanPayment(loanAmount,years);
36.   }
37. }//End of the  AbstractFactoryPatternExample
```

Output



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.

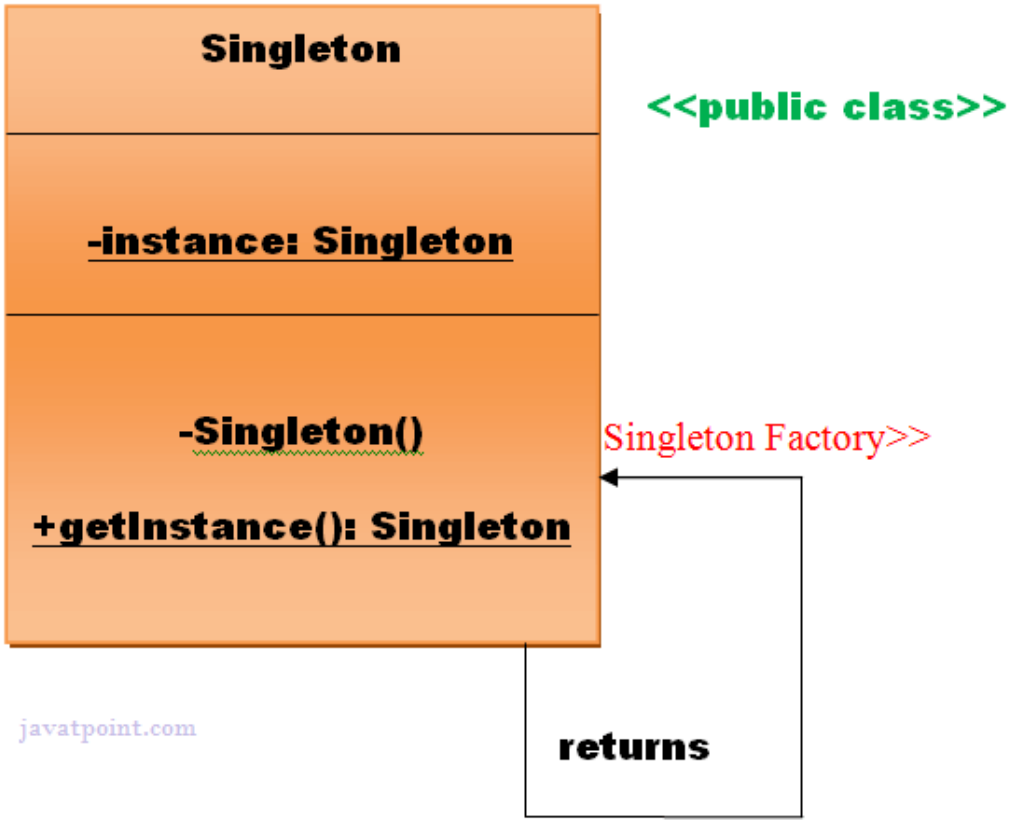
Advantage of Singleton design pattern

- 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

```
1. class A{
2.  private static A obj=new A();//Early, instance will be created at load time
3.  private A(){}
4.
5.  public static A getA(){
```

```
6.    return obj;
7.   }
8.
9.   public void doSomething(){
10. //write your code
11. }
12.}
```

Understanding lazy Instantiation of Singleton Pattern

In such case, we create the instance of the class in synchronized method or synchronized block, so instance of the class is created when required.

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

File: A.java

```
1.  class A{
2.   private static A obj;
3.   private A(){}
4.
5.   public static A getA(){
6.     if (obj == null){
7.       synchronized(Singleton.class){
8.         if (obj == null){
9.           obj = new Singleton();//instance will be created at request time
10.        }
11.    }
12.    }
13.  return obj;
14. }
15.
16. public void doSomething(){
17. //write your code
18. }
19.}
```

Significance of Classloader in Singleton Pattern

If singleton class is loaded by two classloaders, two instance of singleton class will be created, one for each classloader.

Significance of Serialization in Singleton Pattern

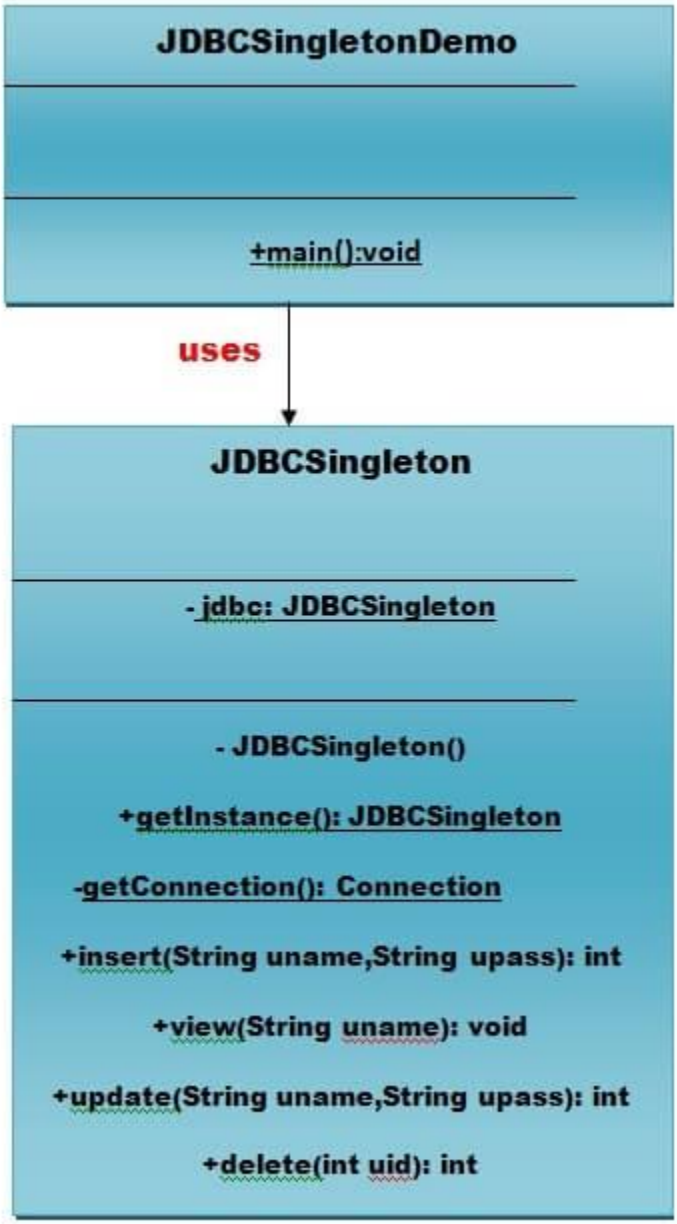
If singleton class is Serializable, you can serialize the singleton instance. Once it is serialized, you can deserialize it but it will not return the singleton object.

To resolve this issue, you need to override the **readResolve() method** that enforces the singleton. It is called just after the object is deserialized. It returns the singleton object.

```
1.  public class A implements Serializable {
2.        //your code of singleton
3.        protected Object readResolve() {
4.           return getA();
5.        }
6.
7.     }
```

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

```
1. import java.io.BufferedReader;
2. import java.io.IOException;
```

```
3.  import java.io.InputStreamReader;
4.  import java.sql.Connection;
5.  import java.sql.DriverManager;
6.  import java.sql.PreparedStatement;
7.  import java.sql.ResultSet;
8.  import java.sql.SQLException;
9.
10. class JDBCSingleton {
11.     //Step 1
12.      // create a JDBCSingleton class.
13.     //static member holds only one instance of the JDBCSingleton class.
14.
15.         private static JDBCSingleton jdbc;
16.
17.     //JDBCSingleton prevents the instantiation from any other class.
18.       private JDBCSingleton() {  }
19.
20.    //Now we are providing gloabal point of access.
21.         public static JDBCSingleton getInstance() {
22.                                 if (jdbc==null)
23.                               {
24.                                       jdbc=new  JDBCSingleton();
25.                               }
26.                     return jdbc;
27.           }
28.
29.   // to get the connection from methods like insert, view etc.
30.         private static Connection getConnection()throws ClassNotFoundException, SQLException
31.         {
32.
33.             Connection con=null;
34.             Class.forName("com.mysql.jdbc.Driver");
35.             con= DriverManager.getConnection("jdbc:mysql://localhost:3306/ashwanirajput", "roo
    t", "ashwani");
36.             return con;
37.
38.         }
39.
40. //to insert the record into the database
41.         public int insert(String name, String pass) throws SQLException
42.         {
43.             Connection c=null;
44.
45.             PreparedStatement ps=null;
46.
47.             int recordCounter=0;
48.
49.             try {
50.
51.                     c=this.getConnection();
52.                     ps=c.prepareStatement("insert into userdata(uname,upassword)values(?,?)");
```

```
53.                 ps.setString(1, name);
54.                 ps.setString(2, pass);
55.                 recordCounter=ps.executeUpdate();
56.
57.             } catch (Exception e) { e.printStackTrace(); } finally{
58.                 if (ps!=null){
59.                   ps.close();
60.                }if(c!=null){
61.                   c.close();
62.                }
63.             }
64.            return recordCounter;
65.         }
66.
67.//to view the data from the database
68.      public  void view(String name) throws SQLException
69.      {
70.              Connection con = null;
71.        PreparedStatement ps = null;
72.        ResultSet rs = null;
73.
74.              try {
75.
76.                    con=this.getConnection();
77.                    ps=con.prepareStatement("select * from userdata where uname=?");
78.                    ps.setString(1, name);
79.                    rs=ps.executeQuery();
80.                    while (rs.next()) {
81.                            System.out.println("Name= "+rs.getString(2)+"\t"+"Paasword= "+rs.ge
    tString(3));
82.
83.                    }
84.
85.         } catch (Exception e) { System.out.println(e);}
86.         finally{
87.                 if(rs!=null){
88.                    rs.close();
89.                 }if (ps!=null){
90.                   ps.close();
91.                }if(con!=null){
92.                   con.close();
93.                }
94.              }
95.      }
96.
97.     // to update the password for the given username
98.      public int update(String name, String password) throws SQLException  {
99.             Connection c=null;
100.                   PreparedStatement ps=null;
101.
102.                   int recordCounter=0;
```

```
103.                 try {
104.                       c=this.getConnection();
105.                       ps=c.prepareStatement(" update userdata set upassword=? where uname
    ='"+name+"' ");
106.                       ps.setString(1, password);
107.                       recordCounter=ps.executeUpdate();
108.                 } catch (Exception e) {  e.printStackTrace(); } finally{
109.
110.                    if (ps!=null){
111.                       ps.close();
112.                    }if(c!=null){
113.                       c.close();
114.                    }
115.                  }
116.                return recordCounter;
117.              }
118.
119.       // to delete the data from the database
120.              public int delete(int userid) throws SQLException{
121.                  Connection c=null;
122.                  PreparedStatement ps=null;
123.                  int recordCounter=0;
124.                  try {
125.                         c=this.getConnection();
126.                       ps=c.prepareStatement(" delete from userdata where uid='"+userid+"' ")
    ;
127.                       recordCounter=ps.executeUpdate();
128.                  } catch (Exception e) { e.printStackTrace(); }
129.                  finally{
130.                  if (ps!=null){
131.                       ps.close();
132.                 }if(c!=null){
133.                       c.close();
134.                  }
135.               }
136.                 return recordCounter;
137.              }
138.        }// End of JDBCSingleton class
```

File: JDBCSingletonDemo.java

