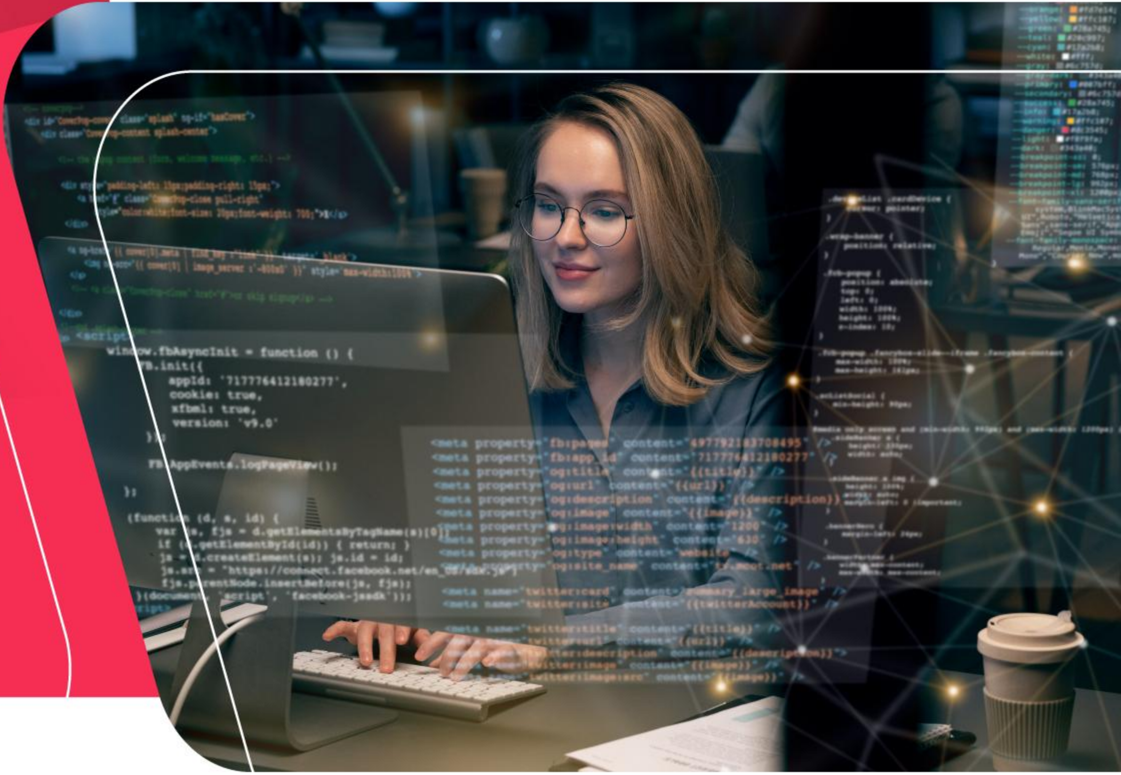
TIA-Academy

Java Bootcamp

Day 31



Technologies will be Use

- JDK 8/**11**/15
- JRE 8/**11**/15
- **Intellij IDEA Community Edition**
- JAVA 3rd Party Library (Network, DB, etc)
- **MySQL Server Community**
- MySQL Workbench or SQLyog (Windows)

TIA-Academy

Design Pattern

Design Patterns in Java

- A design patterns are **well-proved solution** for solving the specific problem/task.
- Now, a question will be arising in your mind what kind of specific problem? Let me explain by taking an example.
  - **Problem Given:**

    Suppose you want to create a class for which only a single instance (or object) should be created and that single object can be used by all other classes.
  - **Solution:**

    Singleton design pattern is the best solution of above specific problem. So, every design pattern has **some specification or set of rules** for solving the problems. What are those specifications, you will see later in the types of design patterns.

Design Patterns in Java

- But remember one-thing, design patterns are programming language independent strategies for solving the common object-oriented design problems. That means, **a design pattern represents an idea, not a particular implementation.**
- By using the design patterns you can make your code more flexible, reusable and maintainable. It is the most important part because java internally follows design patterns.
- To become a professional software developer, you must know at least some popular solutions (i.e. design patterns) to the coding problems.

TIA-Academy

Advantage of design pattern:

1. They are reusable in **multiple projects**.
2. They provide the solutions that help to define the **system architecture**.
3. They capture the **software engineering experiences**.
4. They provide **transparency** to the design of an application.
5. They are **well-proved and testified solutions** since they have been built upon the **knowledge and experience of expert software developers.**
6. Design patterns **don’t guarantee an absolute solution to a problem**. They provide **clarity to the system architecture and the possibility of building a better system**.

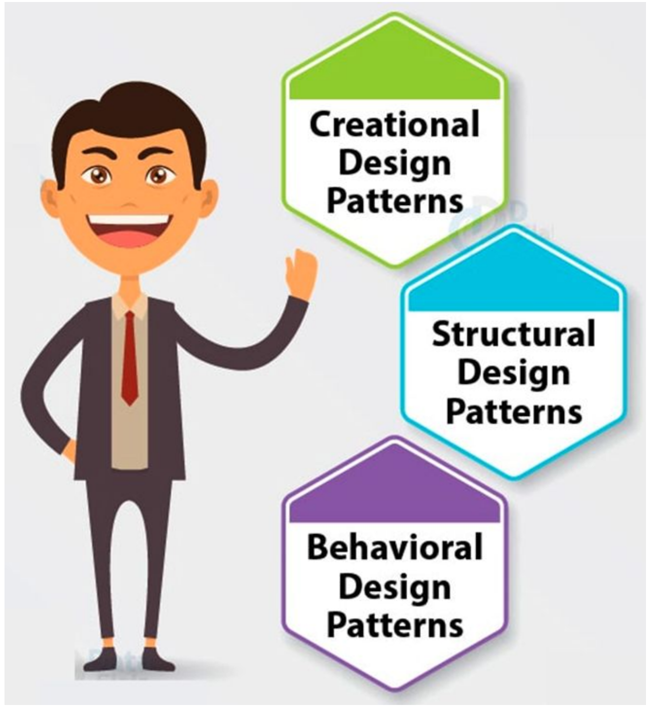
TIA-Academy

When should we use the design patterns?

