**Adapter Design Pattern in Java**

The Adapter Design Pattern is a powerful and essential tool for software developers. It helps to create flexible and maintainable code by allowing objects with incompatible interfaces to work together seamlessly. In this comprehensive guide, we will explore the Adapter Design Pattern in depth, covering its key concepts, benefits, and real-world examples. We will also look at its implementations and how it can be used to solve common software design problems.



**What is Adapter Design Pattern -:**

***Adapter design pattern is one of the structural design pattern and it is used so that two unrelated interfaces can work together.*** It is often used to make existing classes work with others without modifying their source code. The pattern involves creating an adapter class that bridges the gap between the interfaces, allowing them to communicate effectively.

One real-world example of the **Adapter Design Pattern** is a mobile charger. Mobile chargers act as adapters between the power socket and the mobile device, converting the voltage to a suitable level for charging the device. In software development, the Adapter Design Pattern follows the same concept by introducing an additional adapter class between an existing interface and a class that needs to work with it.

**The Problem the Adapter Design Pattern Solves -:**

The Adapter Design Pattern addresses several common issues in software development:

1. How can a class be reused if it does not have an interface that a client requires?
2. How can classes with incompatible interfaces work together?
3. How can an alternative interface be provided for a class?

The Adapter Design Pattern solves this problem by defining a separate adapter class that converts the incompatible interface of a class (called the “adaptee”) into another interface (called the “target”) that clients require.

**The Components of the Adapter Design Pattern -:**

The Adapter pattern involves three main components -:

a) Target: This represents the interface or class that the client code expects to work with. It defines the operations that the client can use.

b) Adaptee: This refers to the existing class or interface that needs to be adapted. It has a different interface that is incompatible with the client’s expectations.

c) Adapter: This is the class that bridges the gap between the Target and Adaptee. It implements the Target interface and internally uses an instance of the Adaptee to perform the desired operations. The Adapter class acts as a wrapper or translator, adapting the Adaptee’s interface to match the Target interface.

**Different implementations of Adapter Design Pattern -:**

There are two main approaches to implementing the Adapter Design Pattern: **the class adapter pattern and the object adapter pattern.** Both approaches produce the same result, but they use different techniques.

**Class Adapter Pattern**

In a class adapter, the Adapter class extends both the Target interface and the Adaptee class. It inherits the behavior of the Adaptee and adapts it to match the Target interface. However, this approach requires multiple inheritance, which may not be supported in some programming languages.

**Object Adapter Pattern**

In an object adapter, the Adapter class implements the Target interface and contains an instance of the Adaptee class. It delegates the calls from the Target interface to the Adaptee object, adapting its interface to match the Target interface. This approach uses composition instead of inheritance, making it more flexible and compatible with different programming languages.

**Implementation of Adapter Design Pattern -:**

Suppose you are having an IPhone6s and your friend is having an IPhone4s now you went to your friend’s house but you forgot to carry your charger with you and you need to charge your phone and the charger you need is not available . Now what will you do‍🤷‍♀️? So here in this situation an adapter is going to help us . You will use an adapter to charge your IPhone6s from IPhone4s charger . So here you have not changed the IPhone4s charger to IPhone6s charger you just adapted the situation and have your work done . But how can we implement it in code ‍🤷‍♀️? Let’s see .

**Step 1 -:**

**Define the Target interface**: This interface should specify the operations that the client code expects. So here our client that is you want to charge your phone .

IPhone Interface -:

public interface IPhone  
{  
 public void OnCharge();  
}

**Step 2 -:**

**Implement the Adaptee class:** This class represents the existing component with an incompatible interface. So this is the IPhone4s charger that your friend has .

IPhone4s Charger -:

public class Iphone4sCharger implements Charger  
{  
 Iphone4sCharger(){};  
  
 public void charge()  
 {  
 System.out.println("charging with 4s charger");  
 }  
}

Charger Interface -:

public interface Charger  
{  
 public void charge();  
}

**Step 3 -:**

**Create the Adapter class**: The Adapter class implements the Target interface and internally uses an instance of the Adaptee class. It adapts the Adaptee’s interface to match the Target interface by delegating the calls appropriately. So here we will use an IPhone4s to IPhone6s adapter so that we can easily charge our IPhone6s .