```
1.  import java.io.BufferedReader;
2.  import java.io.IOException;
3.  import java.io.InputStreamReader;
4.  import java.sql.Connection;
5.  import java.sql.DriverManager;
6.  import java.sql.PreparedStatement;
7.  import java.sql.ResultSet;
8.  import java.sql.SQLException;
9.  class JDBCSingletonDemo{
10.    static int count=1;
11.    static int  choice;
12.    public static void main(String[] args) throws IOException {
```

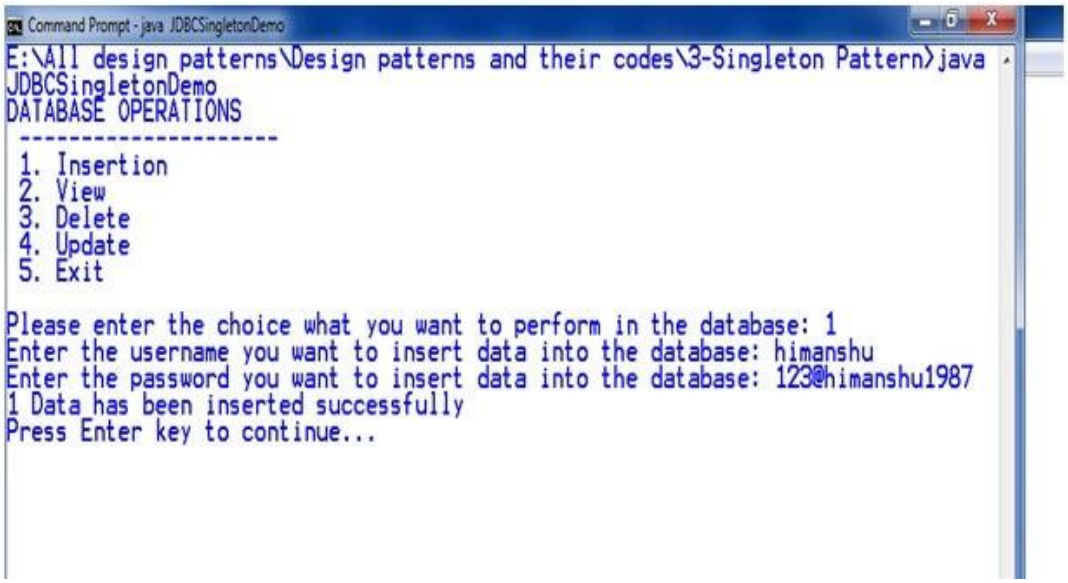
```
13.
14.       JDBCSingleton jdbc= JDBCSingleton.getInstance();
15.
16.
17.       BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
18.   do{
19.       System.out.println("DATABASE OPERATIONS");
20.       System.out.println(" -------------------- ");
21.       System.out.println(" 1. Insertion ");
22.       System.out.println(" 2. View      ");
23.       System.out.println(" 3. Delete    ");
24.       System.out.println(" 4. Update    ");
25.       System.out.println(" 5. Exit      ");
26.
27.       System.out.print("\n");
28.       System.out.print("Please enter the choice what you want to perform in the database: ");
29.
30.       choice=Integer.parseInt(br.readLine());
31.       switch(choice) {
32.
33.         case 1:{
34.                System.out.print("Enter the username you want to insert data into the database: "
    );
35.                String username=br.readLine();
36.                System.out.print("Enter the password you want to insert data into the database: "
    );
37.                String password=br.readLine();
38.
39.                try {
40.                      int i= jdbc.insert(username, password);
41.                      if (i>0) {
42.                      System.out.println((count++) + " Data has been inserted successfully");
43.                      }else{
44.                         System.out.println("Data has not been inserted ");
45.                      }
46.
47.                    } catch (Exception e) {
48.                      System.out.println(e);
49.                    }
50.
51.                 System.out.println("Press Enter key to continue...");
52.                 System.in.read();
53.
54.               }//End of case 1
55.               break;
56.          case 2:{
57.                System.out.print("Enter the username : ");
58.                String username=br.readLine();
59.
60.                try  {
61.                      jdbc.view(username);
```

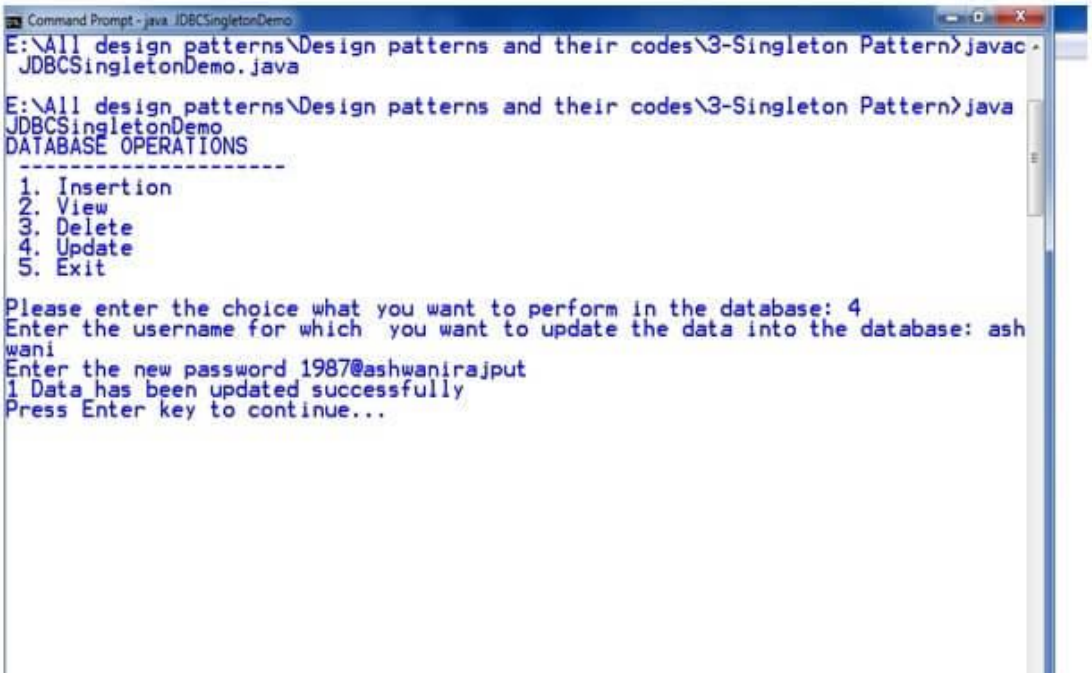
```
62.                 } catch (Exception e) {
63.                  System.out.println(e);
64.                 }
65.               System.out.println("Press Enter key to continue...");
66.               System.in.read();
67.
68.              }//End of case 2
69.             break;
70.          case 3:{
71.               System.out.print("Enter the userid,  you want to delete: ");
72.               int userid=Integer.parseInt(br.readLine());
73.
74.               try {
75.                     int i= jdbc.delete(userid);
76.                     if (i>0) {
77.                     System.out.println((count++) + " Data has been deleted successfully");
78.                     }else{
79.                         System.out.println("Data has not been deleted");
80.                     }
81.
82.                  } catch (Exception e) {
83.                   System.out.println(e);
84.                  }
85.               System.out.println("Press Enter key to continue...");
86.               System.in.read();
87.
88.              }//End of case 3
89.             break;
90.          case 4:{
91.               System.out.print("Enter the username,  you want to update: ");
92.               String username=br.readLine();
93.               System.out.print("Enter the new password ");
94.               String password=br.readLine();
95.
96.               try {
97.                     int i= jdbc.update(username, password);
98.                     if (i>0) {
99.                     System.out.println((count++) + " Data has been updated successfully");
100.                          }
101.
102.                        } catch (Exception e) {
103.                          System.out.println(e);
104.                        }
105.                     System.out.println("Press Enter key to continue...");
106.                     System.in.read();
107.
108.                    }// end of case 4
109.                   break;
110.
111.               default:
112.                     return;
```

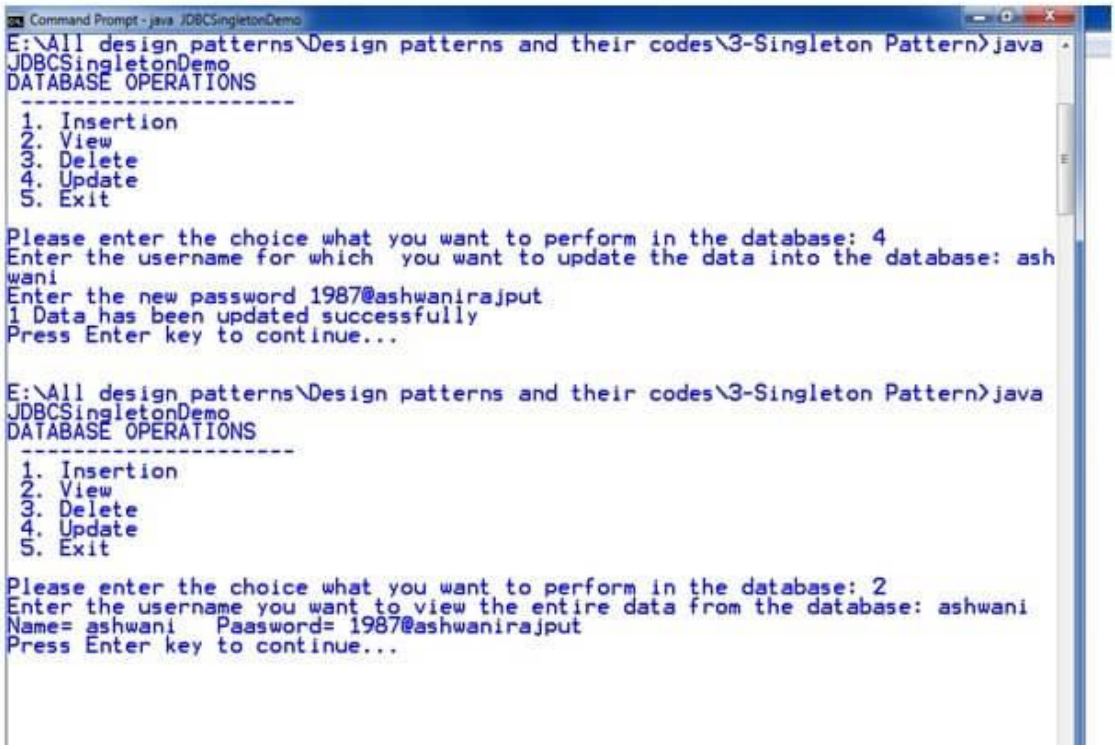
```
113.             }
114.
115.           } while (choice!=4);
116.         }
117.       }
```

download this Singleton Pattern Example

Output


Command Prompt - java JDBCSingletonDemo
E:\All design patterns\Design patterns and their codes\3-Singleton Pattern>java JDBCSingletonDemo
DATABASE OPERATIONS
1. Insertion
2. View
3. Delete
4. Update
5. Exit
Please enter the choice what you want to perform in the database: 1
Enter the username you want to insert data into the database: himanshu
Enter the password you want to insert data into the database: 123@himanshu1987
1 Data has been inserted successfully
Press Enter key to continue...


Command Prompt - java JDBCSingletonDemo
E:\All design patterns\Design patterns and their codes\3-Singleton Pattern>javac JDBCSingletonDemo.java
E:\All design patterns\Design patterns and their codes\3-Singleton Pattern>java JDBCSingletonDemo
DATABASE OPERATIONS
1. Insertion
2. View
3. Delete
4. Update
5. Exit
Please enter the choice what you want to perform in the database: 4
Enter the username for which you want to update the data into the database: ashwani
Enter the new password 1987@ashwanirajput
1 Data has been updated successfully
Press Enter key to continue...



Prototype Design Pattern
Prototype Pattern says that **cloning of an existing object instead of creating new one and can also be customized as per the requirement**.
This pattern should be followed, if the cost of creating a new object is expensive and resource intensive.

Advantage of Prototype Pattern

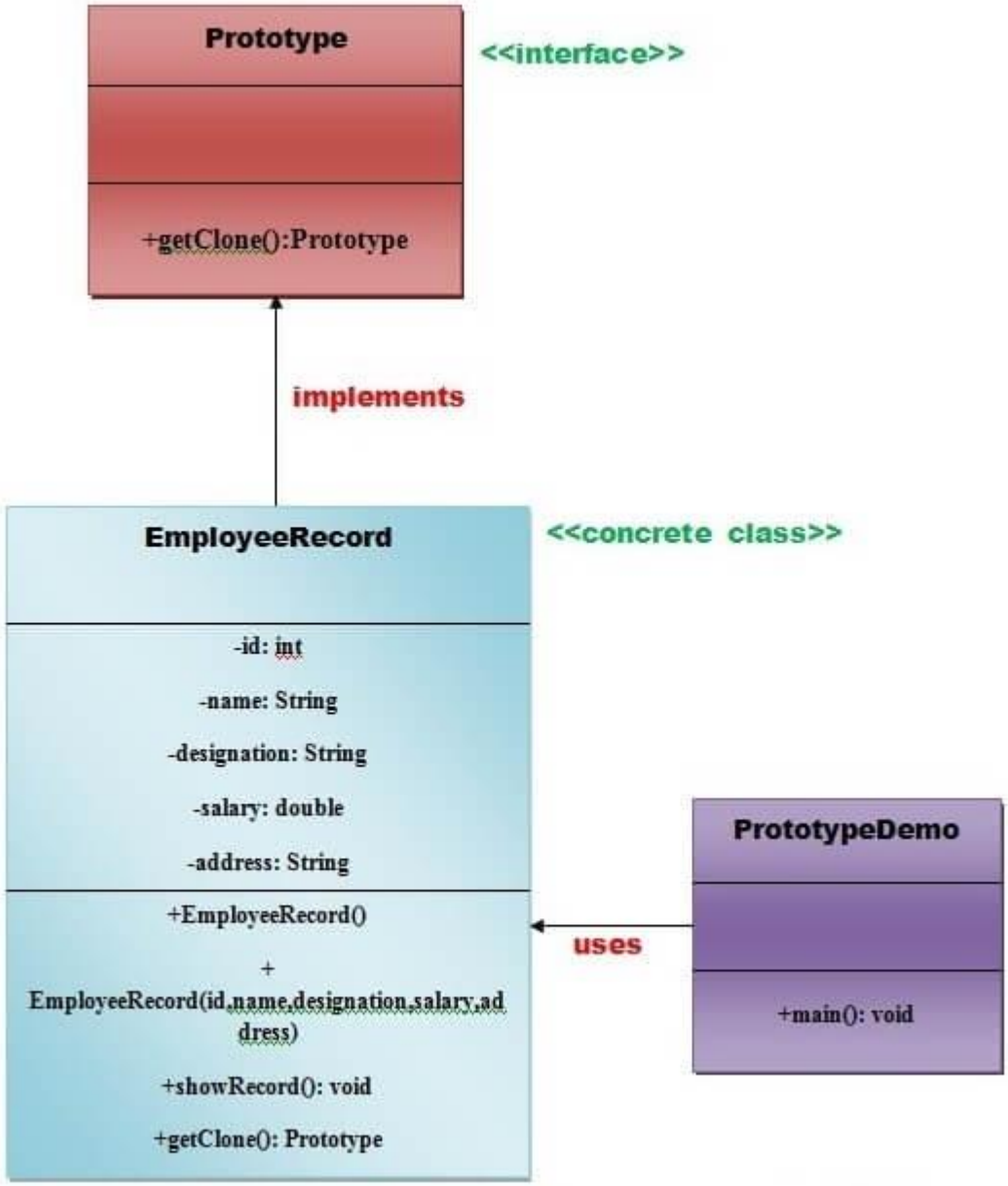
The main advantages of prototype pattern are as follows:

- It reduces the need of sub-classing.
- It hides complexities of creating objects.
- The clients can get new objects without knowing which type of object it will be.
- It lets you add or remove objects at runtime.

Usage of Prototype Pattern

- When the classes are instantiated at runtime.
- When the cost of creating an object is expensive or complicated.
- When you want to keep the number of classes in an application minimum.
- When the client application needs to be unaware of object creation and representation.

UML for Prototype Pattern



- We are going to create **an interface Prototype** that contains a method **getClone()** of **Prototype type.**
- Then, we create **a concrete class EmployeeRecord** which implements **Prototype interface** that does the cloning of EmployeeRecord object.
- **PrototypeDemo class** will uses this concrete class **EmployeeRecord.**

Example of Prototype Design Pattern

Let's see the example of prototype design pattern.
File: Prototype.java

```
1. interface Prototype {
2.
3.      public Prototype getClone();
```

```
4.
5.  }//End of Prototype interface.
```

File: EmployeeRecord.java

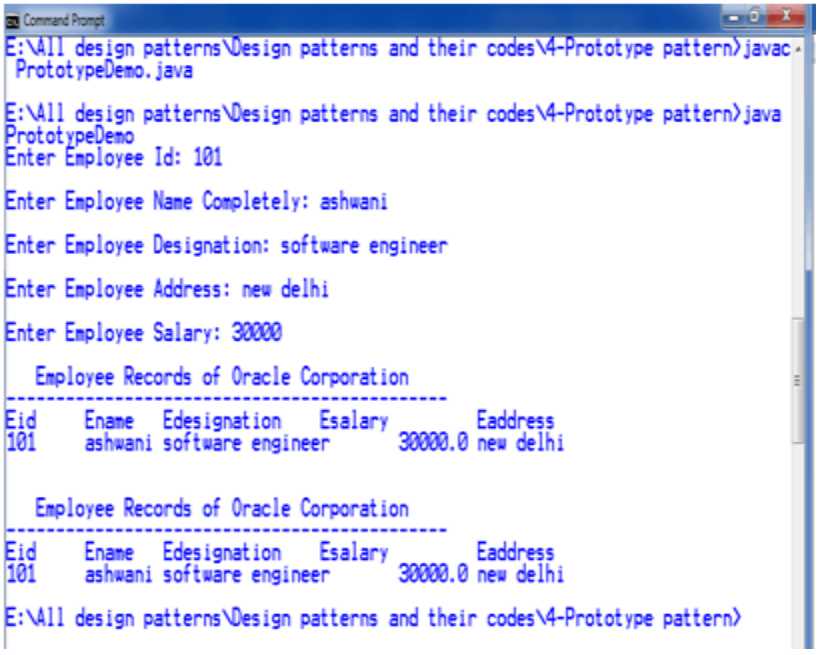
```
1.  class EmployeeRecord implements Prototype{
2.
3.     private int id;
4.     private String name, designation;
5.     private double salary;
6.     private String address;
7.
8.     public EmployeeRecord(){
9.             System.out.println("   Employee Records of Oracle Corporation ");
10.            System.out.println("-----------------------------------------");
11.            System.out.println("Eid"+"\t"+"Ename"+"\t"+"Edesignation"+"\t"+"Esalary"+"\t\t"+"Ea
    ddress");
12.
13. }
14.
15.  public  EmployeeRecord(int id, String name, String designation, double salary, String address) {

16.
17.        this();
18.        this.id = id;
19.        this.name = name;
20.        this.designation = designation;
21.        this.salary = salary;
22.        this.address = address;
23.     }
24.
25.   public void showRecord(){
26.
27.        System.out.println(id+"\t"+name+"\t"+designation+"\t"+salary+"\t"+address);
28.    }
29.
30.    @Override
31.    public Prototype getClone() {
32.
33.        return new EmployeeRecord(id,name,designation,salary,address);
34.    }
35. }//End of EmployeeRecord class.
```

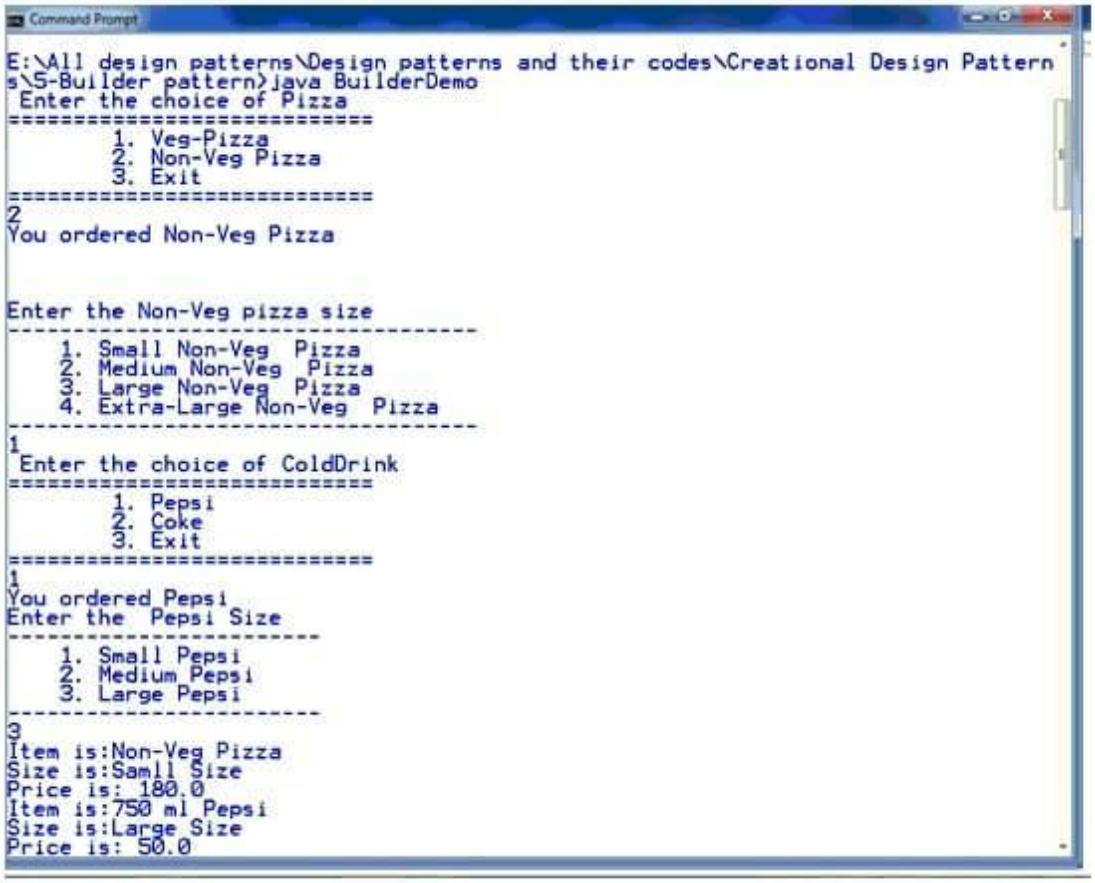
File: PrototypeDemo.java

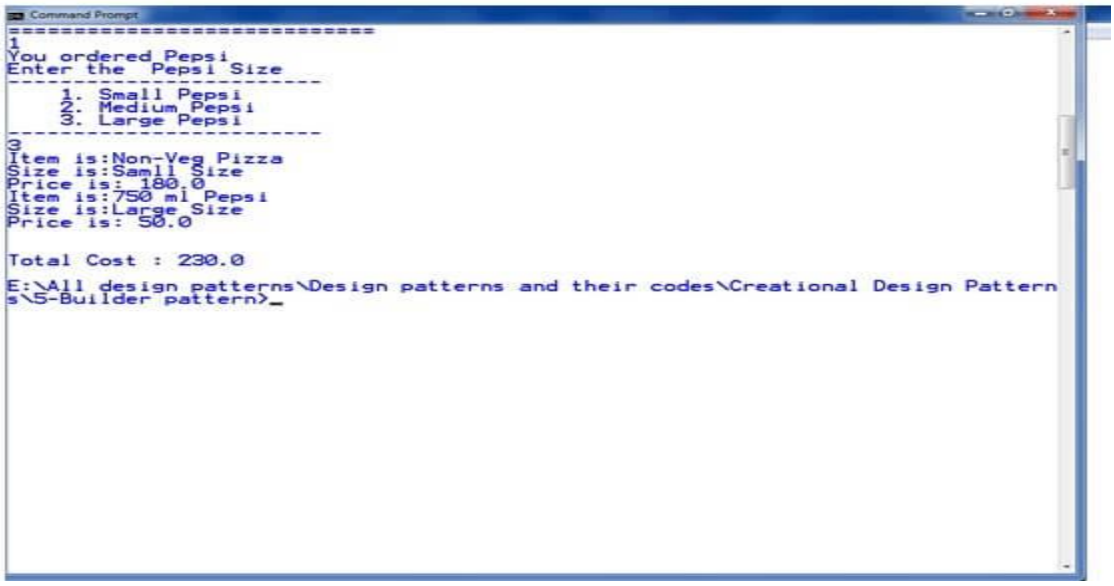
```
1.  import java.io.BufferedReader;
2.  import java.io.IOException;
3.  import java.io.InputStreamReader;
4.
5.  class PrototypeDemo{
6.       public static void main(String[] args) throws IOException {
7.
8.          BufferedReader br =new BufferedReader(new InputStreamReader(System.in));
9.          System.out.print("Enter Employee Id: ");
10.         int eid=Integer.parseInt(br.readLine());
```