- We must use the design patterns **during the analysis and requirement phase of SDLC**(Software Development Life Cycle).
- Design patterns ease the analysis and requirement phase of SDLC by providing information based on prior hands-on experiences.

Java Design Patterns

- In core java, there are mainly three types of design patterns, which are further divided into their sub-parts:


Creational Design Patterns
Structural Design Patterns
Behavioral Design Patterns

TIA-Academy

Creational Design Pattern

1. **Factory Pattern**
2. **Abstract Factory Pattern**
3. **Singleton Pattern**
4. Prototype Pattern
5. Builder Pattern.

Structural Design Pattern

1. **Adapter Pattern**
2. Bridge Pattern
3. Composite Pattern
4. Decorator Pattern
5. Facade Pattern
6. Flyweight Pattern
7. Proxy Pattern

TIA-Academy

Behavioral Design Pattern

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

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:

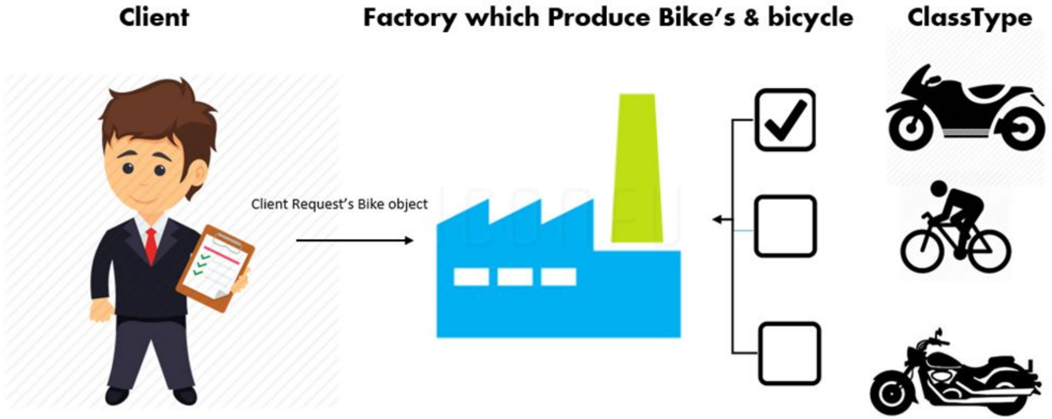
```
StudentRecord s1=new StudentRecord();
```

- **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.

TIA-Academy

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.**


Client
Factory which Produce Bike's & bicycle
ClassType
Client Request's Bike object

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.

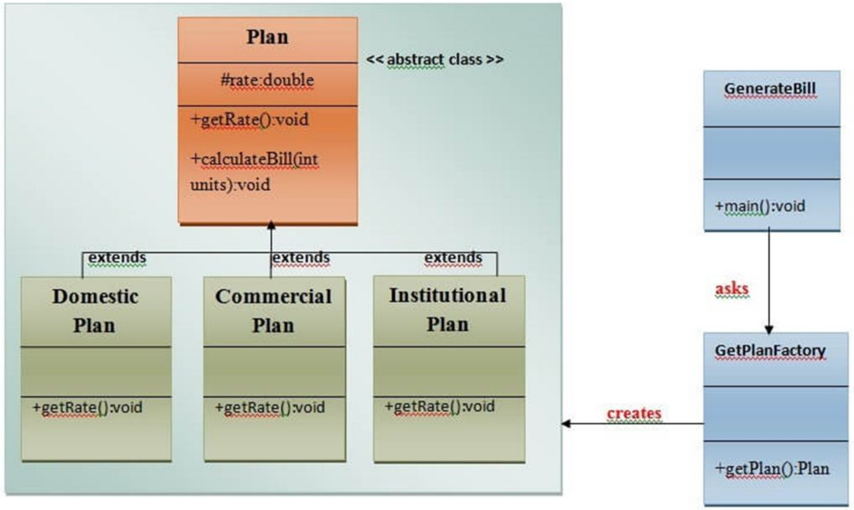
Usage of Factory Design Pattern

- 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.

TIA-Academy

UML for Factory Method Pattern

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



Calculate Electricity Bill : A Real World Example of Factory Method

- **Step 1:** Create a Plan abstract class.

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

         public void calculateBill(int units){
              System.out.println(units*rate);
          }
}//end of Plan class.
```

Calculate Electricity Bill : A Real World Example of Factory Method

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

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


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


class  InstitutionalPlan extends Plan{

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

TIA-Academy

Calculate Electricity Bill : A Real World Example of Factory Method

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

