# Adapter Design Pattern in Java

PANKAJ — 19 COMMENTS

Adapter design pattern is one of the **structural design pattern** and its used so that two unrelated interfaces can work together. The object that joins these unrelated interface is called an **Adapter**.

## Behavioral Design Patterns



## Miscellaneous Design Patterns

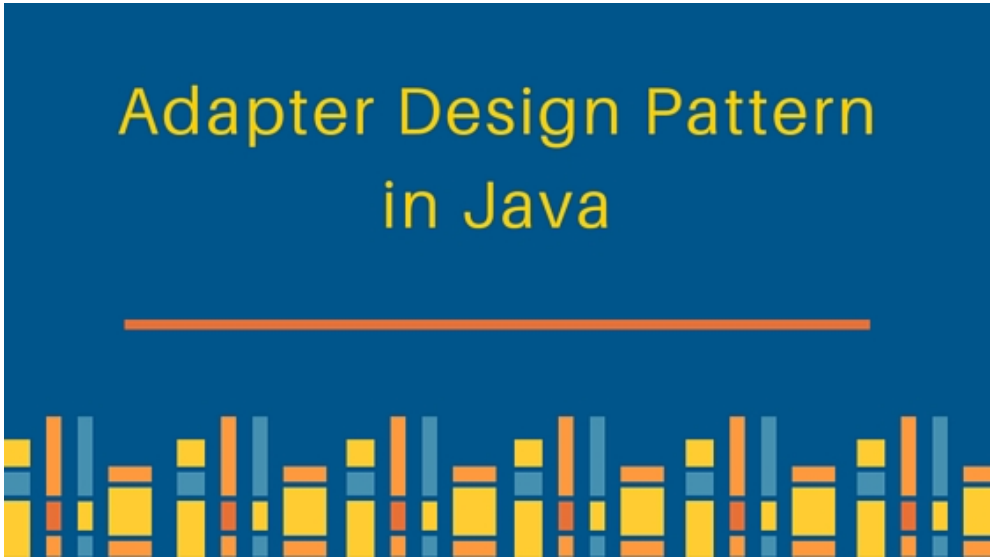


**Recommended Tutorials**





# Adapter Design Pattern




One of the great real life example of Adapter design pattern is mobile charger. Mobile battery needs 3 volts to charge but the normal socket produces either 120V (US) or 240V (India). So the mobile charger works as an adapter between mobile charging socket and the wall socket.

We will try to implement multi-adapter using adapter design pattern in this tutorial.

So first of all we will have two classes – `Volt` (to measure volts) and `Socket` (producing constant volts of 120V).

```
package com.journaldev.design.adapter;

public class Volt {

	private int volts;

	public Volt(int v){
		this.volts=v;
	}

	public int getVolts() {
		return volts;
	}

	public void setVolts(int volts) {
		this.volts = volts;
	}

}
```

```
package com.journaldev.design.adapter;

public class Socket {

	public Volt getVolt(){
```

```
                return new Volt(120);
        }
}
```

Now we want to build an adapter that can produce 3 volts, 12 volts and default 120 volts. So first of all we will create an adapter interface with these methods.

```
package com.journaldev.design.adapter;

public interface SocketAdapter {

        public Volt get120Volt();

        public Volt get12Volt();

        public Volt get3Volt();
}
```

## Two Way Adapter Pattern

While implementing Adapter pattern, there are two approaches – class adapter and object adapter – however both these approaches produce same result.

1. **Class Adapter** – This form uses **java inheritance** and extends the source interface, in our case Socket class.
2. **Object Adapter** – This form uses **Java Composition** and adapter contains the source object.

## Adapter Design Pattern – Class Adapter

Here is the **class adapter** approach implementation of our adapter.

```
package com.journaldev.design.adapter;

//Using inheritance for adapter pattern
public class SocketClassAdapterImpl extends Socket
implements SocketAdapter{

        @Override
        public Volt get120Volt() {
                return getVolt();
        }
```

```
        @Override
        public Volt get12Volt() {
                Volt v= getVolt();
                return convertVolt(v,10);
        }

        @Override
        public Volt get3Volt() {
                Volt v= getVolt();
                return convertVolt(v,40);
        }
```

## Adapter Design Pattern – Object Adapter Implementation

Here is the **Object adapter** implementation of our adapter.

```
package com.journaldev.design.adapter;

public class SocketObjectAdapterImpl implements
SocketAdapter{

        //Using Composition for adapter pattern
        private Socket sock = new Socket();
```

```
	@Override
	public Volt get120Volt() {
		return sock.getVolt();
	}

	@Override
	public Volt get12Volt() {
		Volt v= sock.getVolt();
		return convertVolt(v,10);
	}

	@Override
	public Volt get3Volt() {
		Volt v= sock.getVolt();
```

Notice that both the adapter implementations are almost same and they implement the `SocketAdapter` interface. The adapter interface can also be an **abstract class**.

Here is a test program to consume our adapter design pattern implementation.