```
11.       System.out.print("\n");
12.
13.       System.out.print("Enter Employee Name: ");
14.       String ename=br.readLine();
15.       System.out.print("\n");
16.
17.       System.out.print("Enter Employee Designation: ");
18.       String edesignation=br.readLine();
19.       System.out.print("\n");
20.
21.       System.out.print("Enter Employee Address: ");
22.       String eaddress=br.readLine();
23.       System.out.print("\n");
24.
25.       System.out.print("Enter Employee Salary: ");
26.       double esalary= Double.parseDouble(br.readLine());
27.       System.out.print("\n");
28.
29.       EmployeeRecord e1=new EmployeeRecord(eid,ename,edesignation,esalary,eaddress);
30.
31.       e1.showRecord();
32.       System.out.println("\n");
33.       EmployeeRecord e2=(EmployeeRecord) e1.getClone();
34.       e2.showRecord();
35.    }
36. }//End of the ProtoypeDemo class.
```

Output







- **Structural Design Pattern**

§ Decorator Pattern
§ Facade Pattern

Structural design patterns
Structural design patterns are concerned with how classes and objects can be composed, to form larger structures.
The structural design patterns **simplifies the structure by identifying the relationships**.
These patterns focus on, how the classes inherit from each other and how they are composed from other classes.

Types of structural design patterns

There are following 7 types of structural design patterns.

1. Adapter Pattern
   Adapting an interface into another according to client expectation.
2. Bridge Pattern
   Separating abstraction (interface) from implementation.
3. Composite Pattern
   Allowing clients to operate on hierarchy of objects.
4. Decorator Pattern
   Adding functionality to an object dynamically.
5. Facade Pattern
   Providing an interface to a set of interfaces.
6. Flyweight Pattern
   Reusing an object by sharing it.
7. proxy Pattern
   Representing another object.

Decorator Pattern

1. Decorator Design Pattern
2. Advantage of Decorator DP
3. Usage of Decorator DP
4. UML of Decorator DP
5. Example of Decorator DP

A Decorator Pattern says that just **"attach a flexible additional responsibilities to an object dynamically".**
In other words, The Decorator Pattern uses composition instead of inheritance to extend the functionality of an object at runtime.
The Decorator Pattern is also known as **Wrapper.**

Advantage of Decorator Pattern

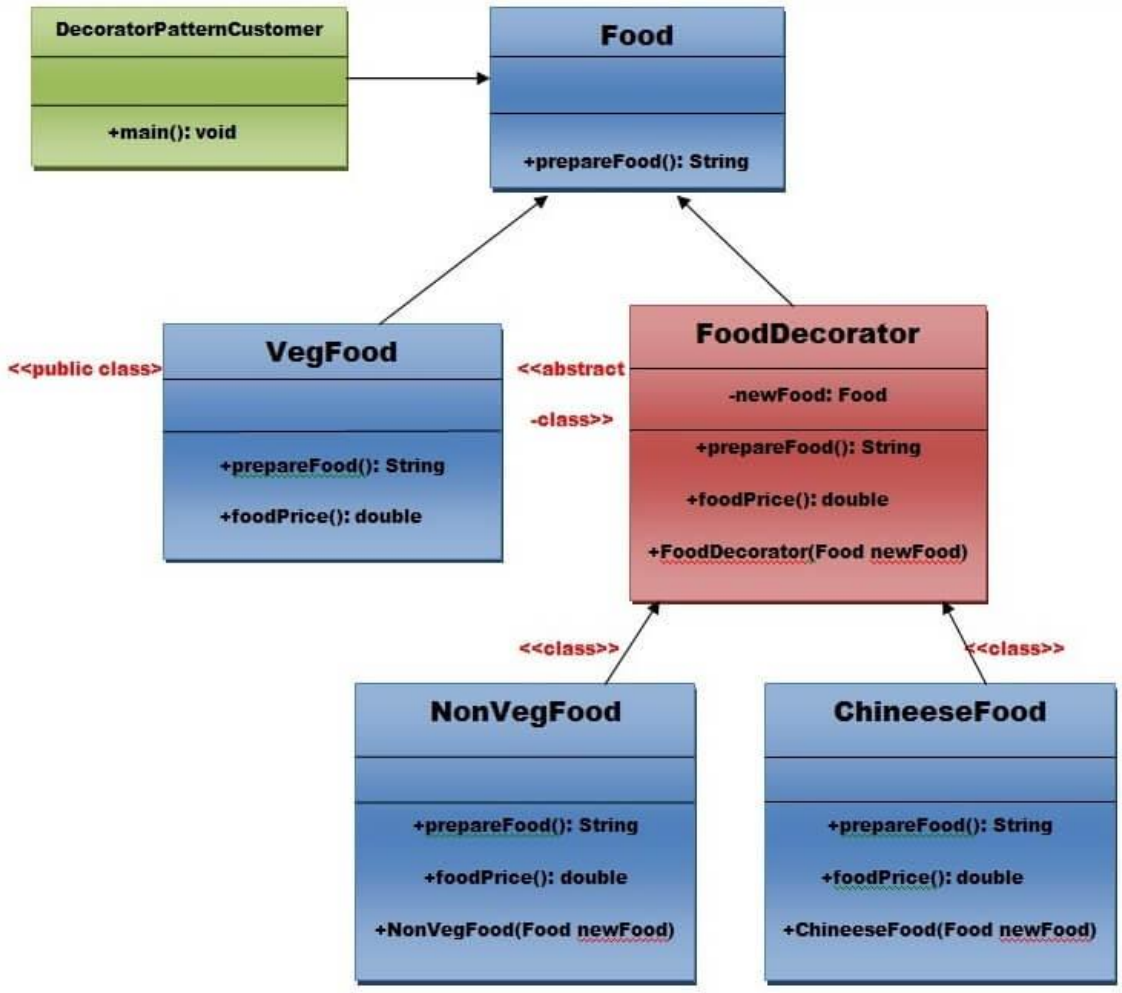
- It provides greater flexibility than static inheritance.
- It enhances the extensibility of the object, because changes are made by coding new classes.
- It simplifies the coding by allowing you to develop a series of functionality from targeted classes instead of coding all of the behavior into the object.

Usage of Decorator Pattern

It is used:

- When you want to transparently and dynamically add responsibilities to objects without affecting other objects.
- When you want to add responsibilities to an object that you may want to change in future.
- Extending functionality by sub-classing is no longer practical.

UML for Decorator Pattern:



Step 1:**Create a Food interface.**

```
1. public interface Food {
2.     public String prepareFood();
3.     public double foodPrice();
4. }// End of the Food interface.
```

Step 2: Create a **VegFood** class that will implements the **Food** interface and override its all methods.

File: VegFood.java

```
1. public class VegFood implements Food {
2.     public String prepareFood(){
3.         return "Veg Food";
4.     }
5.
6.         public double foodPrice(){
```

```
7.          return 50.0;
8.      }
9.  }
```

Step 3:Create a FoodDecorator abstract class that will implements the Food interface and override it's all methods and it has the ability to decorate some more foods.

File: FoodDecorator.java

```
1.  public abstract class FoodDecorator implements Food{
2.      private Food newFood;
3.      public FoodDecorator(Food newFood)  {
4.          this.newFood=newFood;
5.      }
6.      @Override
7.      public String prepareFood(){
8.          return newFood.prepareFood();
9.      }
10.     public double foodPrice(){
11.         return newFood.foodPrice();
12.     }
13. }
```

Step 4:Create a **NonVegFood concrete** class that will extend the **FoodDecorator** class and override it's all methods.

File: NonVegFood.java

```
1.  public class NonVegFood extends FoodDecorator{
2.      public NonVegFood(Food newFood) {
3.          super(newFood);
4.      }
5.      public String prepareFood(){
6.          return super.prepareFood() +" With Roasted Chiken and Chiken Curry  ";
7.      }
8.      public double foodPrice()   {
9.          return super.foodPrice()+150.0;
10.     }
11. }
```

Step 5:Create a **ChineeseFood** concrete class that will extend the **FoodDecorator** class and override it's all methods.

File: ChineeseFood.java

```
1.  public class ChineeseFood extends FoodDecorator{
2.    public ChineeseFood(Food newFood)    {
3.          super(newFood);
4.    }
5.      public String prepareFood(){
6.          return super.prepareFood() +" With Fried Rice and Manchurian  ";
7.      }
8.      public double foodPrice()   {
9.          return super.foodPrice()+65.0;
10.         }
11. }
```

Step 6:Create a **DecoratorPatternCustomer** class that will use Food interface to use which type of food customer wants means (Decorates).

File: DecoratorPatternCustomer.java

```
1.  import java.io.BufferedReader;
```

```
2.  import java.io.IOException;
3.  import java.io.InputStreamReader;
4.  public class DecoratorPatternCustomer {
5.      private static int  choice;
6.      public static void main(String args[]) throws NumberFormatException, IOException    {
7.         do{
8.          System.out.print("========= Food Menu ============ \n");
9.          System.out.print("          1. Vegetarian Food.   \n");
10.         System.out.print("          2. Non-Vegetarian Food.\n");
11.         System.out.print("          3. Chineese Food.        \n");
12.         System.out.print("          4. Exit                  \n");
13.         System.out.print("Enter your choice: ");
14.         BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
15.         choice=Integer.parseInt(br.readLine());
16.         switch (choice) {
17.         case 1:{
18.                 VegFood vf=new VegFood();
19.              System.out.println(vf.prepareFood());
20.              System.out.println( vf.foodPrice());
21.            }
22.            break;
23.
24.               case 2:{
25.               Food f1=new NonVegFood((Food) new VegFood());
26.                  System.out.println(f1.prepareFood());
27.               System.out.println( f1.foodPrice());
28.        }
29.            break;
30.      case 3:{
31.             Food f2=new ChineeseFood((Food) new VegFood());
32.                    System.out.println(f2.prepareFood());
33.                  System.out.println( f2.foodPrice());
34.             }
35.            break;
36.
37.         default:{
38.            System.out.println("Other than these no food available");
39.         }
40.     return;
41.      }//end of switch
42.
43. }while(choice!=4);
44.     }
45. }
```

Output

```
1.  ========= Food Menu ============
2.            1. Vegetarian Food.
3.            2. Non-Vegetarian Food.
4.            3. Chineese Food.
```

```
5.          4. Exit
6.  Enter your choice: 1
7.  Veg Food
8.  50.0
9.  ========== Food Menu ============
10.         1. Vegetarian Food.
11.         2. Non-Vegetarian Food.
12.         3. Chineese Food.
13.         4. Exit
14. Enter your choice: 2
15. Veg Food With Roasted Chiken and Chiken Curry
16. 200.0
17. ========== Food Menu ============
18.         1. Vegetarian Food.
19.         2. Non-Vegetarian Food.
20.         3. Chineese Food.
21.         4. Exit
22. Enter your choice: 3
23. Veg Food With Fried Rice and Manchurian
24. 115.0
25. ========== Food Menu ============
26.         1. Vegetarian Food.
27.         2. Non-Vegetarian Food.
28.         3. Chineese Food.
29.         4. Exit
30. Enter your choice: 4
31. Other than these no food available
```

Next →← Prev

Facade Pattern

1. Facade Design Pattern
2. Advantage of Facade DP
3. Usage of Facade DP
4. UML of Facade DP
5. Example of Facade DP

A Facade Pattern says that just **"just provide a unified and simplified interface to a set of interfaces in a subsystem, therefore it hides the complexities of the subsystem from the client".**

In other words, Facade Pattern describes a higher-level interface that makes the sub-system easier to use.

Practically, **every Abstract Factory** is a type of **Facade.**

Advantage of Facade Pattern

- It shields the clients from the complexities of the sub-system components.
- It promotes loose coupling between subsystems and its clients.

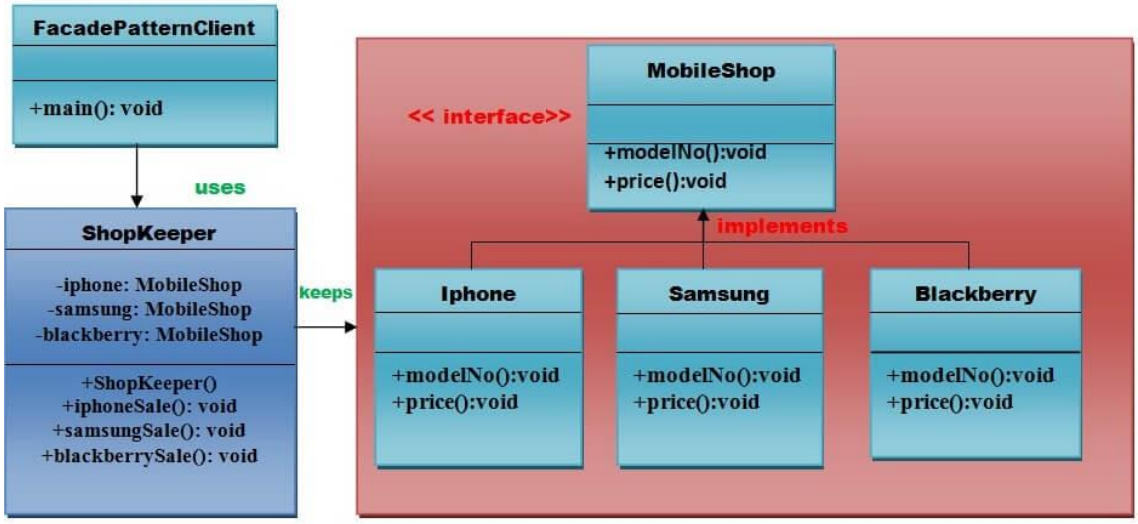
Usage of Facade Pattern:

It is used:

- When you want to provide simple interface to a complex sub-system.
- When several dependencies exist between clients and the implementation classes of an abstraction.

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

UML for Facade Pattern:



Implementation of above UML:
Step 1

Create a **MobileShop** interface.
File: MobileShop.java

```
1. public interface MobileShop {
2.     public void modelNo();
3.     public void price();
4. }
```

Step 2

Create a **Iphone** implementation class that will implement **Mobileshop** interface.
File: Iphone.java

```
1. public class Iphone implements MobileShop {
2.     @Override
3.     public void modelNo() {
4.         System.out.println(" Iphone 6 ");
5.     }
6.     @Override
7.     public void price() {
8.     System.out.println(" Rs 65000.00 ");
```

```
    }
}
```

Step 3

Create a **Samsung** implementation class that will implement **Mobileshop** interface.
File: Samsung.java

```
public 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 ");
    }
}
```

Step 4

Create a **Blackberry** implementation class that will implement **Mobileshop** interface .
File: Blackberry.java

```
public class Blackberry implements MobileShop {
    @Override
    public void modelNo() {
    System.out.println(" Blackberry Z10 ");
    }
    @Override
    public void price() {
        System.out.println(" Rs 55000.00 ");
    }
}
```

Step 5

Create a **ShopKeeper** concrete class that will use **MobileShop** interface.
File: ShopKeeper.java

```
public 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();
```

```
14.    }
15.        public void samsungSale(){
16.        samsung.modelNo();
17.        samsung.price();
18.    }
19.   public void blackberrySale(){
20.    blackberry.modelNo();
21.    blackberry.price();
22.        }
23. }
```

Step 6

Now, Creating a **client** that can purchase the mobiles from **MobileShop** through **ShopKeeper.**
File: FacadePatternClient.java

```
1.  import java.io.BufferedReader;
2.  import java.io.IOException;
3.  import java.io.InputStreamReader;
4.
5.  public class FacadePatternClient {
6.      private static int  choice;
7.      public static void main(String args[]) throws NumberFormatException, IOException{
8.          do{
9.              System.out.print("========== Mobile Shop ============ \n");
10.             System.out.print("          1. IPHONE.            \n");
11.             System.out.print("          2. SAMSUNG.             \n");
12.             System.out.print("          3. BLACKBERRY.            \n");
13.             System.out.print("          4. Exit.                \n");
14.             System.out.print("Enter your choice: ");
15.
16.             BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
17.             choice=Integer.parseInt(br.readLine());
18.             ShopKeeper sk=new ShopKeeper();
19.
20.             switch (choice) {
21.             case 1:
22.                {
23.                  sk.iphoneSale();
24.                    }
25.                break;
26.       case 2:
27.                {
28.                  sk.samsungSale();
29.                    }
30.                break;
31.       case 3:
32.                        {
33.                        sk.blackberrySale();
34.                        }
35.                  break;
36.             default:
```

```
37.         {
38.            System.out.println("Nothing You purchased");
39.         }
40.            return;
41.         }
42.
43.      }while(choice!=4);
44.   }
45.}
```

download this example

Output