```
class GetPlanFactory{

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

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

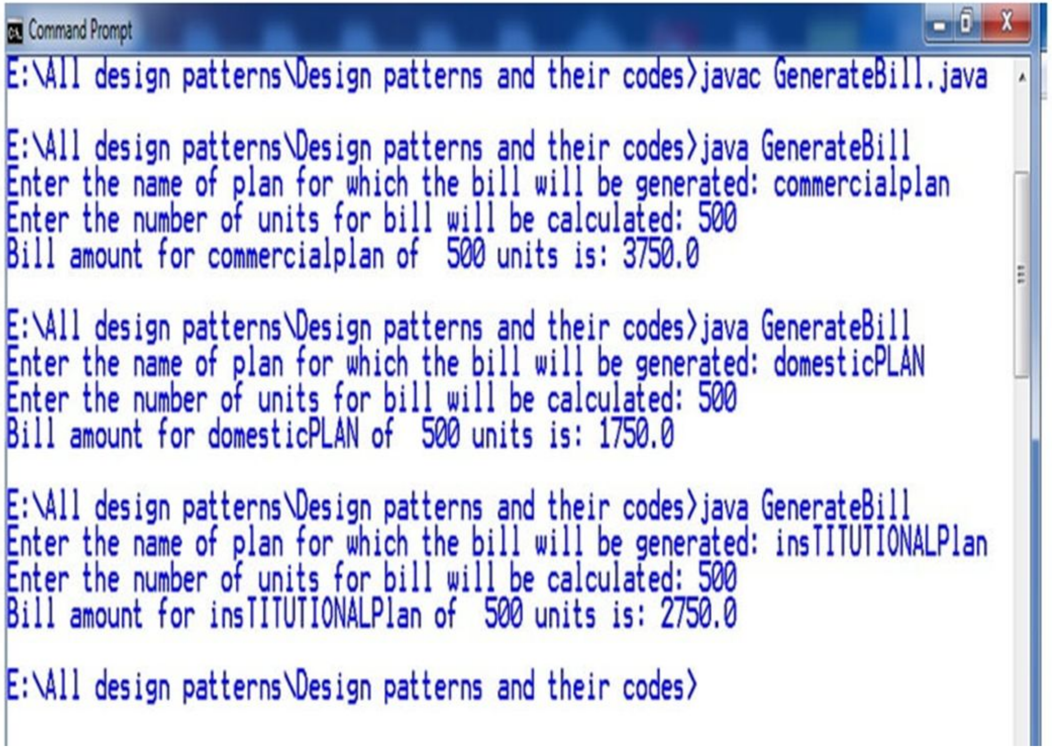
Calculate Electricity Bill : A Real World Example of Factory Method

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

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

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

      String planName=br.readLine();
      System.out.print("Enter the number of units for bill will be calculated: ");
      int units=Integer.parseInt(br.readLine());

      Plan p = planFactory.getPlan(planName);
      //call getRate() method and calculateBill()method of DomesticPaln.

       System.out.print("Bill amount for "+planName+" of  "+units+" units is: ");
           p.getRate();
           p.calculateBill(units);
            }
    }//end of GenerateBill class.
```

TIA-Academy

Calculate Electricity Bill : A Real World Example of Factory Method



Command Prompt

```
E:\All design patterns\Design patterns and their codes>javac GenerateBill.java

E:\All design patterns\Design patterns and their codes>java GenerateBill
Enter the name of plan for which the bill will be generated: commercialplan
Enter the number of units for bill will be calculated: 500
Bill amount for commercialplan of  500 units is: 3750.0

E:\All design patterns\Design patterns and their codes>java GenerateBill
Enter the name of plan for which the bill will be generated: domesticPLAN
Enter the number of units for bill will be calculated: 500
Bill amount for domesticPLAN of  500 units is: 1750.0

E:\All design patterns\Design patterns and their codes>java GenerateBill
Enter the name of plan for which the bill will be generated: insTITUTIONALPlan
Enter the number of units for bill will be calculated: 500
Bill amount for insTITUTIONALPlan of  500 units is: 2750.0

