The Adapter Pattern in Java

baeldung.com/java-adapter-pattern

By
baeldung

February 24, 2019

1. Overview

In this quick tutorial, we'll have a look at the Adapter pattern and its Java implementation.

2. Adapter Pattern

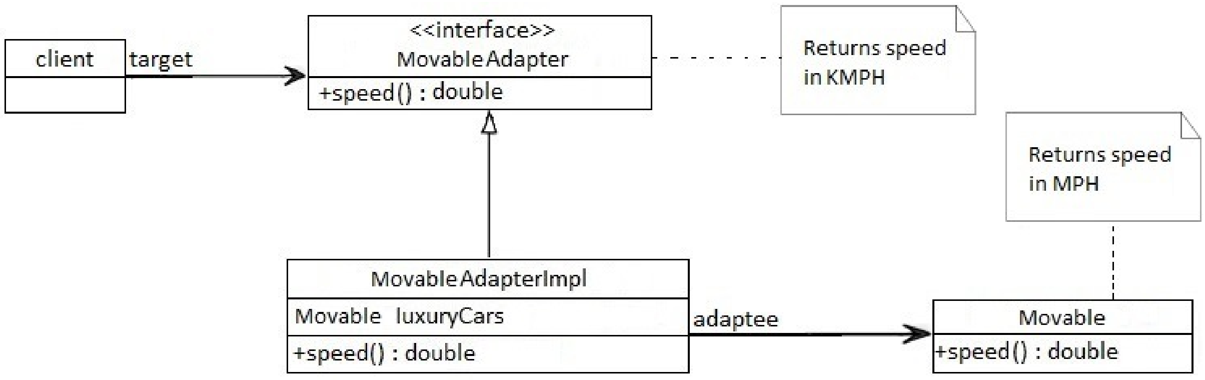
An Adapter pattern acts as a connector between two incompatible interfaces that otherwise cannot be connected directly. An Adapter wraps an existing class with a new interface so that it becomes compatible with the client's interface.

The main motive behind using this pattern is to convert an existing interface into another interface that the client expects. It's usually implemented once the application is designed.

2.1. Adapter Pattern Example

Consider a scenario in which there is an app that's developed in the US which returns the top speed of luxury cars in miles per hour (MPH). Now we need to use the same app for our client in the UK that wants the same results but in kilometers per hour (km/h).

To deal with this problem, we'll create an adapter which will convert the values and give us the desired results:



First, we'll create the original interface *Movable* which is supposed to return the speed of some luxury cars in miles per hour:

```
public interface Movable {
    // returns speed in MPH
    double getSpeed();
}
```

We'll now create one concrete implementation of this interface:

```
public class BugattiVeyron implements Movable {

    @Override
    public double getSpeed() {
        return 268;
    }
}
```

Now we'll create an adapter interface *MovableAdapter* that will be based on the same *Movable* class. It may be slightly modified to yield different results in different scenarios:

```
public interface MovableAdapter {
    // returns speed in KM/H
    double getSpeed();
}
```

The implementation of this interface will consist of private method *convertMPHtoKMPH()* that will be used for the conversion:

```
public class MovableAdapterImpl implements MovableAdapter {
    private Movable luxuryCars;

    // standard constructors

    @Override
    public double getSpeed() {
        return convertMPHtoKMPH(luxuryCars.getSpeed());
    }

    private double convertMPHtoKMPH(double mph) {
        return mph * 1.60934;
    }
}
```

Now we'll only use the methods defined in our Adapter, and we'll get the converted speeds. In this case, the following assertion will be true:

```
@Test
public void whenConvertingMPHToKMPH_thenSuccessfullyConverted() {
    Movable bugattiVeyron = new BugattiVeyron();
    MovableAdapter bugattiVeyronAdapter = new MovableAdapterImpl(bugattiVeyron);

    assertEquals(bugattiVeyronAdapter.getSpeed(), 431.30312, 0.00001);
}
```

As we can notice here, our adapter converts *268 mph* to *431 km/h* for this particular case.

2.2. When to Use Adapter Pattern

- **When an outside component provides captivating functionality that we'd like to reuse, but it's incompatible with our current application**. A suitable Adapter can be developed to make them compatible with each other
- When our application is not compatible with the interface that our client is expecting
- When we want to reuse legacy code in our application without making any modification in the original code

3. Conclusion

In this article, we had a look at the Adapter design pattern in Java.

The full source code for this example is available over on GitHub.