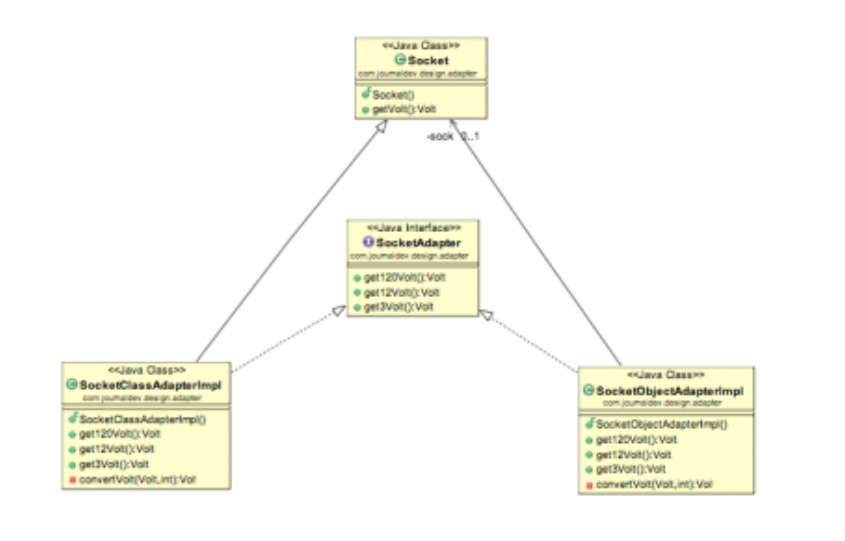
```
package com.journaldev.design.test;

import com.journaldev.design.adapter.SocketAdapter;
import
com.journaldev.design.adapter.SocketClassAdapterImpl;
import
com.journaldev.design.adapter.SocketObjectAdapterImpl;
import com.journaldev.design.adapter.Volt;
```

```
public class AdapterPatternTest {

	public static void main(String[] args) {

		testClassAdapter();
		testObjectAdapter();
	}

	private static void testObjectAdapter() {
		SocketAdapter sockAdapter = new
SocketObjectAdapterImpl();
		Volt v3 = getVolt(sockAdapter,3);
		Volt v12 = getVolt(sockAdapter,12);
```

When we run above test program, we get following output.

```
v3 volts using Class Adapter=3
v12 volts using Class Adapter=12
v120 volts using Class Adapter=120
v3 volts using Object Adapter=3
v12 volts using Object Adapter=12
v120 volts using Object Adapter=120
```

## Adapter Design Pattern Class Diagram

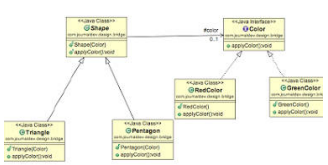



## Adapter Design Pattern Example in JDK

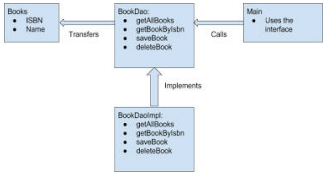
Some of the adapter design pattern example I could easily find in JDK classes are;

- java.util.Arrays#asList()
- java.io.InputStreamReader(InputStream) (returns a Reader)
- java.io.OutputStreamWriter(OutputStream) (returns a Writer)

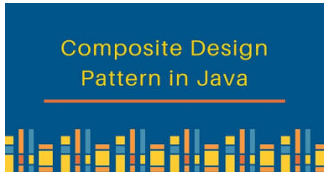
That’s all for adapter design pattern in java.



**Bridge Design Pattern in Java**

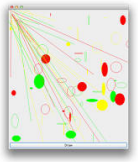


**DAO Design Pattern**

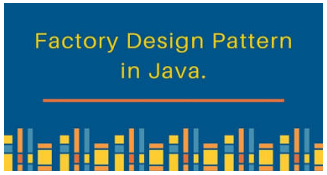



**Composite Design Pattern in Java**

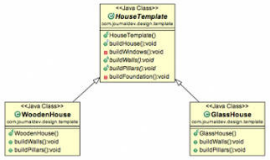

**Facade Design Pattern in Java**



**Flyweight Design Pattern in Java**




**Factory Design Pattern in Java**



**Template Method Design Pattern in Java**

**Java Injec Patte**

« PREVIOUS
Prototype Design Pattern in Java

NEXT »
Bridge Design Pattern in Java

**About Pankaj**

If you have come this far, it means that you liked what you are reading. Why not reach little more and connect with me directly on **Google Plus**, **Facebook** or **Twitter**. I would love to hear your thoughts and opinions on my articles directly.

Recently I started creating video tutorials too, so do check out my videos on **Youtube**.

FILED UNDER: DESIGN PATTERNS

## Comments

**Silent says**

MARCH 19, 2019 AT 2:40 AM

It's helpful, it is simple and detailed.

I try to understand every design pattern as a designer and use it in our daily working or programming.

Thanks.

Reply

**jitesh says**

MARCH 18, 2019 AT 2:56 AM

getVolt(sockAdapter,3); –> which method is actually calling ?