E:\All design patterns\Design patterns and their codes>
```

TIA-Academy

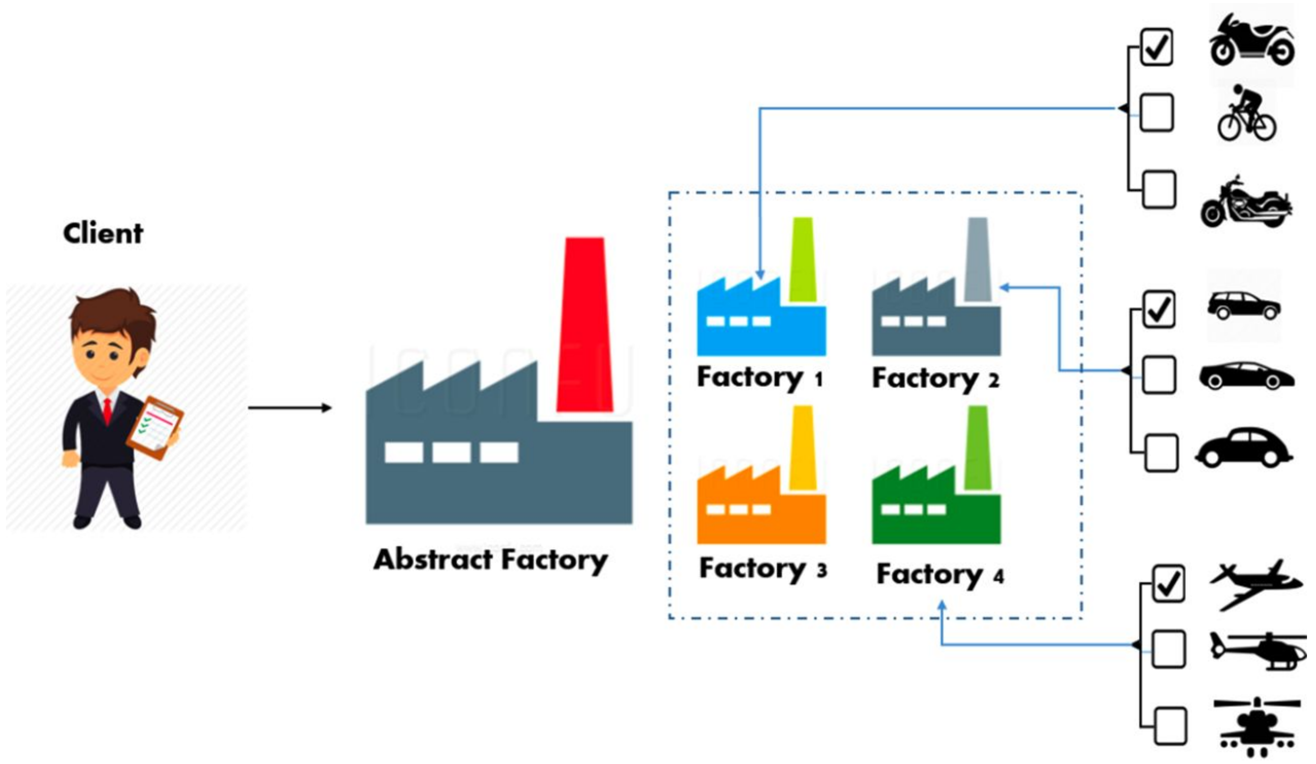
ASSIGNMENT 01

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.**

TIA-Academy

Abstract Factory Pattern



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.

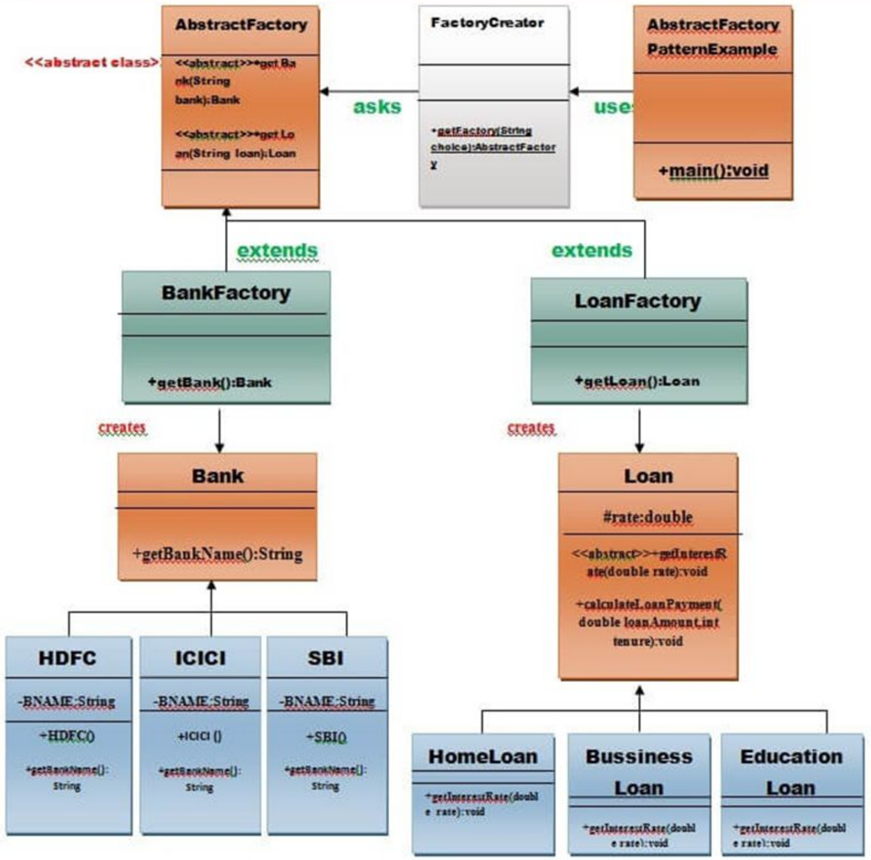
Usage of Abstract Factory Pattern

- 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.

TIA-Academy

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 beside:



Example of Abstract Factory Pattern

- Here, we are calculating the loan payment for different banks like HDFC, ICICI, SBI etc.
- **Step 1:** Create a Bank interface

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

TIA-Academy

Example of Abstract Factory Pattern

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

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

TIA-Academy

Example of Abstract Factory Pattern

- **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 borrowed");
   }
}// end of the Loan abstract class.
```

Example of Abstract Factory Pattern

- **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.
```

TIA-Academy

Example of Abstract Factory Pattern

- **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);
}
```

Example of Abstract Factory Pattern

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

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

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

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

}
```

Example of Abstract Factory Pattern

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

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

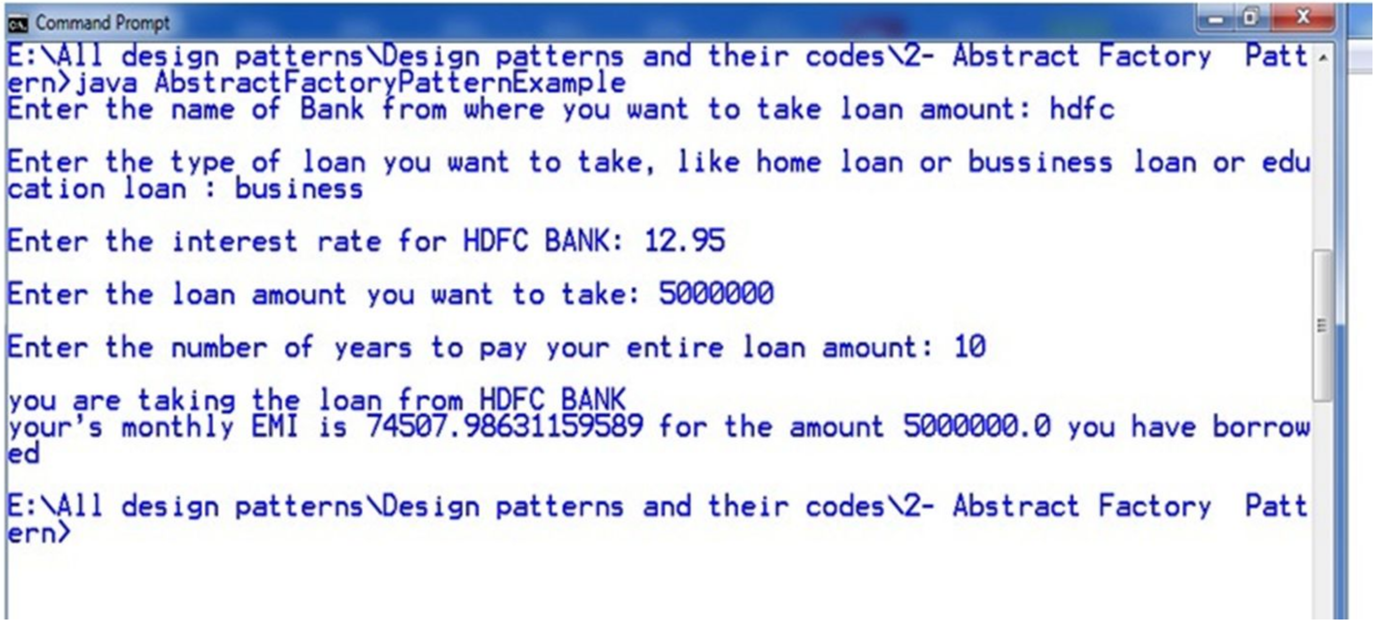
Example of Abstract Factory Pattern

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

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

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

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

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

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

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

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

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

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

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

TIA-Academy

Example of Abstract Factory Pattern