```
1.  ========== Mobile Shop ============
2.           1. IPHONE.
3.           2. SAMSUNG.
4.           3. BLACKBERRY.
5.           4. Exit.
6.  Enter your choice: 1
7.   Iphone 6
8.   Rs 65000.00
9.  ========== Mobile Shop ============
10.          1. IPHONE.
11.          2. SAMSUNG.
12.          3. BLACKBERRY.
13.          4. Exit.
14. Enter your choice: 2
15.  Samsung galaxy tab 3
16.  Rs 45000.00
17. ========== Mobile Shop ============
18.          1. IPHONE.
19.          2. SAMSUNG.
20.          3. BLACKBERRY.
21.          4. Exit.
22. Enter your choice: 3
23.  Blackberry Z10
24.  Rs 55000.00
25. ========== Mobile Shop ============
26.          1. IPHONE.
27.          2. SAMSUNG.
28.          3. BLACKBERRY.
29.          4. Exit.
30. Enter your choice: 4
31. Nothing You purchased
```

- **Behavioral Design Pattern**
  - **Chain of Responsibility Pattern**
  - **Iterator Pattern**

Behavioral Design Patterns
Behavioral design patterns are concerned with **the interaction and responsibility of objects.**
In these design patterns,**the interaction between the objects should be in such a way that they can easily talk to each other and still should be loosely coupled.**
That means the implementation and the client should be loosely coupled in order to avoid hard coding and dependencies.

There are 12 types of structural design patterns:

1. Chain of Responsibility Pattern
2. Command Pattern
3. Interpreter Pattern
4. Iterator Pattern
5. Mediator Pattern
6. Memento Pattern
7. Observer Pattern
8. State Pattern
9. Strategy Pattern
10. Template Pattern
11. Visitor Pattern
12. Null Object

Chain Of Responsibility Pattern

1. Chain Of Responsibility Pattern
2. Advantage of Chain Of Responsibility DP
3. Usage of Chain Of Responsibility DP
4. UML of Chain Of Responsibility DP
5. Example of Chain Of Responsibility DP

In chain of responsibility, sender sends a request to a chain of objects. The request can be handled by any object in the chain.
A Chain of Responsibility Pattern says that just **"avoid coupling the sender of a request to its receiver by giving multiple objects a chance to handle the request".** For example, an ATM uses the Chain of Responsibility design pattern in money giving process.
In other words, we can say that normally each receiver contains reference of another receiver. If one object cannot handle the request then it passes the same to the next receiver and so on.

Advantage of Chain of Responsibility Pattern

- It reduces the coupling.
- It adds flexibility while assigning the responsibilities to objects.
- It allows a set of classes to act as one; events produced in one class can be sent to other handler classes with the help of composition.

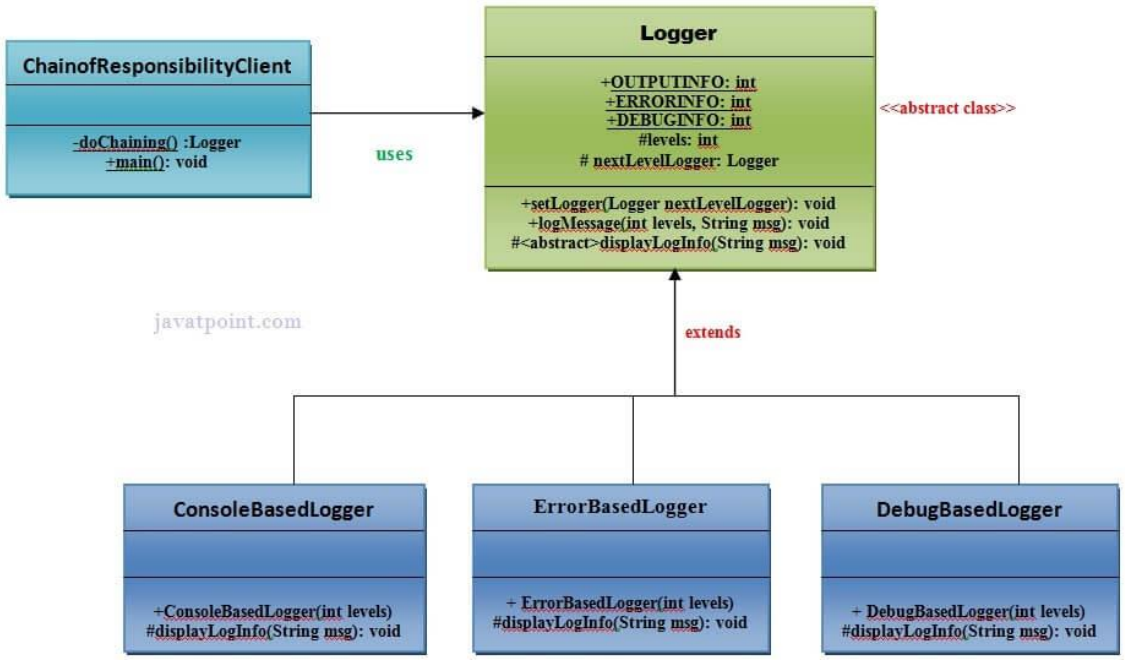
Usage of Chain of Responsibility Pattern:

It is used:

- When more than one object can handle a request and the handler is unknown.
- When the group of objects that can handle the request must be specified in dynamic way.

Example of Chain of Responsibility Pattern
Let's understand the example of Chain of Responsibility Pattern by the above UML diagram.

UML for Chain of Responsibility Pattern:



Implementation of above UML:
Step 1

Create a **Logger** abstract class.

```
1.  public abstract class Logger {
2.      public static int OUTPUTINFO=1;
3.      public static int ERRORINFO=2;
4.      public static int DEBUGINFO=3;
5.      protected int levels;
6.      protected Logger nextLevelLogger;
7.      public void setNextLevelLogger(Logger nextLevelLogger) {
8.          this.nextLevelLogger = nextLevelLogger;
9.      }
10.         public void logMessage(int levels, String msg){
11.         if(this.levels<=levels){
12.             displayLogInfo(msg);
13.         }
14.         if (nextLevelLogger!=null) {
15.             nextLevelLogger.logMessage(levels, msg);
16.         }
17.     }
18.     protected abstract void displayLogInfo(String msg);
19. }
```

Step 2

Create a **ConsoleBasedLogger** class.
File: ConsoleBasedLogger.java

```
1. public class ConsoleBasedLogger extends Logger {
2.     public ConsoleBasedLogger(int levels) {
3.         this.levels=levels;
4.     }
5.     @Override
6.     protected void displayLogInfo(String msg) {
7.         System.out.println("CONSOLE LOGGER INFO: "+msg);
8.     }
9. }
```

Step 3

Create a **DebugBasedLogger** class.
File: DebugBasedLogger.java

```
1. public class DebugBasedLogger extends Logger {
2.     public DebugBasedLogger(int levels) {
3.         this.levels=levels;
4.     }
5.     @Override
6.     protected void displayLogInfo(String msg) {
7.         System.out.println("DEBUG LOGGER INFO: "+msg);
8.     }
9. }// End of the DebugBasedLogger class.
```

Step 4

Create a **ErrorBasedLogger** class.
File: ErrorBasedLogger.java

```
1. public class ErrorBasedLogger extends Logger {
2.     public ErrorBasedLogger(int levels) {
3.         this.levels=levels;
4.     }
5.     @Override
6.     protected void displayLogInfo(String msg) {
7.         System.out.println("ERROR LOGGER INFO: "+msg);
8.     }
9. }// End of the ErrorBasedLogger class.
```

Step 5

Create a **ChainOfResponsibilityClient** class.
File: ChainofResponsibilityClient.java

```
1. public class ChainofResponsibilityClient {
2.     private static Logger doChaining(){
3.           Logger consoleLogger = new ConsoleBasedLogger(Logger.OUTPUTINFO);
4.
```

```
5.          Logger errorLogger = new ErrorBasedLogger(Logger.ERRORINFO);
6.          consoleLogger.setNextLevelLogger(errorLogger);
7.
8.          Logger debugLogger = new DebugBasedLogger(Logger.DEBUGINFO);
9.          errorLogger.setNextLevelLogger(debugLogger);
10.
11.         return consoleLogger;
12.         }
13.         public static void main(String args[]){
14.         Logger chainLogger= doChaining();
15.
16.            chainLogger.logMessage(Logger.OUTPUTINFO, "Enter the sequence of values ");
17.            chainLogger.logMessage(Logger.ERRORINFO, "An error is occured now");
18.            chainLogger.logMessage(Logger.DEBUGINFO, "This was the error now debugging is co
    mpeled");
19.            }
20. }
```

download this example

Output

```
1. bilityClient
2. CONSOLE LOGGER INFO: Enter the sequence of values
3. CONSOLE LOGGER INFO: An error is occured now
4. ERROR LOGGER INFO: An error is occured now
5. CONSOLE LOGGER INFO: This was the error now debugging is compeled
6. ERROR LOGGER INFO: This was the error now debugging is compeled
7. DEBUG LOGGER INFO: This was the error now debugging is compeled
```

Iterator Pattern

1. Iterator Design Pattern
2. Advantage of Iterator DP
3. Usage of Iterator DP
4. UML of Iterator DP
5. Example of Iterator DP

According to GoF, Iterator Pattern is used **"to access the elements of an aggregate object sequentially without exposing its underlying implementation".**
The Iterator pattern is also known as **Cursor.**
In collection framework, we are now using Iterator that is preferred over Enumeration.

java.util.Iterator interface uses Iterator Design Pattern.

Advantage of Iterator Pattern

- It supports variations in the traversal of a collection.
- It simplifies the interface to the collection.

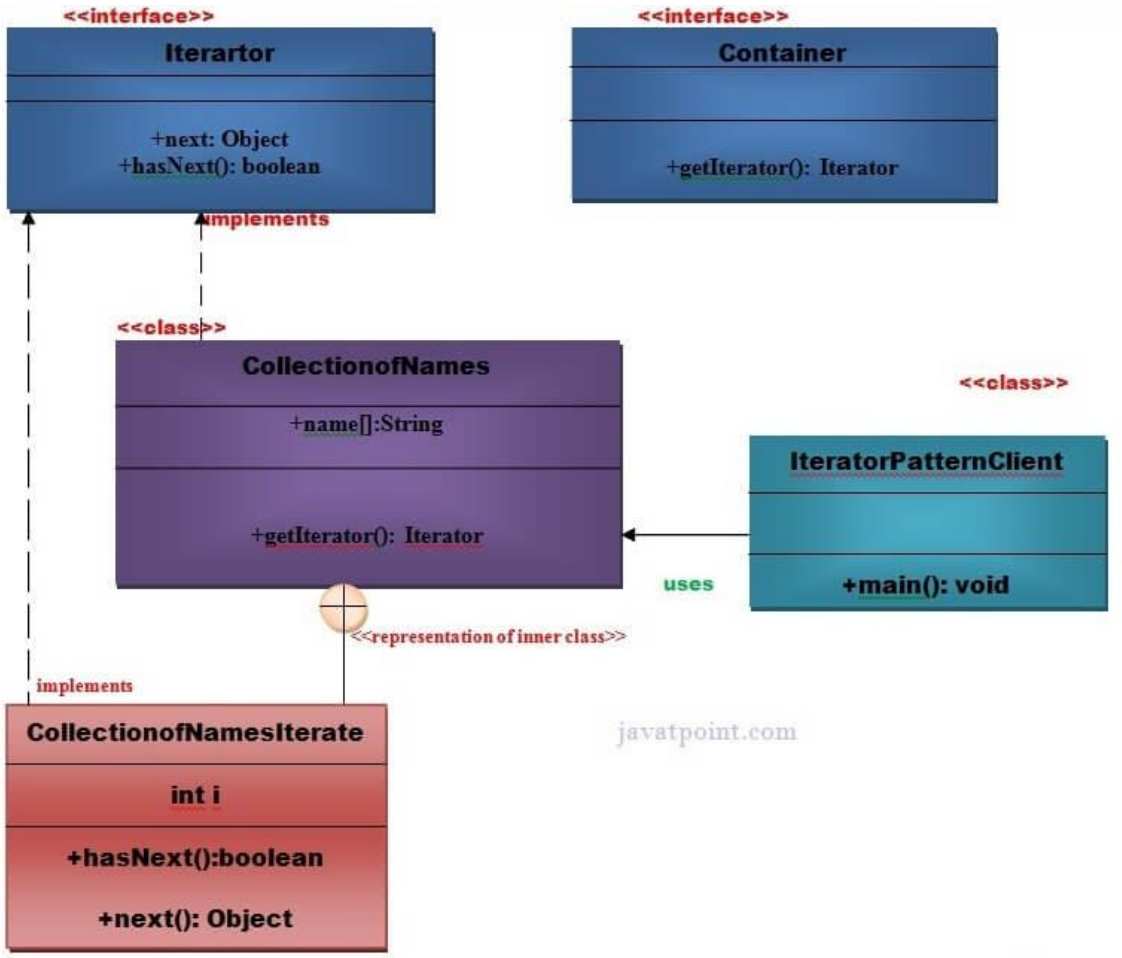
Usage of Iterator Pattern:

It is used:

- When you want to access a collection of objects without exposing its internal representation.
- When there are multiple traversals of objects need to be supported in the collection.

Example of Iterator Pattern
Let's understand the example of iterator pattern pattern by the above UML diagram.

UML for Iterator Pattern:



Implementation of above UML
Step 1

Create a **Iterartor** interface.

```
1. public interface Iterator {
2.     public boolean hasNext();
3.     public Object next();
4. }
```

Step 2

Create a **Container** interface.

```
1. public interface Container {
2.     public Iterator getIterator();
3. }// End of the Iterator interface.
```

Step 3

Create a **CollectionofNames** class that will implement **Container** interface.
File: CollectionofNames.java

```
1. public class CollectionofNames implements Container {
2. public String name[]={"Ashwani Rajput", "Soono Jaiswal","Rishi Kumar","Rahul Mehta","Hemant
    Mishra"};
3.
4. @Override
5.     public Iterator getIterator() {
6.         return new CollectionofNamesIterate() ;
7.     }
8.     private class CollectionofNamesIterate implements Iterator{
9.         int i;
10.        @Override
11.        public boolean hasNext() {
12.            if (i<name.length){
13.                return true;
14.            }
15.            return false;
16.        }
17.        @Override
18.        public Object next() {
19.            if(this.hasNext()){
20.                return name[i++];
21.            }
22.            return null;
23.        }
24.    }
25. }
26. }
```

Step 4

Create a **IteratorPatternDemo** class.
File: IteratorPatternDemo.java

```
1. public class IteratorPatternDemo {
2.     public static void main(String[] args) {
3.          CollectionofNames cmpnyRepository = new CollectionofNames();
4.
5.          for(Iterator iter = cmpnyRepository.getIterator(); iter.hasNext();){
6.              String name = (String)iter.next();
7.              System.out.println("Name : " + name);
8.           }
9.     }
10. }
```

Output

```
1. Name : Ashwani Rajput
2. Name : Soono Jaiswal
3. Name : Rishi Kumar
4. Name : Rahul Mehta
5. Name : Hemant Mishra
```

Presentation Layer Design Pattern

1. Intercepting Filter Pattern
2. Front Controller Pattern
3. View Helper Pattern
4. Composite View Pattern

Intercepting Filter Pattern
An Intercepting Filter Pattern says that "if you want to intercept and manipulate a request and response before and after the request is processed".

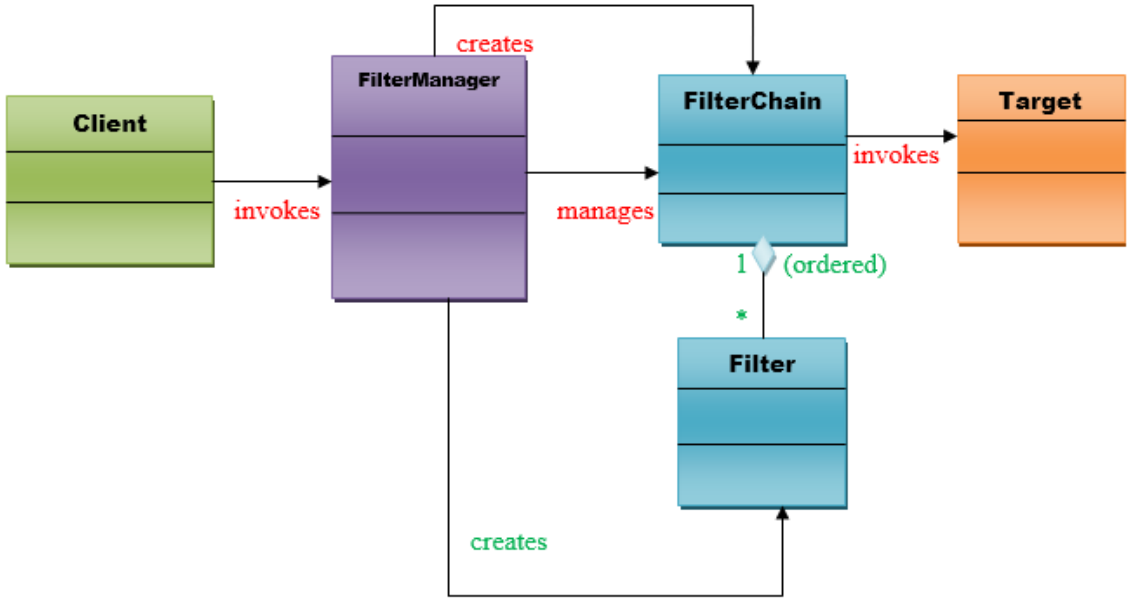
Usage:

- When you want centralization, common processing across requests, such as logging information about each request, compressing an outgoing response or checking the data encoding scheme of each request.
- When you want pre and post processing the components which are loosely coupled with core request-handling services to facilitate which are not suitable for addition and removal.

Benefits:

- It provides central control with loosely coupled handlers.
- It improves reusability.

UML for Intercepting Filter Pattern:



Implementation of Intercepting Filter Pattern:

Step 1
Create a *Login.html* web page.

```
1.  <!DOCTYPE html>
2.  <html>
3.  <head>
4.  <meta charset="US-ASCII">
5.  <title>Login Page</title>
6.  </head>
7.  <body>
8.
9.  <form action="LoginServlet" method="post">
10.
11. Username: <input type="text" name="username">
12. <br><br>
13. Password: <input type="password" name="password">
14. <br><br>
15. <input type="submit" value="Login">
16.
17. </form>
18. </body>
```

Step 2
Create a *LoginServlet* class.

