Adapter Design Pattern

Adapter Design Pattern is also known as Wrapper Design Pattern in java. this desingn pattern is used to work as bridge between two incompatible iterfaces.

As per the list of Gang of Four patterns, Adapter is a structural pattern  
As the name suggest, Adapter allows two classes of a different interface to work together, without changing any code on either side.

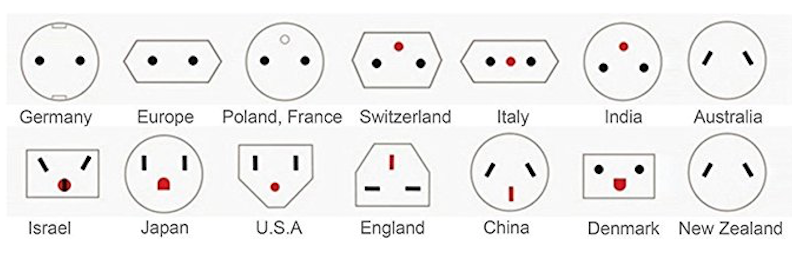
The real time example for adapters are electrical socket adapters. When people from India go to abroad use to face incompatible sockets type in US, UK or any Europian or North American country. Because they have their device’s sockets designed as per Indian standard where

USA has a rectangular socket, as compared to cylindrical one from India

we

USA rectangular socket

Indian cylindrical socket



Basically, when we talk Electrical adapter, we talked about adapter which changes the voltage or the one's which allows using a plug designed for one country in a Socket of another country.

We need to have an adapter to use US or any other country’s power socket to plugin our device.

This example shows an important attribute of Adapter design pattern in software development, you can neither change the Socket of visiting country, neither can change the plug of your  laptop, So you introduce an *Adapter*, which makes things working without changing any party. Similarly, **Adapter design pattern makes incompatible interfaces work together, without changing them**. Only new code which is inserted is in the form of Adapter or Wrapper class.

JdbcSupport

<interface>

OracleDb

Client

+ insertOracle

+ updateOracle

+ deleteOracle

+ add

+ update

+ remove

* JdbcSupprt

🡪Target

* DbAdapter

🡪Adapter

* OracleDb,

PostgresDb,

MysqlDb

🡪Adaptee

+ insertPostgres

+ updatePostgres

+ deletePostgres

PstgresDb

DbAdapter

+ add

+ update

+ remove

+ insertMysql

+ updateMysql

+ deleteMysql

MysqlDb

**Adapter pattern in Java - Map Adapter**

Let's see one more example of Adapter pattern in Java. If you remember the java.util.Map has no way to automatically load a [two-dimensional array](http://javarevisited.blogspot.com/2016/02/6-example-to-declare-two-dimensional-array-in-java.html) of objects into a Map as key-value pairs. We can create an adapter class that does this.  
  
Sometimes the problem you are solving is as simple as *"I don't have the interface I want"*. Two of the patterns in the list of [GOF design patterns](http://www.amazon.com/Design-Patterns-Object-Oriented-Professional-Computing/dp/0201634988?tag=javamysqlanta-20), Adapter, and Facade pattern solve this problem by providing an *alternate* interface. Adapter pattern takes whatever interface you have and produce the interface you need. While Facade pattern provides a simpler interface to deal with a number of classes or bundle of resources.  
  
Here is my implementation of Adapter pattern in Java to convert a two-dimensional array of objects into Map of key-value pairs  
  
  
**Java Program to implement Adapter design Pattern**  
  
**import** java.util.AbstractMap;

**import** java.util.HashMap;

**import** java.util.Map;

**import** java.util.Set;

/\*\*

Implementation of Adapter pattern in Java. The java.util.Map has no way to automatically load a two dimensional array of objects into a Map as key-value pairs. This Java program creates an adapter class that does this.

\* \* \* **@author** WINDOWS 8

\*/

**public** **class** Test {

**public** **static** **void** main(String args[]) {

Integer[][] squares = { { 2, 4 }, { 3, 9 }, { 4, 16 } };

MapAdapter adapter = **new** MapAdapter(squares);

System.***out***.println("adapter map contains : " + adapter);

}

}

/\*

This class is an adapter which allows to create a Map by providing a two dimensional array of keys and values. It extends AbstractMap class so that it become a Map and can be passed around where a Map is needed. All other method is implemented in AbstractMap except the adapter functionality which is implemented in constructor of this class.

\*/ **class** MapAdapter **extends** AbstractMap {

**private** Map map;

**public** MapAdapter(Object[][] array) {

**super**();

map = **new** HashMap();

**for** (Object[] mapping : array) {

map.put(mapping[0], mapping[1]);

}

}

@Override

**public** Set entrySet() {

**return** map.entrySet();

}}

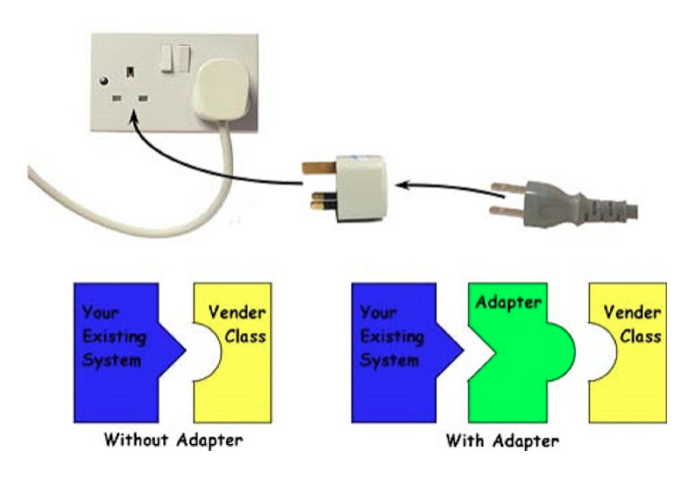
Output adapter

map contains:

{2=4, 3=9, 4=16}

You can see that our MapAdapter extends AbstractMap to create a Map which can take a [two-dimensional array](http://java67.blogspot.com/2014/10/how-to-create-and-initialize-two-dimensional-array-java-example.html) and create a [HashMap](http://www.java67.com/2013/02/10-examples-of-hashmap-in-java-programming-tutorial.html" \t "_blank) from that.

Here is one more interesting diagram which will help you to understand the intent and purpose of Adapter pattern in Java. You can see that your existing system and vendor class doesn't fit initially but when you add an Adapter the puzzle is solved and both systems is able to plugged together.



Note:

A lot of programmers confuse between Adapter and Decorator design pattern in Java, to some extent they are similar, especially object based adapter pattern but there is a subtle difference between Decorator and Adapter pattern in Java. Adapter just converts one interface to another, without adding additional functionalities, while Decorator adds new functionality into an interface