```
E:\All design patterns\Design patterns and their codes\2- Abstract Factory  Patt
ern>java AbstractFactoryPatternExample
Enter the name of Bank from where you want to take loan amount: hdfc

Enter the type of loan you want to take, like home loan or bussiness loan or edu
cation loan : business

Enter the interest rate for HDFC BANK: 12.95

Enter the loan amount you want to take: 5000000

Enter the number of years to pay your entire loan amount: 10

you are taking the loan from HDFC BANK
your's monthly EMI is 74507.98631159589 for the amount 5000000.0 you have borrow
ed

E:\All design patterns\Design patterns and their codes\2- Abstract Factory  Patt
ern>
```

TIA-Academy

ASSIGNMENT 02

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.

Singleton design pattern in Java

Advantage of Singleton design pattern

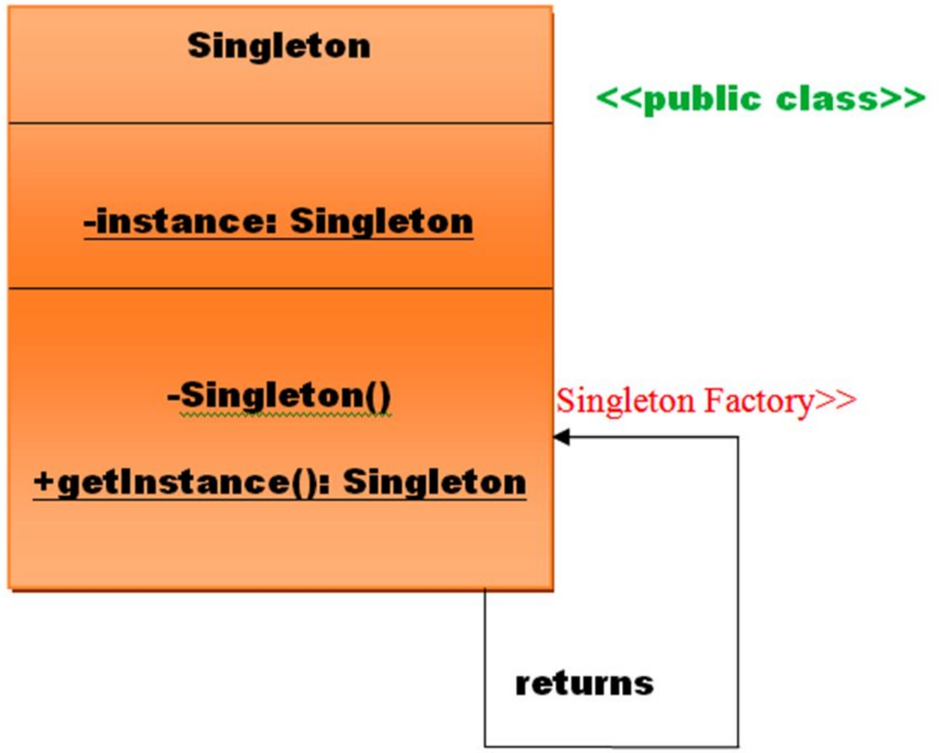
- **Saves memory because object is not created at each request.** Only single instance is reused again and again.

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.

TIA-Academy

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, it contains 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.

TIA-Academy

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.

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

 public static A getA(){
  return obj;
 }

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

Understanding 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.

```
class A{
 private static A obj;
 private A(){}

 public static A getA(){
   if (obj == null){
      synchronized(Singleton.class){
        if (obj == null){
            obj = new Singleton();//instance will be created at request time
        }
    }
    }
  return obj;
 }

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

Significance of Serialization in Singleton

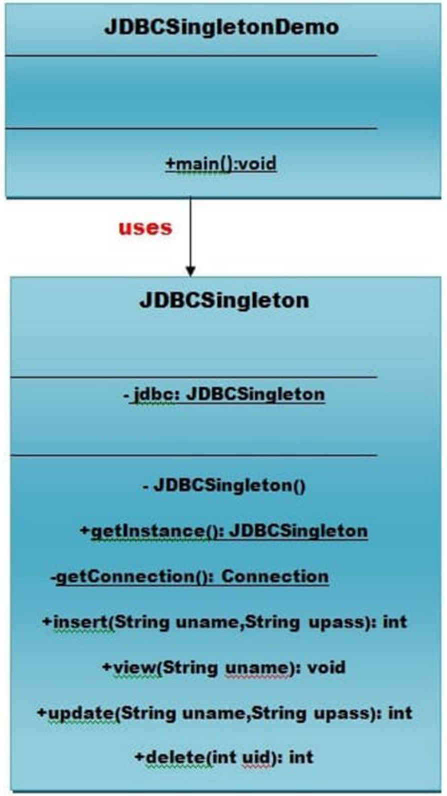
- If singleton class is Serializable, you can serialize the singleton instance.
- Once it is serialized, you can deserialize it but it will not return the singleton object.
- To resolve this issue, you need to override the readResolve() method that enforces the singleton.
- It is called just after the object is deserialized. It returns the singleton object.

```
public class A implements Serializable {
        //your code of singleton
        protected Object readResolve() {
            return getA();
        }

}
```

TIA-Academy

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.