```
1.  package sessions;
2.
3.  import java.io.IOException;
4.  import java.io.PrintWriter;
5.
6.  import javax.servlet.RequestDispatcher;
```

```
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.Cookie;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
**
 * Servlet implementation class LoginServlet
 */
@WebServlet("/LoginServlet")
public class LoginServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    private final String user = "admin";
    private final String password = "admin@1234";

public LoginServlet() {
        super();
    }
protected void doPost(HttpServletRequest request, HttpServletResponse response) throws Servl
etException, IOException {

        // get request parameters for username and passwd
        String username = request.getParameter("username");
        String passwd = request.getParameter("password");

        if(user.equals(username) && password.equals(passwd)){
            HttpSession session = request.getSession();
            session.setAttribute("user", "ashwani");

            //setting session to expire in 1 hour
            session.setMaxInactiveInterval(60*60);

            Cookie userName = new Cookie("user", user);
            userName.setMaxAge(60*60);
            response.addCookie(userName);
            response.sendRedirect("LoginSuccess.jsp");
        }else{
            RequestDispatcher rd = getServletContext().getRequestDispatcher("/Login.html");
            PrintWriter out= response.getWriter();
            out.println("<font color=red>Either user name or password is wrong.</font>");
            rd.include(request, response);
        }
    }
}//End of the LoginServlet class.
```

Step 3

Create a *LoginSuccess.jsp* page.

```
<%@ page language="java" contentType="text/html; charset=US-ASCII"
    pageEncoding="US-ASCII"%>
<!DOCTYPE html PUBLIC "-
//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
```

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=US-ASCII">
<title>Login Success Page</title>
</head>
<body>
<%
//allow access only if session exists
String user = (String) session.getAttribute("user");
String userName = null;
String sessionID = null;
Cookie[] cookies = request.getCookies();
if(cookies !=null){
for(Cookie cookie : cookies){
    if(cookie.getName().equals("user")) userName = cookie.getValue();
    if(cookie.getName().equals("JSESSIONID")) sessionID = cookie.getValue();
}
}
%>
<h3>Hi <%=userName %>, Login successful.Your Session ID=<%=sessionID %></h3>
<br>
User=<%=user %>
<br>
<a href="CheckoutPage.jsp">Checkout Page</a><br>
<form action="LogoutServlet" method="post">
<input type="submit" value="Logout" >
</form>
</body>
</html>
```

Step 4

Create an *AdminPage.jsp* page.

```
<%@ page language="java" contentType="text/html; charset=US-ASCII"
    pageEncoding="US-ASCII"%>
<!DOCTYPE html PUBLIC "-
//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=US-ASCII">
<title>Login Success Page</title>
</head>
<body>
<%
String userName = null;
String sessionID = null;
Cookie[] cookies = request.getCookies();
if(cookies !=null){
for(Cookie cookie : cookies){
    if(cookie.getName().equals("user")) userName = cookie.getValue();
}
}
%>
```

```
<h3>Hi <%=userName %>, These services are only for you to take action.</h3>
<br>
<form action="LogoutServlet" method="post">
<input type="submit" value="Logout" >
</form>
</body>
</html>
```

Step 5

Create an *LogoutServlet* class.

```
package sessions;

import java.io.IOException;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.Cookie;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

/**
 * Servlet implementation class LogoutServlet
 */
@WebServlet("/LogoutServlet")
public class LogoutServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;

    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws Se
rvletException, IOException {
        response.setContentType("text/html");
        Cookie[] cookies = request.getCookies();
        if(cookies != null){
        for(Cookie cookie : cookies){
            if(cookie.getName().equals("JSESSIONID")){
                System.out.println("JSESSIONID="+cookie.getValue());
                break;
            }
        }
        }
        //invalidate the session if exists
        HttpSession session = request.getSession(false);
        System.out.println("User="+session.getAttribute("user"));
        if(session != null){
            session.invalidate();
        }
        response.sendRedirect("Login.html");
    }

}//End of the LogoutServlet class
```

Step 6

Create an *AuthenticationFilter* class.

```
1.  package filters;
2.
3.  import java.io.IOException;
4.  import javax.servlet.Filter;
5.  import javax.servlet.FilterChain;
6.  import javax.servlet.FilterConfig;
7.  import javax.servlet.ServletContext;
8.  import javax.servlet.ServletException;
9.  import javax.servlet.ServletRequest;
10. import javax.servlet.ServletResponse;
11. import javax.servlet.annotation.WebFilter;
12. import javax.servlet.http.HttpServletRequest;
13. import javax.servlet.http.HttpServletResponse;
14. import javax.servlet.http.HttpSession;
15.
16. /**
17.  * Servlet Filter implementation class AuthenticationFilter
18.  */
19. @WebFilter("/AuthenticationFilter")
20. public class AuthenticationFilter implements Filter {
21.
22.     private ServletContext context;
23.
24.     public void init(FilterConfig fConfig) throws ServletException {
25.         this.context = fConfig.getServletContext();
26.         this.context.log("AuthenticationFilter initialized");
27.     }
28.
29.     public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) thr
    ows IOException, ServletException {
30.
31.         HttpServletRequest req = (HttpServletRequest) request;
32.         HttpServletResponse res = (HttpServletResponse) response;
33.
34.         String uri = req.getRequestURI();
35.         this.context.log("Requested Resource::"+uri);
36.
37.         HttpSession session = req.getSession(false);
38.
39.         if(session == null && !(uri.endsWith("html") || uri.endsWith("LoginServlet"))){
40.             this.context.log("Unauthorized access request");
41.             res.sendRedirect("Login.html");
42.         }else{
43.             // pass the request along the filter chain
44.             chain.doFilter(request, response);
45.         }
46.     }
47.     public void destroy() {
48.         //close any resources here
49.     }
```

```
50. }//End of the AuthenticationFilter class
```

Step 7

Create an *RequestLoggingFilter* class.

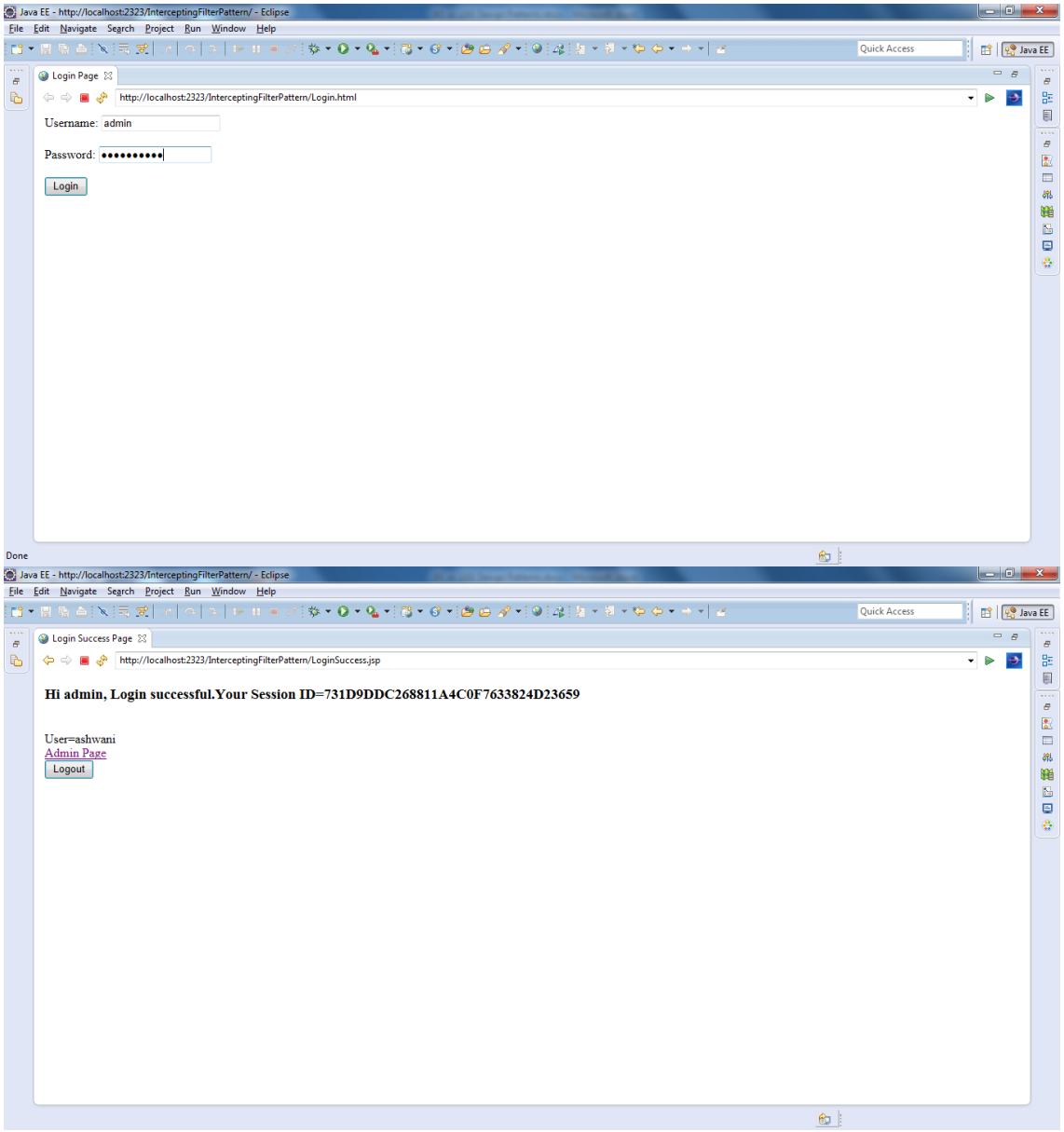
```
1.  package filters;
2.  import java.io.IOException;
3.  import java.util.Enumeration;
4.  import javax.servlet.Filter;
5.  import javax.servlet.FilterChain;
6.  import javax.servlet.FilterConfig;
7.  import javax.servlet.ServletContext;
8.  import javax.servlet.ServletException;
9.  import javax.servlet.ServletRequest;
10. import javax.servlet.ServletResponse;
11. import javax.servlet.annotation.WebFilter;
12. import javax.servlet.http.Cookie;
13. import javax.servlet.http.HttpServletRequest;
14. @WebFilter("/RequestLoggingFilter")
15. public class RequestLoggingFilter implements Filter {
16. private ServletContext context;
17.  public void init(FilterConfig fConfig) throws ServletException {
18.        this.context = fConfig.getServletContext();
19.        this.context.log("RequestLoggingFilter initialized");
20.    }
21. public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws
     IOException, ServletException {
22.        HttpServletRequest req = (HttpServletRequest) request;
23.        Enumeration<String> params = req.getParameterNames();
24. while(params.hasMoreElements()){
25.            String name = params.nextElement();
26.            String value = request.getParameter(name);
27.            this.context.log(req.getRemoteAddr() + "::Request Params::{"+name+"="+value+"}")
    ;
28.        }
29.
30.        Cookie[] cookies = req.getCookies();
31.        if(cookies != null){
32.            for(Cookie cookie : cookies){
33.                this.context.log(req.getRemoteAddr() + "::Cookie::{"+cookie.getName()+","+cookie
    .getValue()+"}");
34.            }
35.        }
36.        // pass the request along the filter chain
37.        chain.doFilter(request, response);
38.    }
39.
40.    public void destroy() {
41.        //we can close resources here
42.    }
43. }// End of the RequestLoggingFilter class
```

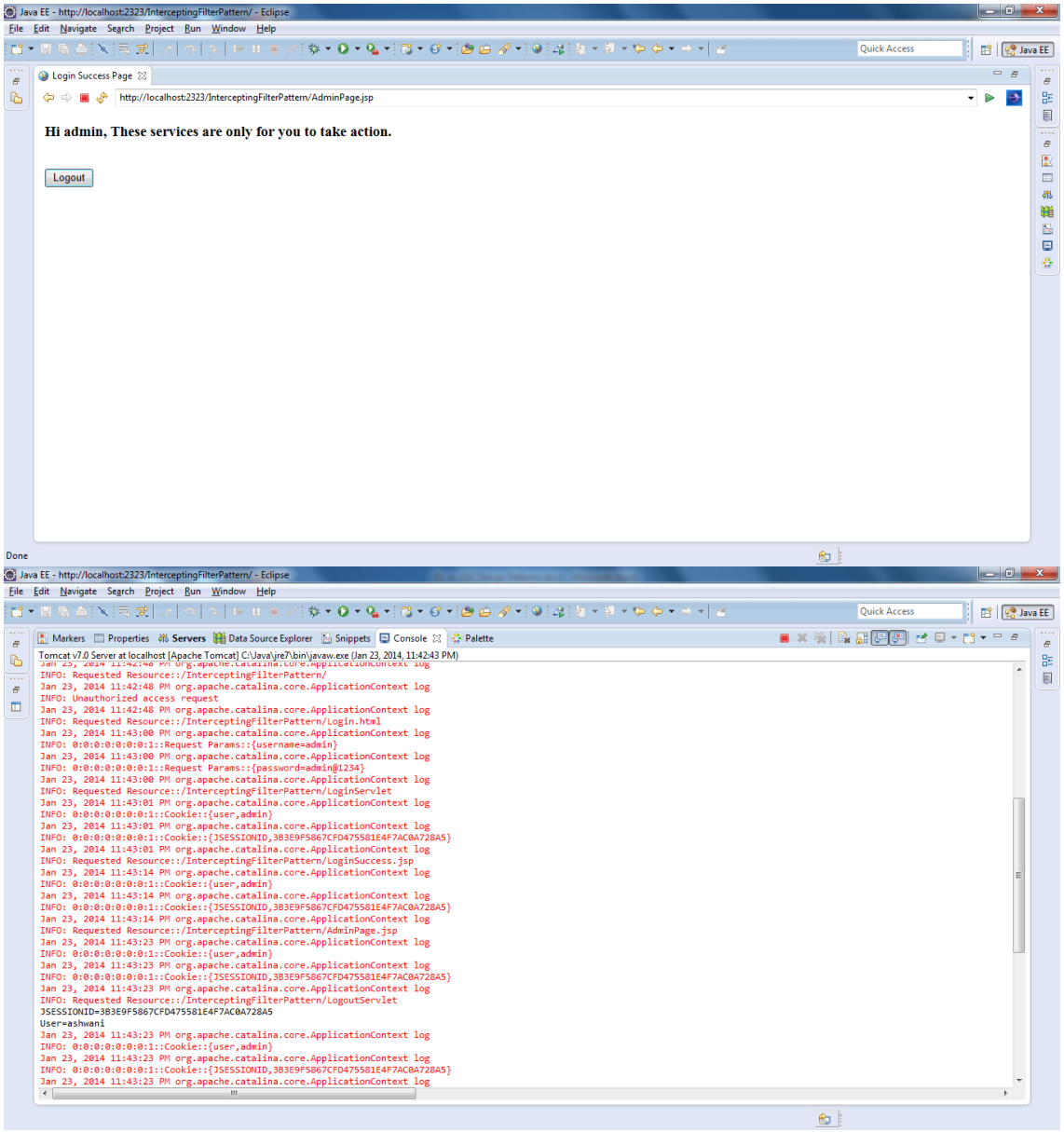
Step 8

Create a *web.xml* file.

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-
    instance" xmlns="http://java.sun.com/xml/ns/javaee" xsi:schemaLocation="http://java.sun.co
    m/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd" version="3.0">
3.    <display-name>ServletFilterExample</display-name>
4.    <welcome-file-list>
5.      <welcome-file>Login.html</welcome-file>
6.    </welcome-file-list>
7.    <filter>
8.      <filter-name>RequestLoggingFilter</filter-name>
9.      <filter-class>filters.RequestLoggingFilter</filter-class>
10.   </filter>
11.   <filter>
12.     <filter-name>AuthenticationFilter</filter-name>
13.     <filter-class>filters.AuthenticationFilter</filter-class>
14.   </filter>
15.   <filter-mapping>
16.     <filter-name>RequestLoggingFilter</filter-name>
17.     <url-pattern>/*</url-pattern>
18.     <dispatcher>REQUEST</dispatcher>
19.   </filter-mapping>
20.   <filter-mapping>
21.     <filter-name>AuthenticationFilter</filter-name>
22.     <url-pattern>/*</url-pattern>
23.   </filter-mapping>
24. </web-app>
```

Output:


Java EE - http://localhost:2323/InterceptingFilterPattern/ - Eclipse
File Edit Navigate Search Project Run Window Help
Quick Access
Java EE
Login Page
http://localhost:2323/InterceptingFilterPattern/Login.html
Username: admin
Password:
Login
Done
Java EE - http://localhost:2323/InterceptingFilterPattern/ - Eclipse
File Edit Navigate Search Project Run Window Help
Quick Access
Java EE
Login Success Page
http://localhost:2323/InterceptingFilterPattern/LoginSuccess.jsp
Hi admin, Login successful.Your Session ID=731D9DDC268811A4C0F7633824D23659
User=ashwani
Admin Page
Logout



Next →← Prev

Front Controller Pattern
A Front Controller Pattern says that if you want to provide the centralized request handling mechanism so that all the requests will be handled by a single handler". This handler can do the authentication or authorization or logging or tracking of request and then pass the requests to corresponding handlers.

Usage:

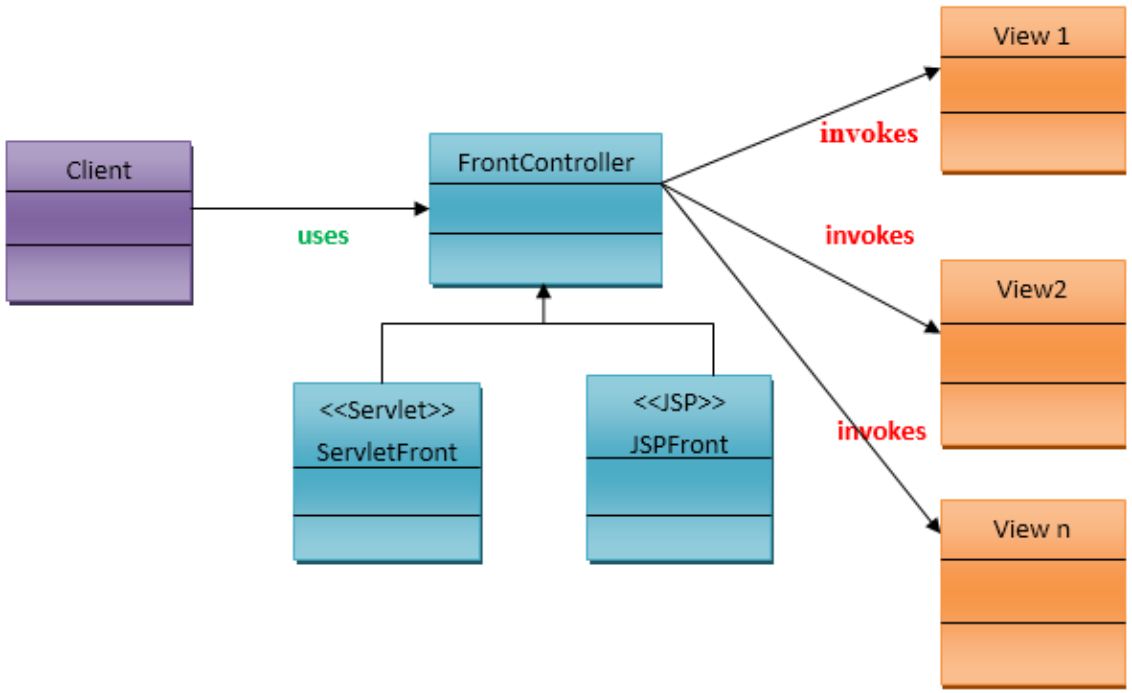
- When you want to control the page flow and navigation.
- When you want to access and manage the data model.

- When you want to handle the business processing.

Benefits:

- It reduces the duplication of code in JSP pages, especially in those cases where several resources require the same processing.
- It maintains and controls a web application more effectively.
- A web application of two-tier architecture, the recommended approach is front controller to deal with user requests.

UML for Front Controller Pattern:



Implementation of Front Controller Pattern:

Step 1
Create a *Login.html* web page.