IPhone4stoIPhone6s Adapter class -:

public class Iphone4sTo6sAdapter implements Charger  
{  
 Iphone4sCharger iphone4sCharger;  
  
 Iphone4sTo6sAdapter()  
 {  
 iphone4sCharger = new Iphone4sCharger();  
 }  
  
 @Override  
 public void charge()  
 {  
 iphone4sCharger.charge();  
 }  
}

So here we have created an instance of IPhone4s Charger and used its charge() method .

**Step 4 -:**

**Connect the client code to the Adapter**: Instantiate the Adapter class and use it as a bridge between the client code and the Adaptee. So here you will connect your IPhone6s to IPhone4s Charger to charge your phone .

IPhone 6s class -:

public class IPhone6s implements IPhone  
{  
 Charger Iphone4sTo6sAdapter;  
 public IPhone6s(Charger iphone4sTo6sAdapter)  
 {  
 this.Iphone4sTo6sAdapter = iphone4sTo6sAdapter;  
 };  
  
 @Override  
 public void OnCharge()  
 {  
 Iphone4sTo6sAdapter.charge();  
 }  
}

Main class -:

public class main  
{  
 public static void main(String args[])  
 {  
 IPhone6s iphone6s = new IPhone6s(new Iphone4sTo6sAdapter());  
 iphone6s.OnCharge();  
 }  
}

So here you can clearly see that if you will directly use IPhone4s Charger to charge your phone it won’t work but using an adapter will do so for you .

**Real life examples of Adapter Design Pattern -:**

The Adapter pattern can be applied in various scenarios. Here are a few real-world examples:

a)**Database Adapters:** When working with different database systems, each may have its own specific API. An adapter can be used to convert the operations and queries from one database system to another, allowing the client code to work with a common interface.

b)**Legacy System Integration:** When integrating new software components with existing legacy systems, the Adapter pattern can be used to translate the legacy system’s interface into a more modern and compatible one.

c) **Plug Adapters:** In electrical systems, different countries may have different types of electrical outlets. Plug adapters allow devices with one type of plug to be used with different outlet types by adapting the plug to fit the specific outlet.

d) **Java’s InputStreamReader and OutputStreamWriter :**The java.io.InputStreamReader and java.io.OutputStreamWriter classes act as adapters that convert an InputStream into a Reader and an OutputStream into a Writer, respectively. These adapters allow for seamless data conversion between different formats, making it easier to work with various types of input and output streams.

e) **Java’s Arrays.asList() Method :**The java.util.Arrays.asList() method is another example of the Adapter Design Pattern in action. This method converts an array into a List object, allowing for easier manipulation and processing of the data.

**Benefits and Drawbacks of the Adapter Pattern -:**

**Benefits -:**

* Promotes code reusability by adapting existing components instead of rewriting them.
* Enables collaboration between incompatible interfaces, fostering interoperability.
* Allows the client code to work with a common interface, regardless of the underlying implementation.

**Drawbacks -:**

* Can introduce additional complexity if not used judiciously.
* May result in performance overhead due to the translation between interfaces.
* Increases the number of classes and complexity in the codebase.

**Conclusion -:**

The Adapter design pattern is a powerful tool for bridging the gap between incompatible interfaces. By providing a translation layer, it allows objects with different interfaces to collaborate and work together seamlessly. Whether you’re integrating legacy systems or working with third-party libraries, the Adapter pattern offers an elegant solution to ensure compatibility and reusability in your software projects.

So this was my knowledge on adapter design pattern that I shared with you guys hope you understood it clearly . Please follow me for more such content also don’t forget to leave some claps 👏.

To see complete code -: <https://github.com/akshatsh0610/Adapter-Design-Pattern-Low-level-Design->