JDBCSingletonDemo
+main():void
uses
JDBCSingleton
- jdbc: JDBCSingleton
- JDBCSingleton()
+getInstance(): JDBCSingleton
-getConnection(): Connection
+insert(String uname,String upass): int
+view(String uname): void
+update(String uname,String upass): int
+delete(int uid): int

Understanding Real Example of Singleton Pattern

- **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.

JDBCSingletonDemo.java

TIA-Academy

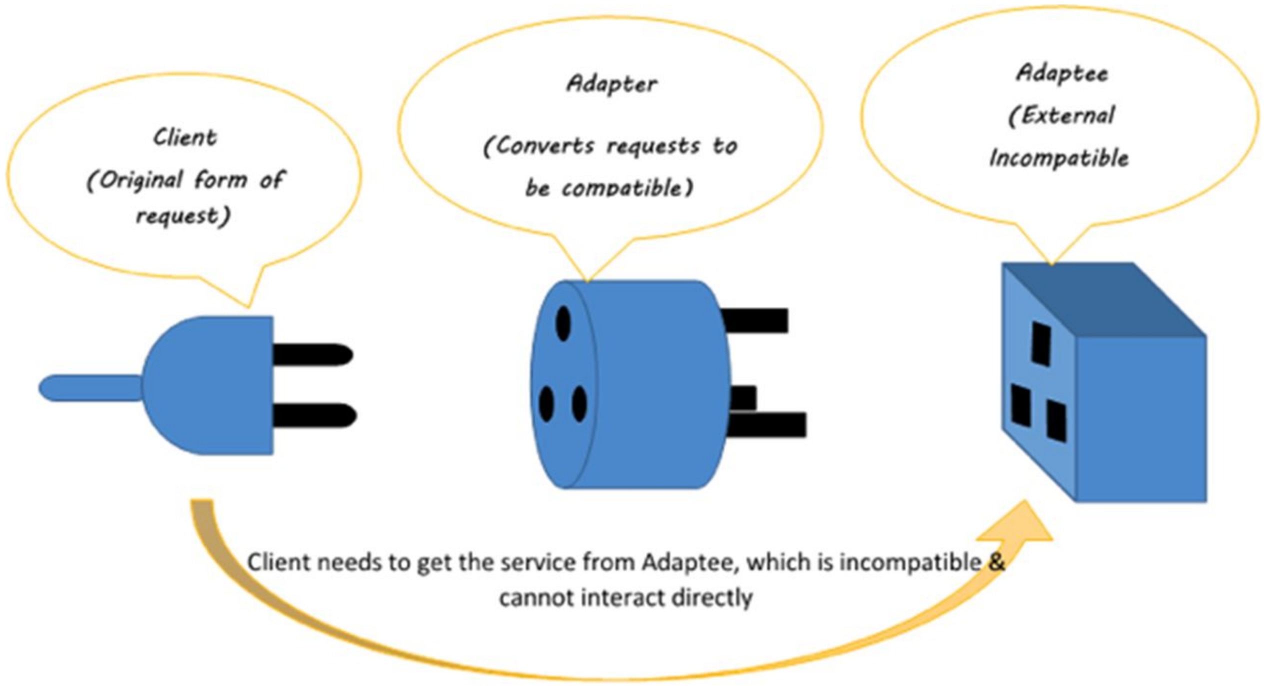
Structural design patterns

- **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.

Adapter Pattern

- An Adapter Pattern says that just **"converts the interface of a class into another interface that a client wants".**
- In other words, to provide the interface according to client requirement while using the services of a class with a different interface.
- The Adapter Pattern is also known as **Wrapper.**

Adapter Pattern


Client
(Original form of request)
Adapter
(Converts requests to be compatible)
Adaptee
(External Incompatible
Client needs to get the service from Adaptee, which is incompatible & cannot interact directly

TIA-Academy

Advantage of Adapter Pattern

- It allows two or more previously incompatible objects to interact.
- It allows reusability of existing functionality.

TIA-Academy

Usage of Adapter pattern

- When an object needs to utilize an existing class with an incompatible interface.
- When you want to create a reusable class that cooperates with classes which don't have compatible interfaces.
- When you want to create a reusable class that cooperates with classes which don't have compatible interfaces.