```
1. <html>
2. <head>
3. <title>
4. Login page
5. </title>
6. </head>
7. <body style="color:green;">
8. <h1 style="font-family:Comic Sans Ms;text-align="center";font-size:20pt;
9. color:#00FF00;>
```

```
Login Page
</h1>
<form method="POST" action="FrontControllerServlet" onsubmit="return checkForm(this);"
>
<p>Username: <input type="text" name="username"></p>
<p>Password: <input type="password" name="pwd1"></p>
<p>Confirm Password: <input type="password" name="pwd2"></p>
<p><input type="submit" value="Login"></p>
</form>
<script type="text/javascript">

 function checkForm(form)
 {
  if(form.username.value == "") {
    alert("Error: Username cannot be blank!");
    form.username.focus();
    return false;
  }
  re = /^\w+$/;
  if(!re.test(form.username.value)) {
    alert("Error: Username must contain only letters, numbers and underscores!");
    form.username.focus();
    return false;
  }

  if(form.pwd1.value != "" && form.pwd1.value == form.pwd2.value) {
    if(form.pwd1.value.length < 6) {
      alert("Error: Password must contain at least six characters!");
      form.pwd1.focus();
      return false;
    }
    if(form.pwd1.value == form.username.value) {
      alert("Error: Password must be different from Username!");
      form.pwd1.focus();
      return false;
    }
  re = /[0-9]/;
    if(!re.test(form.pwd1.value)) {
      alert("Error: password must contain at least one number (0-9)!");
      form.pwd1.focus();
      return false;
    }
    re = /[a-z]/;
    if(!re.test(form.pwd1.value)) {
      alert("Error: password must contain at least one lowercase letter (a-z)!");
      form.pwd1.focus();
      return false;
    }
    re = /[A-Z]/;
    if(!re.test(form.pwd1.value)) {
      alert("Error: password must contain at least one uppercase letter (A-Z)!");
```

```
60.         form.pwd1.focus();
61.         return false;
62.       }
63.     } else {
64.       alert("Error: Please check that you've entered and confirmed your password!");
65.       form.pwd1.focus();
66.       return false;
67.     }
68.     return true;
69.   }
70.
71. </script>
72. </body>
73. </html>
```

Step 2

Create a *FrontControllerServlet.java* class which is a servlet and it may be a JSP page also.

```
1.  package controller;
2.
3.  import java.io.IOException;
4.  import java.io.PrintWriter;
5.
6.  import javax.servlet.RequestDispatcher;
7.  import javax.servlet.ServletException;
8.  import javax.servlet.annotation.WebServlet;
9.  import javax.servlet.http.HttpServlet;
10. import javax.servlet.http.HttpServletRequest;
11. import javax.servlet.http.HttpServletResponse;
12.
13. /**
14.  * Servlet implementation class FrontControllerServlet
15.  */
16. @WebServlet("/FrontControllerServlet")
17. public class FrontControllerServlet extends HttpServlet {
18.
19.     protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
    ServletException, IOException {
20.
21.         response.setContentType("text/html");
22.         PrintWriter out = response.getWriter();
23.
24.         String username=request.getParameter("username");
25.         String password=request.getParameter("pwd2");
26.
27.         if (password.equals("Ashwani1987")) {
28.
29.             RequestDispatcher rd=request.getRequestDispatcher("/Success.jsp");
30.             rd.forward(request, response);
31.         } else {
32.
33.             RequestDispatcher rd=request.getRequestDispatcher("/Error.jsp");
34.             rd.forward(request, response);
```

```
35.       }
36.
37.    }
38.
39. }
```

Step 3

Create a *Success.jsp* page.

```
1.  <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2.     pageEncoding="ISO-8859-1"%>
3.  <!DOCTYPE html PUBLIC "-
    //W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
4.  <html>
5.  <head>
6.  <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
7.  <title>Welcome Page</title>
8.  </head>
9.  <body style="background-color: gray;">
10.
11.   <%  String name=request.getParameter("username"); %>
12.
13.    <b>Welcome,</b> <% out.print(name); %>
14.
15. </body>
16. </html>
```

Step 4

Create a *Error.jsp* page.

```
1.  <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2.     pageEncoding="ISO-8859-1"%>
3.  <!DOCTYPE html PUBLIC "-
    //W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
4.  <html>
5.  <head>
6.  <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
7.  <title>Insert title here</title>
8.  </head>
9.  <body style="background-color: olive;">
10.
11.    <b>You are have entered wrong username or password!!</b><br>
12.
13.    <a href="Login.html">Back To Home Page</a>
14. </body>
15. </html>
```

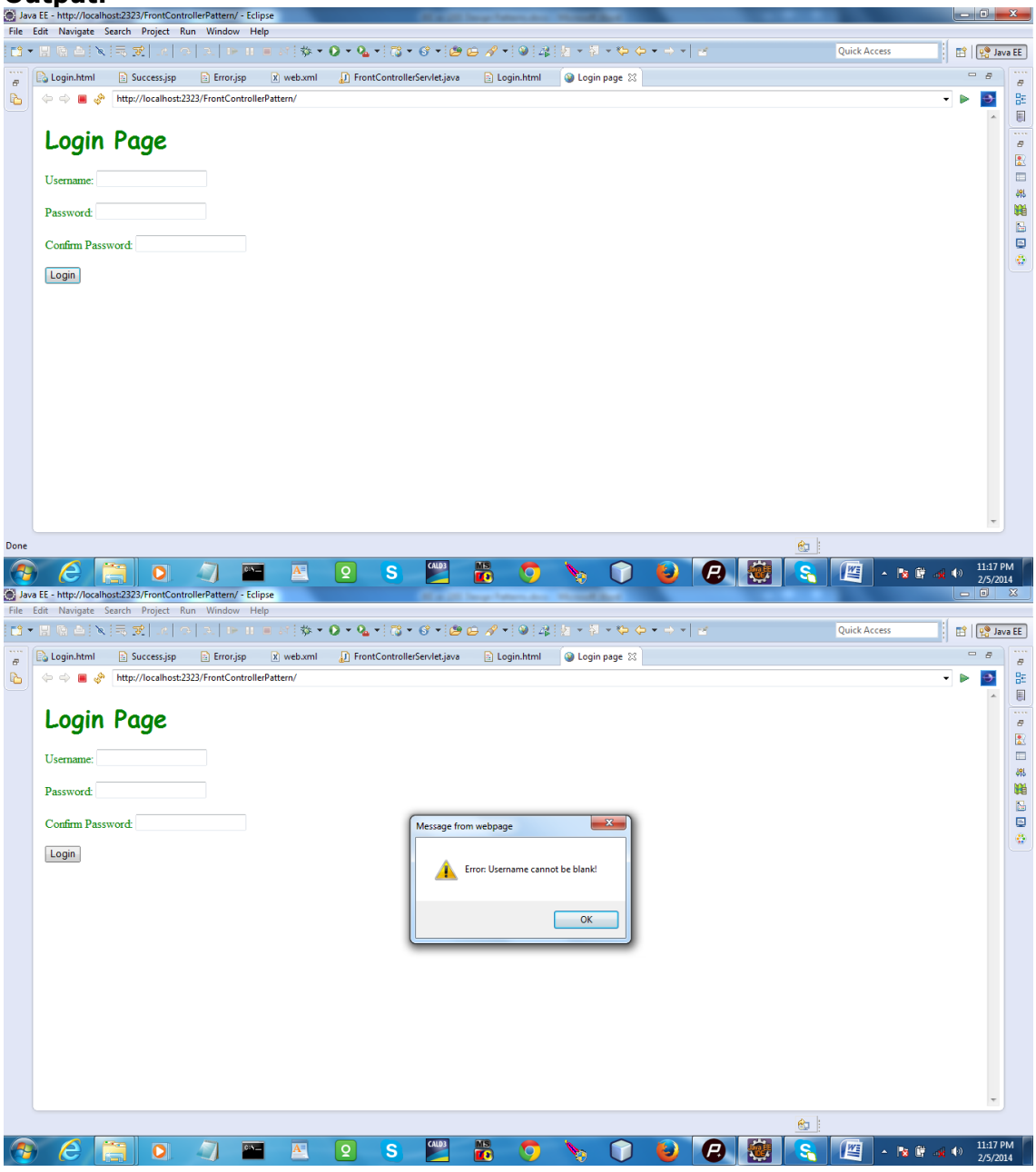
Step 5

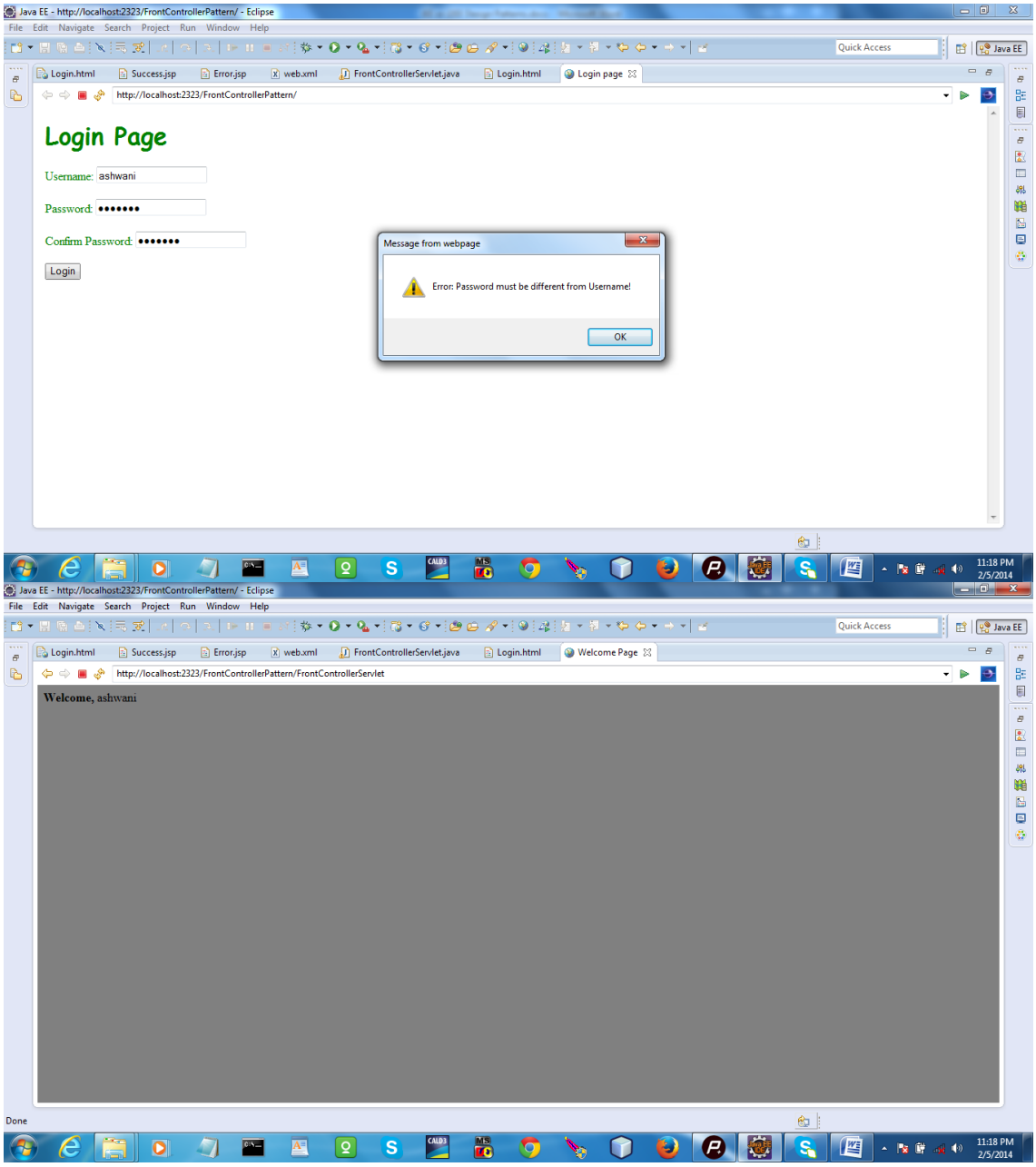
Create a *web.xml* file.

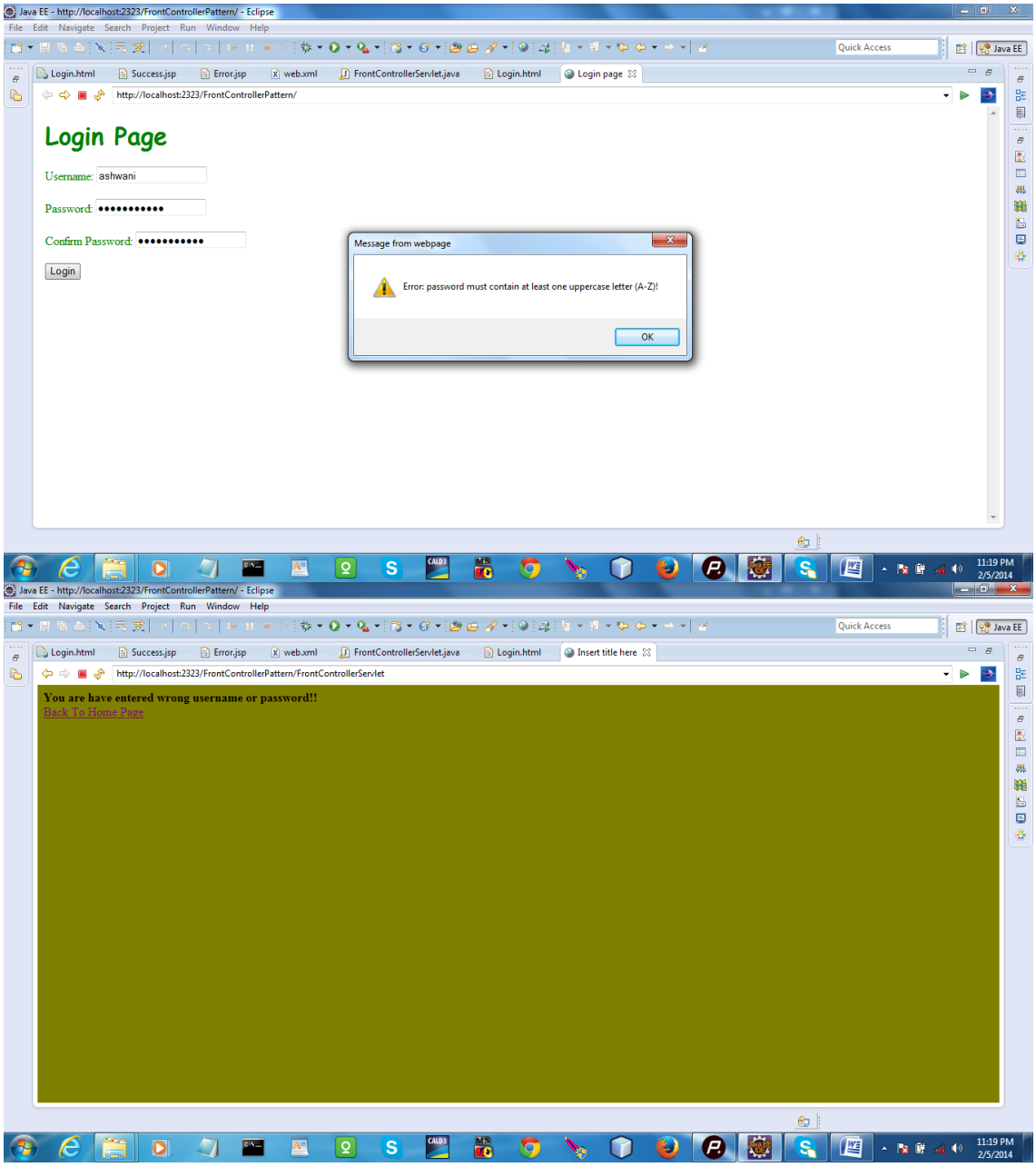
```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-
    instance" xmlns="http://java.sun.com/xml/ns/javaee" xsi:schemaLocation="http://java.sun.
    com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd" version="3.0">
3.    <display-name>Front Controller Example</display-name>
4.    <welcome-file-list>
5.      <welcome-file>Login.html</welcome-file>
```

