HowToDoInJava

Adapter Design Pattern in Java

Last Updated: August 24, 2021 By: Lokesh Gupta Structural Patterns Design Patterns

Ever tried to use a your *camera memory card* in your laptop. You cannot use it directly simply because there is no port in laptop which accept it. You must use a compatible card reader. You put your memory card into the card reader and then inject the card reader into the laptop. This card reader can be called the adapter.

A similar example is your *mobile charger* or your *laptop charger* which can be used with any power supply without fear of the variance power supply in different locations. That is also called power "adapter".

In programming as well, adapter pattern is used for similar purposes. It enables two incompatible interfaces to work smoothly with each other. Going by definition:

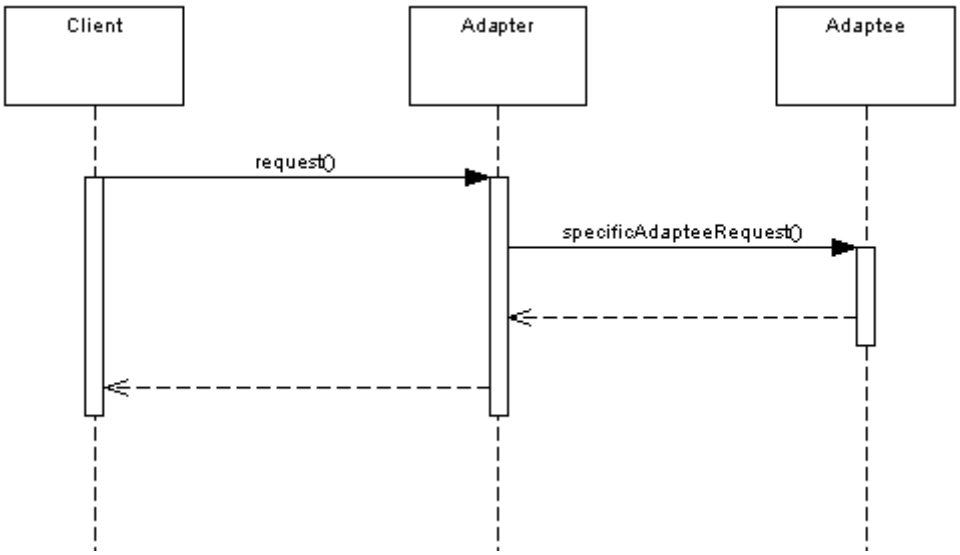
> Adapter design pattern is one of the **structural design pattern** and its used so that two unrelated interfaces can work together. The object, that joins these unrelated interfaces, is called an Adapter.

The definition of Adapter provided in the original **Gang of Four** book on Design Patterns states:

> "Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces."

An adapter pattern is also known as **Wrapper pattern** as well. Adapter Design is very useful for the system integration when some other existing components have to be adopted by the existing system without sourcecode modifications.

A typical interaction happen like this:



Where to use Adapter Design Pattern?

The main use of this pattern is when a class that you need to use doesn't meet the requirements of an interface. e.g. If you want to read the system input through command prompt in java then given below code is common way to do it:

```
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
System.out.print("Enter String");
String s = br.readLine();
System.out.print("Enter input: " + s);
```

Now observe the above code carefully.

1) `System.in` is `static` instance of `InputStream` declared as:

```
public final static InputStream in = null;
```

This input stream natively reads the data from the console in bytes stream.

2) BufferedReader as java docs define, reads a character stream.

```
//Reads text from a character-input stream, buffering characters so as to
//provide for the efficient reading of characters, arrays, and lines.

public class BufferedReader extends Reader{..}
```

Now here is the problem. `System.in` provides byte stream where `BufferedReader` expects character stream. How they will work together?

This is the **ideal situation to put a adapter** in between two incompatible interfaces. `InputStreamReader` **does exactly this thing and works adapter between** `System.in` **and** `BufferedReader`.

```
/** An InputStreamReader is a bridge from byte streams to character streams:
  * It reads bytes and decodes them into characters using a specified charset.
  * The charset that it uses may be specified by name or may be given explicitly,
  * or the platform's default charset may be accepted.
  */

public class InputStreamReader extends Reader {...}
```

I hope the above usecase makes sense to all of you. Now, the next question is how much work adapter should do to make two incompatible interfaces work together?

How much work the Adapter Pattern should do?

Answer is really simple, it should do only that much work so that both incompatible interfaces can adapt each other and that's it. e.g. in our above case study, A `InputStreamReader` simply wraps the `InputStream` and nothing else. Then `BufferedReader` is capable of using underlying `Reader` to read the characters in stream.

```
/**
 * Creates an InputStreamReader that uses the default charset.
```

```
   * @param  in   An InputStream
   */
  public InputStreamReader(InputStream in) {
    super(in);
    try {
      sd = StreamDecoder.forInputStreamReader(in, this, (String)null); // ## check
    } catch (UnsupportedEncodingException e) {
      // The default encoding should always be available
      throw new Error(e);
    }
  }
```

> Also note that if the Target and Adaptee are similar then the adapter has just to delegate the requests from the Target to the Adaptee. If Target and Adaptee are not similar, then the adapter might have to convert the data structures between those and to implement the operations required by the Target but not implemented by the Adaptee.

Now when we have a good understanding of what's an adapter looks like, let's identify the actors used into adapter design pattern:

Participants of Adapter Design Pattern

The classes and/or objects participating in this pattern are listed as below:

- **Target** (BufferedReader): It defines the application-specific interface that Client uses directly.
- **Adapter** (InputStreamReader): It adapts the interface Adaptee to the Target interface. It's middle man.
- **Adaptee** (System.in): It defines an existing incompatible interface that needs adapting before using in application.
- **Client**: It is your application that works with Target interface.