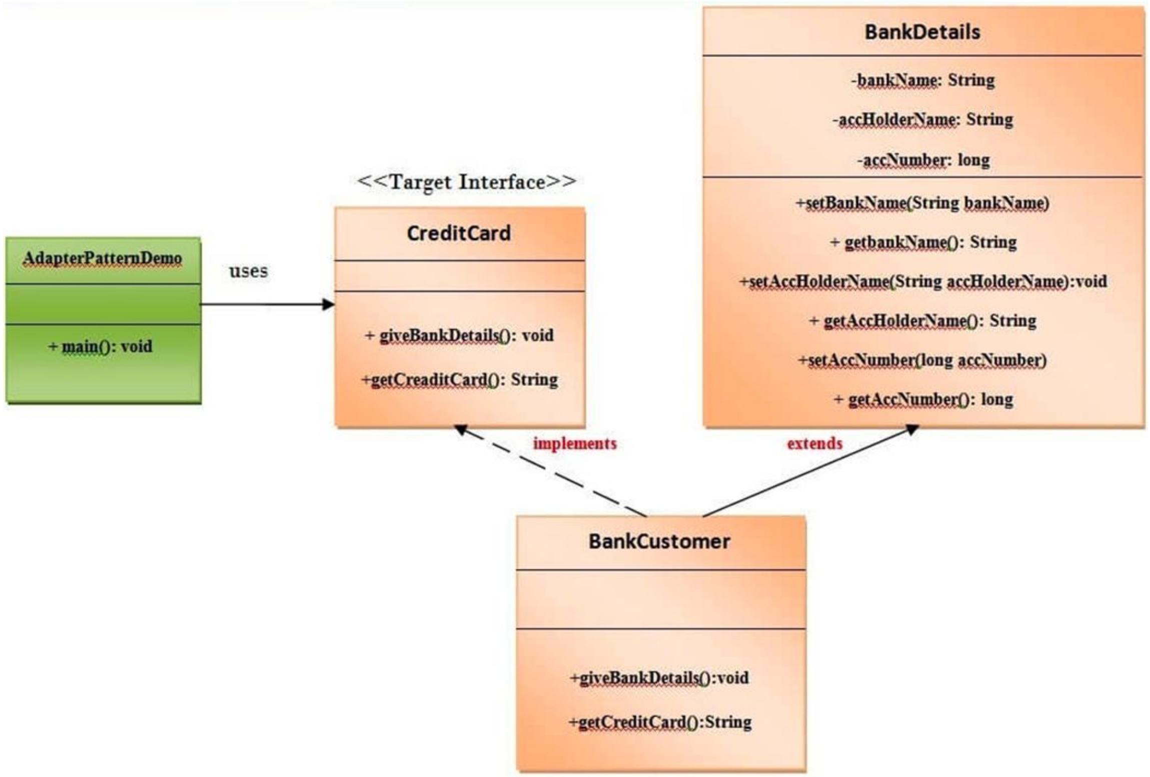
UML for Adapter Pattern

There are the following specifications for the adapter pattern:

- **Target Interface:** This is the desired interface class which will be used by the clients.
- **Adapter class:** This class is a wrapper class which implements the desired target interface and modifies the specific request available from the Adaptee class.
- **Adaptee class:** This is the class which is used by the Adapter class to reuse the existing functionality and modify them for desired use.
- **Client:** This class will interact with the Adapter class.

TIA-Academy

UML for Adapter Pattern



Step 1

- Create a **CreditCard** interface (Target interface).

```
public interface CreditCard {
    public void giveBankDetails();
    public String getCreditCard();
}// End of the CreditCard interface.
```

Step 2

- Create a **BankDetails** class (Adaptee class).

```
public class BankDetails{
    private String bankName;
    private String accHolderName;
    private long accNumber;

    public String getBankName() {
        return bankName;
    }
    public void setBankName(String bankName) {
        this.bankName = bankName;
    }
    public String getAccHolderName() {
        return accHolderName;
    }
    public void setAccHolderName(String accHolderName) {
        this.accHolderName = accHolderName;
    }
    public long getAccNumber() {
        return accNumber;
    }
    public void setAccNumber(long accNumber) {
        this.accNumber = accNumber;
    }
}// End of the BankDetails class.
```

TIA-Academy

Step 3

- Create a **BankCustomer** class (Adapter class).

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
public class BankCustomer  extends BankDetails  implements CreditCard {
 public void giveBankDetails(){
  try{
   BufferedReader br= new BufferedReader( new InputStreamReader(System.in));

   System.out.print("Enter the account holder name :");
   String customername=br.readLine();
   System.out.print("\n");

   System.out.print("Enter the account number:");
   long accno=Long.parseLong(br.readLine());
   System.out.print("\n");

   System.out.print("Enter the bank name :");
   String bankname=br.readLine();
   setAccHolderName(customername);
   setAccNumber(accno);
   setBankName(bankname);
   }catch(Exception e){
        e.printStackTrace();
   }
  }
  @Override
  public String getCreditCard()  {
   long accno=getAccNumber();
   String accholdername=getAccHolderName();
   String bname=getBankName();

   return ("The Account number "+accno+" of "+accholdername+" in "+bname+ "
                     bank is valid and authenticated    for issuing the credit card. ");
  }
}//End of the BankCustomer class.
```

Step 4

- Create a **AdapterPatternDemo** class (client class).

```
public class AdapterPatternDemo {
 public static void main(String args[]){
  CreditCard targetInterface=new BankCustomer();
  targetInterface.giveBankDetails();
  System.out.print(targetInterface.getCreditCard());
 }
}//End of the BankCustomer class.
```

Output

