Design Patterns in Java

A design patterns are **well-proved solution** for solving the specific problem/task.

1) Creational design patterns
2) Structural design patterns
3) Behavioral Design Patterns

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".**

To use an adapter:

1. The client makes a request to the adapter by calling a method on it using the target interface.
2. The adapter translates that request on the adaptee using the adaptee interface.
3. Client receive the results of the call and is unaware of adapter's presence.

The Adapter Pattern is also known as **Wrapper.**

Advantage of Adapter Pattern

1) It allows two or more previously incompatible objects to interact.
2) It allows reusability of existing functionality.

Step 1

Create a CreditCard interface (Target interface).

```
1. public interface CreditCard {
2.     public void giveBankDetails();
3.     public String getCreditCard();
4. }// End of the CreditCard interface.
5.
```

Step 2

Create a BankDetails class (Adaptee class).

File: BankDetails.java

```
1. // This is the adapter class.
2. public class BankDetails{
3.     private String bankName;
4.     private String accHolderName;
5.     private long accNumber;
6.
7.     public String getBankName() {
8.         return bankName;
9.     }
10.    public void setBankName(String bankName) {
11.        this.bankName = bankName;
12.    }
13.    public String getAccHolderName() {
14.        return accHolderName;
15.    }
16.    public void setAccHolderName(String accHolderName) {
17.        this.accHolderName = accHolderName;
18.    }
19.    public long getAccNumber() {
20.        return accNumber;
21.    }
22.    public void setAccNumber(long accNumber) {
23.        this.accNumber = accNumber;
```

```
24.    }
25.}// End of the BankDetails class.
26.
```

Step 3

Create a BankCustomer class (Adapter class).

File: BankCustomer.java

```
1.  // This is the adapter class
2.
3.  import java.io.BufferedReader;
4.  import java.io.InputStreamReader;
5.  public class BankCustomer extends BankDetails implements CreditCard {
6.   public void giveBankDetails(){
7.    try{
8.    BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
9.
10.   System.out.print("Enter the account holder name :");
11.   String customername=br.readLine();
12.   System.out.print("\n");
13.
14.   System.out.print("Enter the account number:");
15.   long accno=Long.parseLong(br.readLine());
16.   System.out.print("\n");
17.
18.   System.out.print("Enter the bank name :");
19.   String bankname=br.readLine();
20.
21.   setAccHolderName(customername);
22.   setAccNumber(accno);
23.   setBankName(bankname);
24.   }catch(Exception e){
25.       e.printStackTrace();
26.   }
27.  }
28.  @Override
29.  public String getCreditCard() {
```

```
30.  long accno=getAccNumber();
31.  String accholdername=getAccHolderName();
32.  String bname=getBankName();
33.
34.  return ("The Account number "+accno+" of "+accholdername+" in "+bname
35.          + " bank is valid and authenticated for issuing the credit card. ");
36.  }
37.}//End of the BankCustomer class.
38.
```

Step 4

Create a AdapterPatternDemo class (client class).

File*: *AdapterPatternDemo.java

```
1.  //This is the client class.
2.  public class AdapterPatternDemo {
3.   public static void main(String args[]){
4.    CreditCard targetInterface=new BankCustomer();
5.    targetInterface.giveBankDetails();
6.    System.out.print(targetInterface.getCreditCard());
7.   }
8.  }//End of the BankCustomer class.
9.
```

Output

```
1.  Enter the account holder name :Sonoo Jaiswal
2.
3.  Enter the account number:10001
4.
5.  Enter the bank name :State Bank of India
6.
7.  The Account number 10001 of Sonoo Jaiswal in State Bank of India bank is vali
    d  and authenticated for issuing the credit card.
```

The client sees only the target interface and not the adapter. The adapter implements the target interface. Adapter delegates all requests to Adaptee.

Example:

Suppose you have a Bird class with fly() , and makeSound()methods. And also a ToyDuck class with squeak() method. Let's assume that you are short on ToyDuck objects and you would like to use Bird objects in their place. Birds have some similar functionality but implement a different interface, so we can't use them directly. So we will use adapter pattern. Here our client would be ToyDuck and adaptee would be Bird.

Below is Java implementation of it.

// Java implementation of Adapter pattern

```
interface Bird
{
   // birds implement Bird interface that allows
   // them to fly and make sounds adaptee interface
   public void fly();
   public void makeSound();
}

class Sparrow implements Bird
{
   // a concrete implementation of bird
   public void fly()
   {
      System.out.println("Flying");
   }
   public void makeSound()
   {
      System.out.println("Chirp Chirp");
   }
}
```

```
interface ToyDuck
{
   // target interface
   // toyducks dont fly they just make
   // squeaking sound
   public void squeak();
}

class PlasticToyDuck implements ToyDuck
{
   public void squeak()
   {
      System.out.println("Squeak");
   }
}

class BirdAdapter implements ToyDuck
{
   // You need to implement the interface your
   // client expects to use.
   Bird bird;
   public BirdAdapter(Bird bird)
   {
      // we need reference to the object we
      // are adapting
      this.bird = bird;
   }

   public void squeak()
   {
      // translate the methods appropriately
      bird.makeSound();
   }
}
```

```
class Main
{
   public static void main(String args[])
   {
      Sparrow sparrow = new Sparrow();
      ToyDuck toyDuck = new PlasticToyDuck();

      // Wrap a bird in a birdAdapter so that it
      // behaves like toy duck
      ToyDuck birdAdapter = new BirdAdapter(sparrow);

      System.out.println("Sparrow...");
      sparrow.fly();
      sparrow.makeSound();

      System.out.println("ToyDuck...");
      toyDuck.squeak();

      // toy duck behaving like a bird
      System.out.println("BirdAdapter...");
      birdAdapter.squeak();
   }
}
```

Output:

```
Sparrow...
Flying
Chirp Chirp
ToyDuck...
Squeak
BirdAdapter...
Chirp Chirp
```

Explanation:

Suppose we have a bird that can makeSound(), and we have a plastic toy duck that can squeak(). Now suppose our client changes the requirement and he wants the toyDuck to makeSound than ?

Simple solution is that we will just change the implementation class to the new adapter class and tell the client to pass the instance of the bird(which wants to squeak()) to that class.

Before : ToyDuck toyDuck = new PlasticToyDuck();

After : ToyDuck toyDuck = new BirdAdapter(sparrow);

You can see that by changing just one line the toyDuck can now do Chirp Chirp !!