Other example implementations of Adapter Design Pattern

Some other examples worth noticing is as below:

1) java.util.Arrays#asList()

This method accepts multiple strings and return a list of input strings. Though it's very basic usage, but it's what an adapter does, right?

2) java.io.OutputStreamWriter(OutputStream)

It's similar to above usecase we discussed in this post:

```
Writer writer = new OutputStreamWriter(new FileOutputStream("c:\\data\\output.txt
writer.write("Hello World");
```

3) javax.xml.bind.annotation.adapters.XmlAdapter#marshal() and #unmarshal()

Adapts a Java type for custom marshaling. Convert a bound type to a value type.

That's all for this simple and easy topic.

Happy Learning !!

Was this post helpful?

Let us know if you liked the post. That's the only way we can improve.

Yes

No

Recommended Reading:

1. Decorator Design Pattern in Java
2. Bridge Design Pattern
3. Composite Design Pattern
4. Facade Design Pattern
5. Flyweight Design Pattern
6. Proxy Design Pattern
7. Prototype design pattern in Java
8. Visitor Design Pattern Example
9. Iterator Design Pattern
0. Memento Design Pattern

Join 7000+ Awesome Developers

Get the latest updates from industry, awesome resources, blog updates and much more.

Email Address

Subscribe

* We do not spam !!

10 thoughts on “Adapter Design Pattern in Java”

Arun Kumar

June 13, 2022 at 2:16 pm

Adapter class is a good example of object composition. Adapter class “has a” instance of the adaptee class.

Reply

Vishal

March 15, 2020 at 10:09 pm

Very good explanation Lokesh, thank you.

Reply

po

January 8, 2018 at 9:16 pm

Diff between facade and factory pattern?

Reply

Pradip

March 18, 2017 at 10:23 am

Can we say Hibernate is a real time example of Adaptor design pattern? It converts RDBMS specific query.

Reply

confused

December 24, 2014 at 1:46 pm

Thanks Lokesh. This was of great help!!!!

Reply

Bala

May 28, 2014 at 6:54 pm

Dear Lokesh,
Adapter pattern is simple and powerful. Could you please post for Facade pattern as well would be appreciated,
Thanks,
Bala

Reply

Lokesh Gupta

May 28, 2014 at 6:55 pm

I will post soon.

Reply

Vinodkumar

May 16, 2014 at 1:55 pm

Hi Lokesh ,

I liked your simple and straight explanation about concept and you wont eat time in telling big story, that's really made me crazy to read your articles as and when I get time from my Job.

Please do you have real time examples to focus on real time entities on Design pattern planned in future..awaiting for those:)

Singleton : Per JVM one object Example: Helper, Utility classes and Service classes using spring
Strategy Pattern : WE wanted to display price and based on rule defined by the Author , through UI he switch the price pattern then Spring I used to inject respective.
Factory : Used this one scenario for payment gateway Integration, when command pattern asked me to care an object , then Factory will give back the Objects
Builder pattern : DB and Front end data compatibility issue, DB stored in Interger and Fromt end need Integer or Front end want with formatting price bla bla …

Thanks for your Adptor design pattern.

Best Regards
Vinod

Reply

Lokesh Gupta

May 19, 2014 at 6:54 am

Thanks for the above suggestions. They are really quit interesting. I have added them in my TODO list.

Reply

mahendra

October 19, 2014 at 6:54 pm

Thanks Lokesh!

Reply

Leave a Comment

Name *

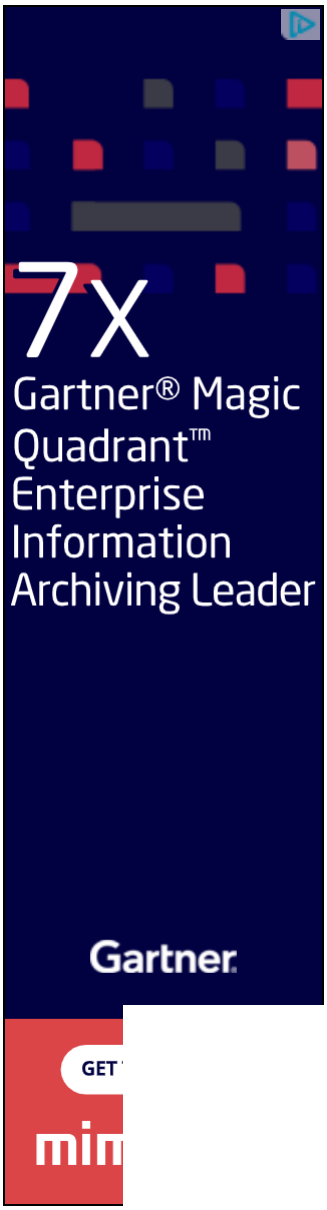
Email *

Website

Add me to your newsletter and keep me updated whenever you publish new blog posts

Post Comment

Search ...


7x
Gartner® Magic Quadrant™ Enterprise Information Archiving Leader
Gartner
GET
min


7x
Gartner® Magic Quadrant™
Enterprise Information
Archiving Leader
Gartner
mimecast
GET THE REPORT

HowToDoInJava

A blog about Java and related technologies, the best practices, algorithms, and interview questions.

Meta Links

- About Me
- Contact Us
- Privacy policy
- Advertise
- Guest Posts

Blogs

REST API Tutorial

Copyright © 2022 · Hosted on Cloudways · Sitemap