```
Enter the account holder name :Sonoo Jaiswal

Enter the account number:10001

Enter the bank name :State Bank of India

The Account number 10001 of Sonoo Jaiswal in State Bank of India bank is valid
and authenticated for issuing the credit card.
```

TIA-Academy

ASSIGNMENT 03

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.

TIA-Academy

Chain Of Responsibility Pattern

- 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.

Chain Of Responsibility Pattern

- 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.

TIA-Academy

Advantage of Chain of Responsibility Pattern

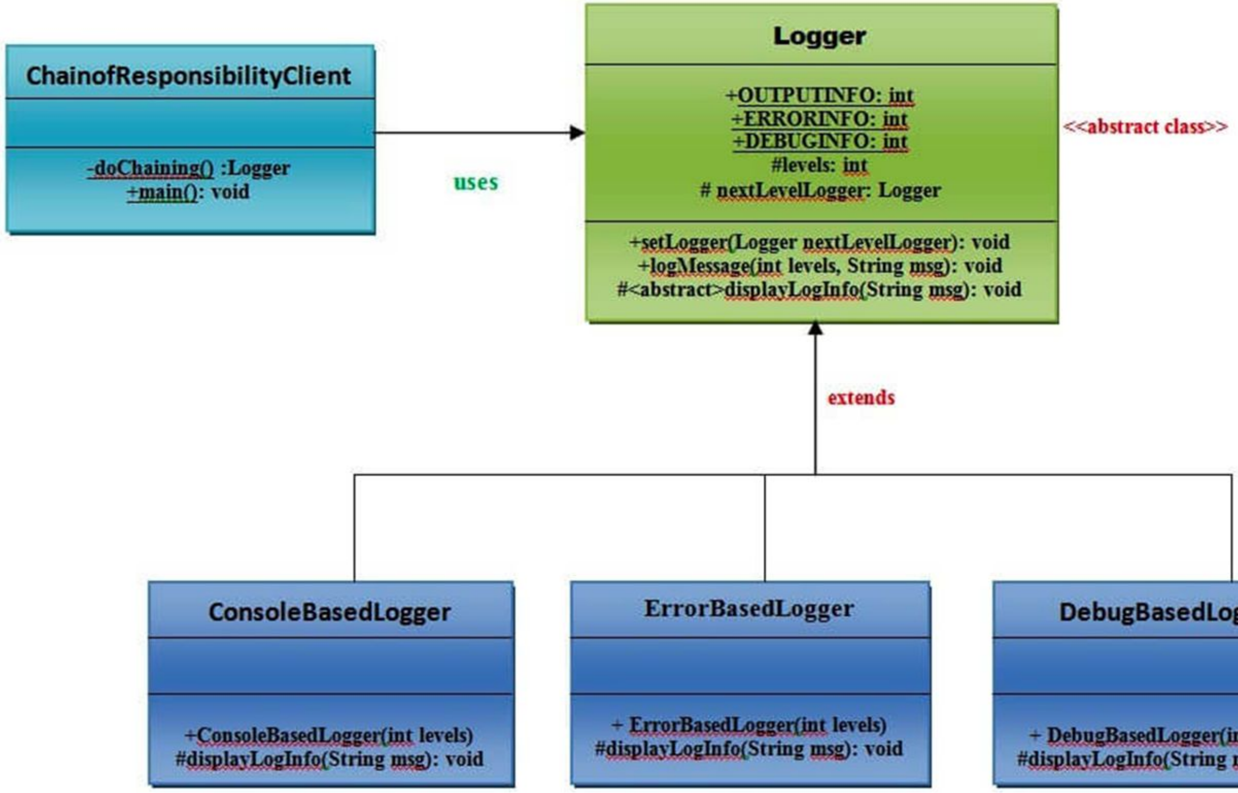
- It reduces the coupling.
- It adds flexibility while assigning the responsibilities to objects.
- It allows a set of classes to act as one; events produced in one class can be sent to other handler classes with the help of composition.

Usage of Chain of Responsibility Pattern

- 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.

TIA-Academy

UML for Chain of Responsibility Pattern


ChainofResponsibilityClient
-doChaining() :Logger
+main(): void
uses
Logger
+OUTPUTINFO: int
+ERRORINFO: int
+DEBUGINFO: int
#levels: int
# nextLevelLogger: Logger
+setLogger(Logger nextLevelLogger): void
+logMessage(int levels, String msg): void
#<abstract>displayLogInfo(String msg): void
<<abstract class>>
extends
ConsoleBasedLogger
+ConsoleBasedLogger(int levels)
#displayLogInfo(String msg): void
ErrorBasedLogger
+ ErrorBasedLogger(int levels)
#displayLogInfo(String msg): void
DebugBasedLogger
+ DebugBasedLogger(int levels)
#displayLogInfo(String msg): void

TIA-Academy

Step 1

- Create a **Logger** abstract class.

```
public abstract class Logger {
    public static int OUTPUTINFO=1;
    public static int ERRORINFO=2;
    public static int DEBUGINFO=3;
    protected int levels;
    protected Logger nextLevelLogger;
    public void setNextLevelLogger(Logger nextLevelLogger) {
        this.nextLevelLogger = nextLevelLogger;
    }
        public void logMessage(int levels, String msg){
        if(this.levels<=levels){
            displayLogInfo(msg);
        }
        if (nextLevelLogger!=null) {
            nextLevelLogger.logMessage(levels, msg);
        }
    }
    protected abstract void displayLogInfo(String msg);
}
```

TIA-Academy

Step 2

- Create a **ConsoleBasedLogger** class.

```
public class ConsoleBasedLogger extends Logger {
    public ConsoleBasedLogger(int levels) {
        this.levels=levels;
    }
    @Override
    protected void displayLogInfo(String msg) {
        System.out.println("CONSOLE LOGGER INFO: "+msg);
    }
}
```

TIA-Academy

Step 3

- Create a **DebugBasedLogger** class.

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

Step 4

- Create a **ErrorBasedLogger** class.

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

TIA-Academy

- Create a **ChainOfResponsibilityClient** class.

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

          Logger errorLogger =new ErrorBasedLogger(Logger.ERRORINFO);
          consoleLogger.setNextLevelLogger(errorLogger);

          Logger debugLogger =new DebugBasedLogger(Logger.DEBUGINFO);
          errorLogger.setNextLevelLogger(debugLogger);

          return consoleLogger;
          }
          public static void main(String args[]){
          Logger chainLogger= doChaining();

              chainLogger.logMessage(Logger.OUTPUTINFO, "Enter the sequence of values ");
              chainLogger.logMessage(Logger.ERRORINFO, "An error is occured now");
              chainLogger.logMessage(Logger.DEBUGINFO, "This was the error now debugging is compeled");
              }
}
```

Output

```
bilityClient
CONSOLE LOGGER INFO: Enter the sequence of values
CONSOLE LOGGER INFO: An error is occured now
ERROR LOGGER INFO: An error is occured now
CONSOLE LOGGER INFO: This was the error now debugging is compeled
ERROR LOGGER INFO: This was the error now debugging is compeled
DEBUG LOGGER INFO: This was the error now debugging is compeled
```

TIA-Academy

ASSIGNMENT 04 (HOME ASSIGNMENT)

TIA-Academy

Thank You