```
6.    </welcome-file-list>
7.  </web-app>
```

Output:




Java EE - http://localhost:2323/FrontControllerPattern/ - Eclipse
File Edit Navigate Search Project Run Window Help
Quick Access
Java EE
Login.html
Success.jsp
Error.jsp
web.xml
FrontControllerServlet.java
Login.html
Login page
http://localhost:2323/FrontControllerPattern/
Login Page
Username: ashwani
Password: •••••••
Confirm Password: •••••••
Login
Message from webpage
Error: Password must be different from Username!
OK
11:18 PM
2/5/2014
Java EE - http://localhost:2323/FrontControllerPattern/ - Eclipse
File Edit Navigate Search Project Run Window Help
Quick Access
Java EE
Login.html
Success.jsp
Error.jsp
web.xml
FrontControllerServlet.java
Login.html
Welcome Page
http://localhost:2323/FrontControllerPattern/FrontControllerServlet
Welcome, ashwani
Done
11:18 PM
2/5/2014



The Business Logic Layer

The *business logic layer* typically contains deployed EJB components that encapsulate business rules and other business functions in:

- Session Beans
- Entity Beans
- Message-Driven Beans

For more information about components in the business logic layer, see the *Sun Java System Application Server Developer's Guide to Enterprise JavaBeans Technology*.

Session Beans

Session beans encapsulate the business processes and rules logic. For example, a session bean could calculate taxes for a billing invoice. When there are complex business rules that change frequently (for example, due to new business practices or new government regulations), an application typically uses more session beans than entity beans, and session beans may need continual revision.
Session beans are likely to call a full range of JDBC interfaces, as well as other EJB components.
Applications perform better when session beans are stateless, although session beans can be stateful. A stateful session bean is needed when a user-specific state, such as a shopping cart, must be maintained on the server.

Entity Beans

Entity beans represent persistent objects, such as a database row. Entity beans are likely to call a full range of JDBC interfaces. However, entity beans typically do not call other EJB components. The entity bean developer's role is to design an object-oriented view of an organization's business data. Creating this object-oriented view often means mapping database tables into entity beans. For example, the developer might translate a customer table, invoice table, and order table into corresponding customer, invoice, and order objects.
An entity bean developer works with session bean and servlet developers to ensure that the application provides fast, scalable access to persistent business data.
There are two types of entity bean persistence:

- Container managed persistence (CMP) - The EJB container is responsible for maintaining the interactions between the business logic and the database.
- Bean managed persistence (BMP) - The developer is responsible for writing the code that controls interaction with the database.

Message-Driven Beans

Message-driven beans are persistent objects that are likely to call a full range of JDBC interfaces, much like entity beans. However, message-driven beans have no local or remote interfaces as do other EJB components, and they differ from entity beans in how they are accessed.
A message-driven bean is a message listener that can reliably consume messages from a queue or a durable subscription. The messages may be sent by any J2EE component—from an application client, another EJB component, or a Web component—or from an application or a system that does not use J2EE technology.
For example, an inventory entity bean may send a message to a stock ordering message-driven bean when the amount of an item is below a set lower limit.
The Data Access Layer
In the *data access layer*, JDBC (Java database connectivity) is used to connect to databases, make queries, and return query results, and custom connectors work with the Sun Java System Application Server to enable communication with legacy EIS systems, such as IBM's CICS.
Developers are likely to integrate access to the following systems using J2EE CA (connector architecture):

- Enterprise resource management systems
- Mainframe systems
- Third-party security systems

For more information about JDBC, see the *Sun Java System Application Server Developer's Guide to J2EE Services and APIs*.
For more information about connectors, see the *Sun Java System J2EE CA Service Provider Implementation Administrator's Guide* and the corresponding release notes.

Best Practices for Designing J2EE Applications

This section lists guidelines to consider when designing and developing an Sun Java System Application Server application, and is merely a summary. For more details, you may want to consult *Core J2EE Patterns: Best Practices and Design Strategies* by Deepak Alur, John Crupi, and Dan Malks.
The guidelines are grouped into the following goals:

- Presenting Data with Servlets and JSPs
- Creating Reusable Application Code
- Modularizing Applications

Presenting Data with Servlets and JSPs
Servlets are often used for presentation logic and serve as central dispatchers of user input and data presentation. JSPs are used to dynamically generate the presentation layout. Both servlets and JSPs can be used to conditionally generate different pages.
If the page layout is its main feature and there is minimal processing involved to generate the page, it may be easier to use a JSP for the interaction.
For example, after an online bookstore application authenticates a user, it provides a boilerplate portal front page for the user to choose one of several tasks, including a book search, purchase selected items, and so on. Since this portal conducts very little processing, it can be implemented as a JSP.
Think of JSPs and servlets as opposite sides of the same coin. Each can perform all the tasks of the other, but each is designed to excel at one task at the expense of the other. The strength of servlets is in processing and adaptability. However, performing HTML output from them involves many cumbersome println statements. Conversely, JSPs excel at layout tasks because they are simply HTML files and can be edited with HTML editors, though performing complex computational or processing tasks with them can be awkward. You can use JSPs and servlets together to get the benefits of both.
For more information on servlets and JSPs, see the *Sun Java System Application Server Developer's Guide to Web Applications*.
Creating Reusable Application Code
Aside from using good object-oriented design principles, there are several things to consider when developing an application to maximize reusability, including the following tips:

- Use relative paths and URLs so links remain valid if the code tree moves.
- Minimize Java in JSPs; instead, put Java in servlets and helper classes. JSP designers can revise JSPs without being Java experts.
- Use property files or global classes to store hard-coded strings such as the data source names, tables, columns, JNDI objects, or other application properties.
- Use session beans, rather than servlets and JSPs, to store business rules that are domain-specific or likely to change often, such as input validation.
- Use entity beans for persistent objects; using entity beans allows management of multiple beans per user.
- For maximum flexibility, use Java interfaces rather than Java classes.
- Use J2EE CA to access legacy data.

Modularizing Applications
The major factors to keep in mind when designing your J2EE Applications are:

- Functional Isolation
- Reusable Code
- Prepackaged Components
- Shared Framework Classes
- Security Issues

For more information about assembling modules and applications, see Chapter 4, "Assembling and Deploying J2EE Applications."

Functional Isolation

Each component should do one thing and one thing only. For example, in a payroll system, one EJB component should access the 401k accounts while a separate bean accesses the salary database. This functional isolation of tasks leads to the physical isolation of business logic into two separate beans. If separate development teams create these beans, each team should develop its own EJB JAR package.
Scenario 1
Assume that the user interface development team works with both of the bean development teams. In this case, the UI development team should assemble its servlets, JSPs, and static files into one WAR file. For example:
payroll system EAR file = payroll EJB jar
+ 401k EJB JAR
+ 1 common war from the UI team
This isolation of functionality within an EAR file does not mean that components cannot interact with each other. The beans (in separate EJB JAR files) can call business methods from each other.
Scenario 2
Assume that each bean development team has its own UI development team. If this is the case, then each web development team should assemble its servlets, JSPs, and static files into separate WAR files. For example:
payroll system EAR file = payroll EJB jar
+ 401k EJB JAR
+ 1 payroll UI team's war + 1 401k UI team's war
With this setup, the components in each WAR file can access components from the other WAR file.
Assembly Formulas
Some general formulas should be followed when assembling modules and applications. The following table outlines assembly formulas.

Table 1-1 Assembly Formulas

Type of Development Group	Teams in Group	Modularizing Scheme
Small workgroup	1 web team + 1 EJB team	1 EAR = 1 EJB + 1 WAR
Enterprise workgroup	2 EJB teams + 1 web team + 1 component	1 EAR = 2 EJB + 1 WAR + 1 individual component

Reusable Code

Reusable components are the primary reason for assembling and deploying individual modules rather than applications. If the code developed by one team of developers is a reusable component that may be accessed by several applications (different EAR files), then that code should be deployed as an individual module. For more information, see Chapter 4, "Assembling and Deploying J2EE Applications."

Prepackaged Components

If you do not want to create your application from scratch, you can use prepackaged components. Today's leading J2EE component vendors offer many prepackaged components that provide a whole host of services. Their goal is to provide up to 60% of the standard components needed for an application. With Sun Java System Application Server, you can easily assemble applications that make use of these readily available components.

Shared Framework Classes

Sometimes several applications need to access a single modular library. In such cases, including the library in each J2EE application is not a good idea for these reasons:

- **Library size:** Most framework libraries are large, so including them in an application increases the size of the assembled application.
- **Different versions:** Because a separate classloader loads each application, several copies of the framework classes exist during runtime.

For tips on how to set up a library so multiple applications can share it.

Security Issues

You should not allow unauthorized runtime access to classes, EJB components, and other resources. A component should only contain classes that are permitted to access other resources included in the component.

Business Delegate Design Pattern in Java

The ***Business Delegate pattern*** adds an abstraction layer between presentation and business tiers. By using the pattern we gain loose coupling between the tiers and encapsulate knowledge about how to locate, connect to, and interact with the business objects that make up the application.

Business Delegate Pattern is used to decouple presentation tier and business tier.

Let's discuss how ***Business Delegate Pattern*** decouples presentation tier and business tier with a class diagram and source code.

This pattern is divided into a number of sections for simplicity like problem, forces, solution etc.

Table of contents
Problem
Forces
Solution
Structure - Class Diagram, Sequence Diagram
Participants and Responsibilities
Implementation
Consequences
Applicability
References

Problem

(Problem section describes the design issues faced by the developer)

You want to hide clients from the complexity of remote communication with business service components.

Forces

(This section describes Lists the reasons and motivations that affect the problem and the solution. The list of forces highlights the reasons why one might choose to use the pattern and provides a justification for using the pattern)

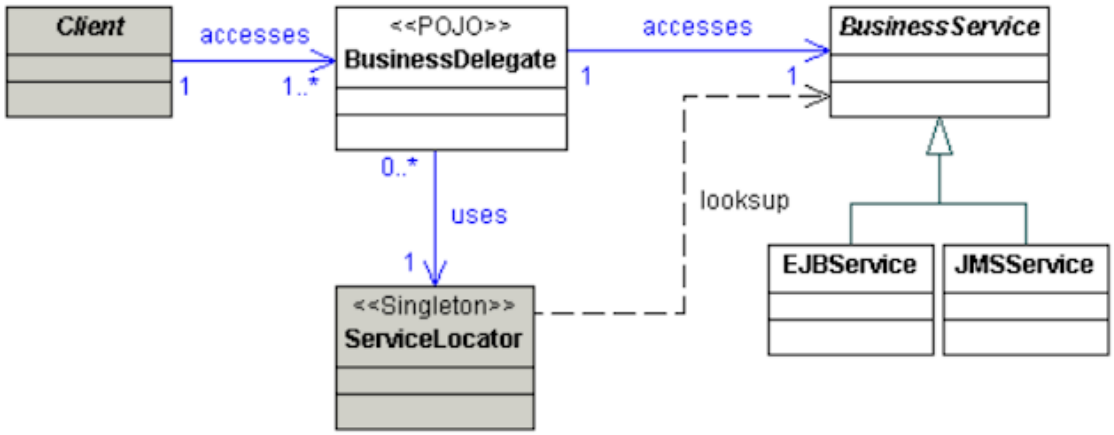
- You want to access the business-tier components from your presentation-tier components and clients, such as devices, web services, and rich clients.
- You want to minimize coupling between clients and the business services, thus hiding the underlying implementation details of the service, such as lookup and access.
- You want to avoid unnecessary invocation of remote services.
- You want to translate network exceptions into application or user exceptions.
- You want to hide the details of service creation, reconfiguration, and invocation retries from the clients.

Solution

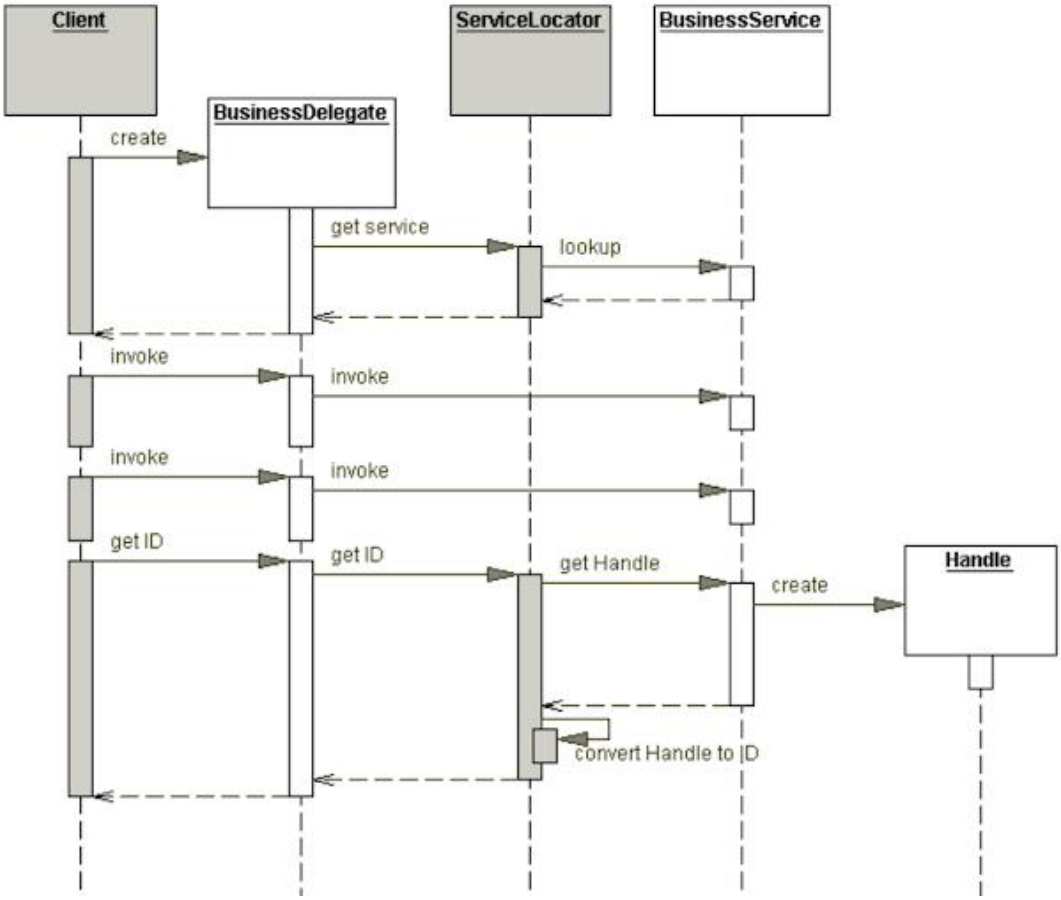
(Here solution section describes the solution approach briefly and the solution elements in detail)
Use a *Business Delegate* to encapsulate access to a business service. The Business Delegate hides the implementation details of the business service, such as lookup and access mechanisms.

Structure

Let's use UML class diagram to show the basic structure of the solution and the UML Sequence diagram in this section present the dynamic mechanisms of the solution.
Class Diagram



Sequence Diagram



Participants and Responsibilities

BusinessDelegate - The BusinessDelegate's role is to provide control and protection for the business service.
ServiceLocator - The ServiceLocator encapsulates the implementation details of locating a BusinessService component.
BusinessService - The BusinessService is a business-tier component, such as an enterprise bean, that is being accessed by the client.
Client - Client can be JSP or Servlet or UI code

Implementation

(This section includes source code implementations and code listings for the patterns)
Let's refer the above class diagram and write source code to define Business Delegate Pattern.
Let's create source code step by step with reference to the class diagram.
Step 1: Create *BusinessService* Interface.
Interface for Business service implementations like *JMSService* and *EJBService*.

```
public interface BusinessService {
  void doProcessing();
```

```
}
```

Step 2 : Create concrete Service classes.
This is Service EJB implementation.

```
public class EJBService implements BusinessService {
  @Override
  public void doProcessing() {
    System.out.println("EJBService is now processing");
  }
}
//This is Service JMS implementation.

public class JMSService implements BusinessService {
  @Override
  public void doProcessing() {
    System.out.println("JMSService is now processing");
  }
}
//Service Email implementation.

public class EmailService implements BusinessService {
  @Override
  public void doProcessing() {
    System.out.println("EmailService is now processing");
  }
}
```

Step 3 : Create Business Lookup Service.
Class for performing service lookups.
This class acts as *ServiceLocator* encapsulates the implementation details of locating *BusinessService* components.

```
public class BusinessLookup {

 private EjbService ejbService;

 private JmsService jmsService;

 private EmailService emailService;

 /**
  * @param serviceType
  *            Type of service instance to be returned.
  * @return Service instance.
  */
 public BusinessService getBusinessService(ServiceType serviceType) {
  if (serviceType.equals(ServiceType.EJB)) {
   return ejbService;
  } else if (serviceType.equals(ServiceType.JMS)) {
```

```
  return jmsService;
 } else {
  return emailService;
 }
}

public void setJmsService(JmsService jmsService) {
 this.jmsService = jmsService;
}

public void setEjbService(EjbService ejbService) {
 this.ejbService = ejbService;
}

public void setEmailService(EmailService emailService) {
 this.emailService = emailService;
}
}
```

Step 4: Create Business Delegate.
BusinessDelegate separates the presentation and business tiers.

```
public class BusinessDelegate {

  private BusinessLookup lookupService;
  private BusinessService businessService;
  private ServiceType serviceType;

  public void setLookupService(BusinessLookup businessLookup) {
    this.lookupService = businessLookup;
  }

  public void setServiceType(ServiceType serviceType) {
    this.serviceType = serviceType;
  }

  public void doTask() {
    businessService = lookupService.getBusinessService(serviceType);
    businessService.doProcessing();
  }
}
 Enumeration of service types

public enum ServiceType {
  EJB, JMS,EMAIL;
}
```

Step 5: Create *Client.* The client utilizes *BusinessDelegate* to call the business tier.