Reply

**Pitos says**

JULY 31, 2018 AT 4:46 AM

Hi,

Personally, I would inject Socket instead of letting the adapter to initialize it, e.g

public SocketObjectAdapterImpl(Socket s){

this.s = s;

}

The reason for this change is that if you want to add setters to the Socket, then you won't be able to set the state from the client as the reference to the object is within the state of the Adapter. I would limit the adapter's responsibility to getting state and/or converting, nothing more.

Any reason why you favour the current implementation over the DI ?

Reply

**Yogesh says**

SEPTEMBER 12, 2017 AT 7:31 PM

Hi Pankaj,

Your example was really good, it did help me understand the adapter pattern. I referenced your post while creating my demo for the same.

Reply

**Debi Boxi says**
JULY 26, 2017 AT 12:01 AM
Hi,
I love this tutorial. its really help me a lots to understanding.
Thanks a lot
Reply

**ravi says**
JUNE 18, 2017 AT 10:22 PM
What you have explained here, to some extent looks like strategy pattern.
adapter pattern must be like below
DestinationInterface adapt(SourceInterface si){
DestinationInterface di = new DestinationInterface ();
//logic to convert / map SourceInterface to DestinationInterface
return di;
}
Reply

**Jose M Quijada says**
JUNE 7, 2017 AT 11:18 AM
Keep in mind that with class adapter, you're extending the adaptee (Socket.java in this case), which might pollute your API with

undesirable methods/members of Socket.java and all of its parents, because you're inheriting everything from Socket.java, giving the client more code than it needs/wants. This is important to keep in mind if you're adapting a class that you don't own from a 3rd party or something like that. Also, because the adapter is statically linked at compile time, you cannot adapt child classes of Socket.java. Object adapter is more flexible because you can adapt not only Socket.java, but all of its child classes as well.

Reply

**Raed says**

OCTOBER 11, 2016 AT 2:37 AM

Hello Pankaj,

could you please show us how does Object Adapter work ? i mean as Code in Java ?

best regards

Raed

Reply

**Raed says**

JULY 26, 2017 AT 1:00 AM

Hello Pankaj,

sorry i did not see it (SocketObjectAdapterImpl) with Composition.

best regards

Raed

Reply

**JIten says**
SEPTEMBER 6, 2016 AT 2:59 AM
Good one but required more complex example.
Reply

**Mahadev Shinde says**
AUGUST 27, 2015 AT 11:41 PM
Where are your two unrelated interfaces in these examples ?
Reply

**Zeki says**
NOVEMBER 30, 2016 AT 2:55 PM
I was thinking the same.
Reply

**DEBARATI MAJUMDER says**
AUGUST 13, 2017 AT 7:57 AM
Same question… where are the unrelated interfaces?
Reply

**Pankaj says**
AUGUST 13, 2017 AT 8:05 AM

I have not created two “Socket” classes but you can imagine two Socket classes as “Socket120V” and “Socket3V”. Now you have two unrelated interfaces and we are applying Adapter pattern to create a single class to produce all different types of Volt outputs.
Reply

**Ana says**
DECEMBER 9, 2018 AT 2:55 PM
Kindly Update Your Page With These Unrelated Interfaces.So We Wont require to imagine.It Willl Be Easy To Understand.

**Giorgio says**
NOVEMBER 13, 2014 AT 2:34 AM
Hy Pankaj Kumar,
thanks for your post, I have a question: reading Head first Design pattern, it seem in the Adapter design pattern get involved a bit od delegation, since all the requests made from the client using target interface, will be done delegating the job to the adaptee object. Actually I can’t see this in the Arrays.asList, because after all you’re creating a

new object from using this method and not adapting array to respond to List interface using an adapter.
It seems for me more reasonable and Adapter compliant in this manner:

```
//ListAdapter.java
public class ListAdapter implements List{
private String[] array;
public ListAdapter(String[] array) {
this.array=array;
}
@Override
public int size() {
return array.length;
}
/*Other methods*/
}
// Test.java
public static void main(String[] args) {
String[] array = new String[2];
array[0]=”Giorgio”;
array[1]=”Sara”;
/* Versione artigianale del adapter tra array e list */
List list = new ListAdapter(array);
System.out.println(list.size());
}
```

Reply

**MANOJ @ CITE says**

MAY 16, 2014 AT 2:41 AM

Nice One. Pretty clear cut explanation. keep it on.

Thanks,

Manoj@ CITE

Reply

**HIMANSU NAYAK says**

APRIL 14, 2014 AT 7:45 AM

Hi Pankaj,

Very simple and clear example. Please provide some knowledge on test framework also like junit or testng.

Reply

**Comment Policy:**Please submit comments to add value to the post. Comments like "Thank You" and "Awesome Post" will be not published. If you want to post code then wrap them inside <pre> tags. For example **<pre>class Foo { }</pre>**.

If you want to post XML content, then please escape < with < and > with > otherwise they will not be shown properly.

## Leave a Reply

Your email address will not be published. Required fields are marked *

Comment

Name *

Email *

POST COMMENT