```
public class Client {
```

```
  private BusinessDelegate businessDelegate;

  public Client(BusinessDelegate businessDelegate) {
    this.businessDelegate = businessDelegate;
  }

  public void doTask() {
    businessDelegate.doTask();
  }
}
```

Step 6 : Use *BusinessDelegate* and *Client* classes to demonstrate Business Delegate pattern.
In this example, the client utilizes a business delegate to execute a task. The Business Delegate then selects the appropriate service and makes the service call.

```
public class App {

  /**
   * Program entry point.
   *
   * @param args command line args
   */
  public static void main(String[] args) {

    BusinessDelegate businessDelegate = new BusinessDelegate();
    BusinessLookup businessLookup = new BusinessLookup();
    businessLookup.setEjbService(new EjbService());
    businessLookup.setJmsService(new JmsService());
    businessLookup.setEmailService(new EmailService());

    businessDelegate.setLookupService(businessLookup);
    businessDelegate.setServiceType(ServiceType.EJB);

    Client client = new Client(businessDelegate);
    client.doTask();

    businessDelegate.setServiceType(ServiceType.JMS);
    client.doTask();

    businessDelegate.setServiceType(ServiceType.EMAIL);
    client.doTask();
  }
}
```

The source code of this pattern is available on GitHub.

Applicability

Use the Business Delegate pattern when

- you want loose coupling between presentation and business tiers
- you want to orchestrate calls to multiple business services
- you want to encapsulate service lookups and service calls

Transfer Object Assembler Pattern in Java

Use a ***Transfer Object Assembler*** to build an application model as a composite Transfer Object.
The *Transfer Object Assembler* aggregates multiple Transfer Objects from various business components and services and returns it to the client.
This pattern is divided into a number of sections for simplicity like a problem, forces, solution, class diagram, sequence diagram etc.

Table of contents
Problem
Forces
Solution
Structure - Class Diagram, Sequence Diagram
Participants and Responsibilities
Implementation
Consequences
Applicability
References

Problem

(Problem section describes the design issues faced by the developer)
You want to obtain an application model that aggregates transfer objects from several business components.

Forces

(This section describes Lists the reasons and motivations that affect the problem and the solution. The list of forces highlights the reasons why one might choose to use the pattern and provides a justification for using the pattern)

- You want to encapsulate business logic in a centralized manner and prevent implementing it in the client.
- You want to minimize the network calls to remote objects when building a data representation of the business-tier object model.
- You want to create a complex model to hand over to the client for presentation purposes.
- You want the clients to be independent of the complexity of model implementation, and you want to reduce coupling between the client and the business components.

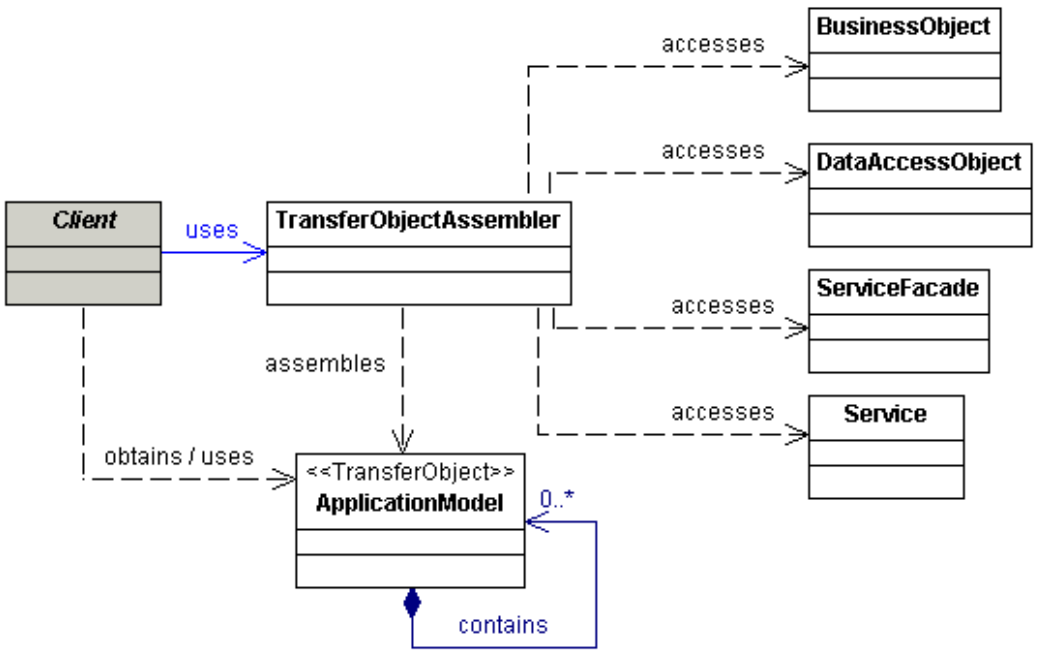
Solution

(Here solution section describes the solution approach briefly and the solution elements in detail)
Use a Transfer Object Assembler to build an application model as a composite Transfer Object. The Transfer Object Assembler aggregates multiple Transfer Objects from various business components and services and returns it to the client.

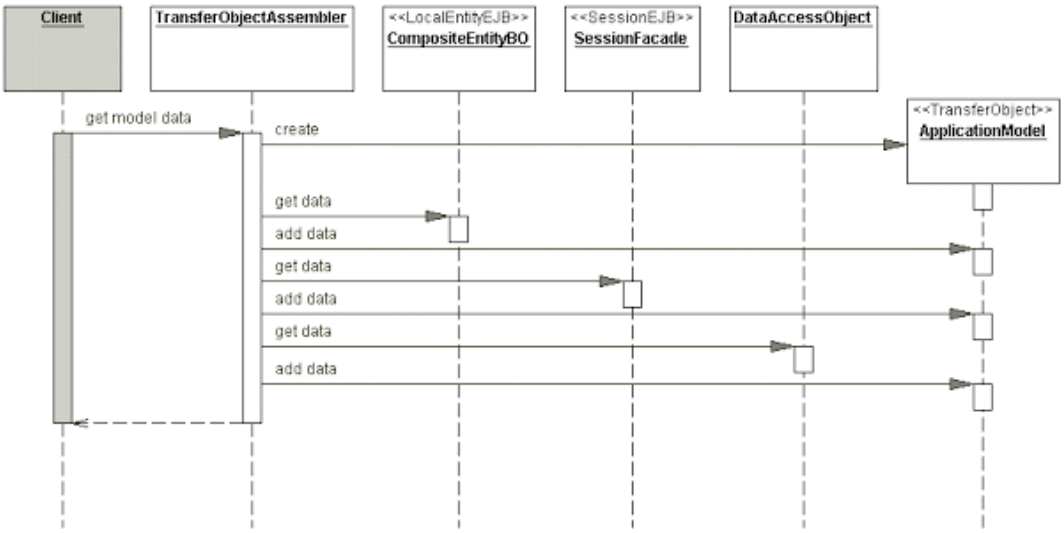
Structure

Let's use UML class diagram to show the basic structure of the solution and the UML Sequence diagram in this section present the dynamic mechanisms of the solution.
Below is the class diagram representing the relationships for the Transfer Object Assembler Pattern.

Class diagram



Sequence Diagram



Participants and Responsibilities

Client - Client invokes the *TransferObjectAssembler* to obtain the application model data. The Client can be a component of the presentation tier or can be a Session Façade that provides the remote access layer for the clients accessing the *TransferObjectAssembler*.

TransferObjectAssembler - The *TransferObjectAssembler* is the main class of this pattern. The *TransferObjectAssembler* constructs a new composite transfer object based on the requirements of the application when the client requests the application model data.
ApplicationModel - The *ApplicationModel* object is a composite Transfer Object that is constructed by the *TransferObjectAssembler* and returned to the Client.
BusinessObject - *BusinessObject* represents a Business Object that provides Transfer Objects to the *TransferObjectAssembler* to assemble the *ApplicationModel*.
SessionFacade - The *SessionFacade* represents a Session Facade that provides part of the data required to construct the *ApplicationModel* transfer object.
DataAccessObject - *DataAccessObject* represents a Data Access Object, used when the *TransferObjectAssembler* needs to obtain data directly from the persistent store.
Service - The Service is any arbitrary service object including an Application Service in the business tier that provides the data required to construct the *ApplicationModel* object.

Implementation

(This section includes source code implementations and code listings for the patterns).
Implementing the Transfer Object Assembler
Consider a ***project management application*** where a number of business-tier components define the complex model. Suppose a client wants to obtain the model data composed of data from various business objects, such as:

- *Project* information from the Project component
- Project manager information from the *ProjectManager* component
- List of project tasks from the *Project* component
- *Resource* information from the *Resource* component.

Step 1: Create Composite Transfer Object Class
A Transfer Object Assembler pattern can be implemented to assemble this composite transfer object.

```
public class ProjectDetailsData {
   private ProjectTO projectData;
   private ProjectManagerTO projectManagerData;
   private Collection < TaskResourceTO > listOfTasks;

   public ProjectDetailsData(ProjectTO projectData, ProjectManagerTO projectManagerData,
      Collection < TaskResourceTO > listOfTasks) {
      super();
      this.projectData = projectData;
      this.projectManagerData = projectManagerData;
      this.listOfTasks = listOfTasks;
   }
}
```

The list of tasks in the *ProjectDetailsData* is a collection of *TaskResourceTO o*bjects.
The *TaskResourceTO* is a combination of *TaskTO* and *ResourceTO*.
Step 2: Create *ResourceTO*, *taskTO* , *ProjectTO* and *ProjectManagerTO* transfer object classes.

```
public class ResourceTO {
   private String resourceId;
   private String resourceName;
   private String resourceEmail;
```

```
    public String getResourceId() {
        return resourceId;
    }
    public void setResourceId(String resourceId) {
        this.resourceId = resourceId;
    }
    public String getResourceName() {
        return resourceName;
    }
    public void setResourceName(String resourceName) {
        this.resourceName = resourceName;
    }
    public String getResourceEmail() {
        return resourceEmail;
    }
    public void setResourceEmail(String resourceEmail) {
        this.resourceEmail = resourceEmail;
    }
}
public class TaskTO {
 private String projectId;
 private String taskId;
 private String name;
 private String description;
 private Date startDate;
 private Date endDate;
 private String assignedResourceId;
 public String getProjectId() {
  return projectId;
 }
 public void setProjectId(String projectId) {
  this.projectId = projectId;
 }
 public String getTaskId() {
  return taskId;
 }
 public void setTaskId(String taskId) {
  this.taskId = taskId;
 }
 public String getName() {
  return name;
 }
 public void setName(String name) {
  this.name = name;
 }
 public String getDescription() {
  return description;
 }
 public void setDescription(String description) {
  this.description = description;
```

```
 }
 public Date getStartDate() {
  return startDate;
 }
 public void setStartDate(Date startDate) {
  this.startDate = startDate;
 }
 public Date getEndDate() {
  return endDate;
 }
 public void setEndDate(Date endDate) {
  this.endDate = endDate;
 }
 public String getAssignedResourceId() {
  return assignedResourceId;
 }
 public void setAssignedResourceId(String assignedResourceId) {
  this.assignedResourceId = assignedResourceId;
 }
}
public class TaskResourceTO {
    private String projectId;
    private String taskId;
    private String name;
    private String description;
    private Date startDate;
    private Date endDate;
    private TaskResourceTO assignedResource;
    public String getProjectId() {
        return projectId;
    }
    public void setProjectId(String projectId) {
        this.projectId = projectId;
    }
    public String getTaskId() {
        return taskId;
    }
    public void setTaskId(String taskId) {
        this.taskId = taskId;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getDescription() {
        return description;
    }
    public void setDescription(String description) {
```

```
        this.description = description;
    }
    public Date getStartDate() {
        return startDate;
    }
    public void setStartDate(Date startDate) {
        this.startDate = startDate;
    }
    public Date getEndDate() {
        return endDate;
    }
    public void setEndDate(Date endDate) {
        this.endDate = endDate;
    }
    public TaskResourceTO getAssignedResource() {
        return assignedResource;
    }
    public void setAssignedResource(TaskResourceTO assignedResource) {
        this.assignedResource = assignedResource;
    }
}
public class ProjectManagerTO {

    private String name;
    private String address;
    private String projects;
    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;
    }
    public String getProjects() {
        return projects;
    }
    public void setProjects(String projects) {
        this.projects = projects;
    }

}
public class ProjectTO {

    private String projectId;
    private String projectName;
```

```
    private String projectDesc;
    public String getProjectId() {
        return projectId;
    }
    public void setProjectId(String projectId) {
        this.projectId = projectId;
    }
    public String getProjectName() {
        return projectName;
    }
    public void setProjectName(String projectName) {
        this.projectName = projectName;
    }
    public String getProjectDesc() {
        return projectDesc;
    }
    public void setProjectDesc(String projectDesc) {
        this.projectDesc = projectDesc;
    }
}
```

Step 3: Implementing the *Transfer Object Assembler*.
The *ProjectDetailsAssembler* class that assembles the *ProjectDetailsData* object here.

```
public class ProjectDetailsAssembler {

    public ProjectDetailsData getData(String projectId) {

        // Construct the composite transfer object
        // get project related information from database and set to ProjectDetailsData class object.
        ProjectTO projectData = new ProjectTO();

        // get ProjectManager info and add to ProjectDetailsData
        ProjectManagerTO projectManagerData = new ProjectManagerTO();

        // construct a new TaskResourceTO using Task and Resource data.
        //get the Resource details from database.
        // construct a list of TaskResourceTOs
        Collection < TaskResourceTO > listOfTasks = new ArrayList < > ();


        // Add Project Info to ProjectDetailsData
        // Add ProjectManager info to ProjectDetailsData
        ProjectDetailsData pData = new ProjectDetailsData(projectData, projectManagerData,
listOfTasks);
        return pData;
    }
}
```

Consequences

- Separates business logic, simplifies client logic
- Reduces coupling between clients and the application model
- Improves network performance
- Improves client performance
- Can introduce stale data

J2EE Design Pattern : Integration Tier Patterns : Data Access Object Design Pattern
Data Access Object abstracts data sources provides transparent access to data.

Problem

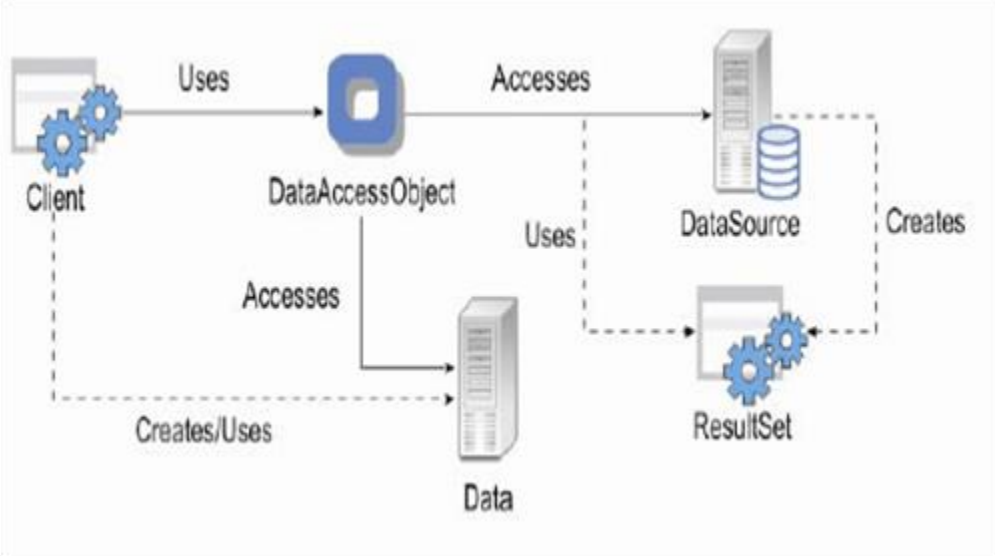
You want to encapsulate data access and manipulation in a separate layer.
Mingling persistence logic with application logic creates a direct dependency between the application and the persistence storage implementation. Such code dependencies in components make it difficult and tedious to migrate the application from one type of data source to another.

Context

Data Access Object, Service Activator, Domain Store, Web Service Broker.

Solution

Use a Data Access Object to abstract and encapsulate all access to the persistent store. The Data Access Object manages the connection with the data source to obtain and store data. The DAO completely hides the data source implementation details from its clients. Because the interface exposed by the DAO to clients does not change when the underlying data source implementation changes, this allows you to change a DAO implementation without changing the DAO client's implementation.



Participants and Collaborations

Client
The Client is an object that requires access to the data source to obtain and store data. The Client can be a Business Object, a Session Façade, an Application Services, a Value List Handler, a Transfer Object Assembler, or any other helper object that needs access to persistent data.

DataAccessObject
The DataAccessObject is the primary role object of this pattern. The DataAccessObject abstracts the underlying data access implementation for the Client to enable transparent access to the data source. The DataAccessObject implements create (insert), find (load), update (store), and delete operations.

DataSource
The DataSource represents a data source implementation. A DataSource could be a database, such as an RDBMS, OODBMS, XML repository, flat file system, and so on. A DataSource can also be another system (legacy/mainframe), service (B2B service or credit card bureau), or some kind of repository (LDAP).

ResultSet
The ResultSet represents the results of a query execution. For an RDMBS DataSource, when an application is using JDBC API, this role is fulfilled by an instance of the java.sql.ResultSet.

Data
The Data represents a transfer object used as a data carrier. The DataAccessObject can use a Transfer Object to return data to the client. The DataAccessObject could also receive the data from the client as a Transfer Object to update the data in the data source.

Implementation Strategies

- Custom Data Access Object Strategy
- Data Access Object Factory Strategies
- Transfer Object Collection Strategy
- Cached Rowset Strategy
- Read Only Rowset Strategy