THE ADAPTER PATTERN

Chandan R. Rupakheti

Week 5-1

Today

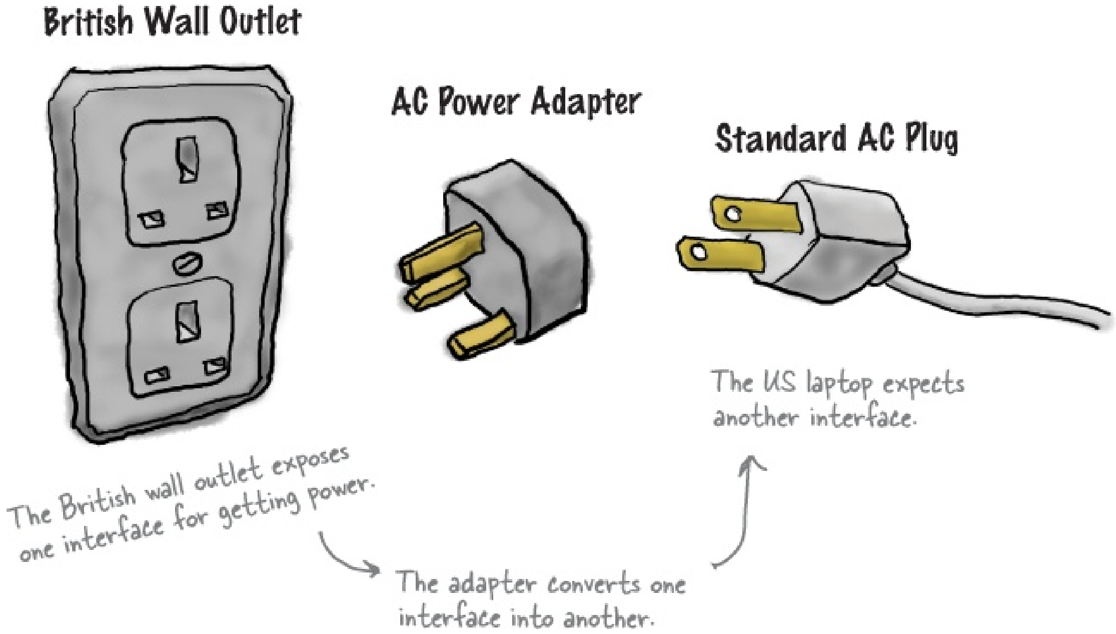
Do you think the readers are really getting the impression we're watching a horse race rather than sitting in a photo studio?
That's the beauty of our profession: we can make things look like something they're not!
You mean it's not supposed to be a football match?
Wrapped in this coat, I'm a different man!

Today, we're going to attempt such impossible feats as putting a square peg in a round hole!

But first, do you all know what features you are implementing for Sprint 7?

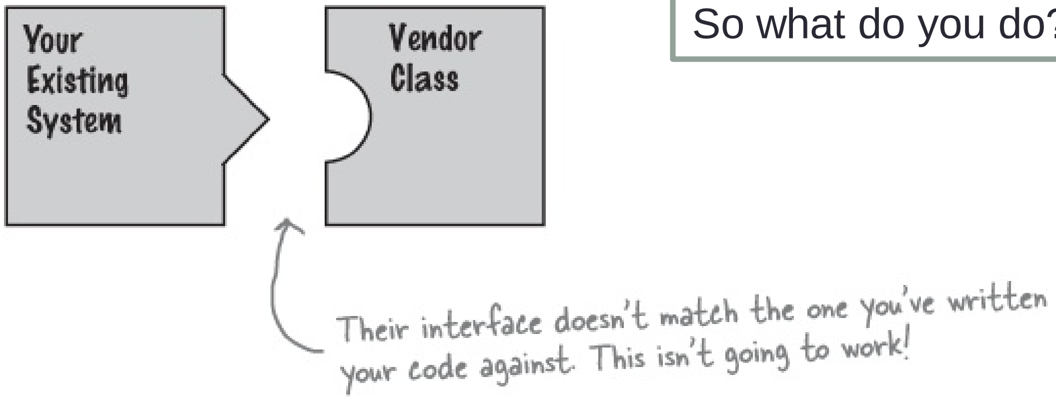
Adapters all around us

Have you ever needed to use a US-made laptop in Great Britain?

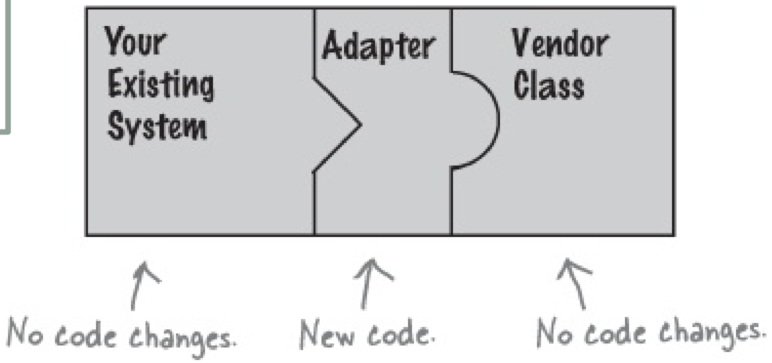

British Wall Outlet
AC Power Adapter
Standard AC Plug
The US laptop expects another interface.
The British wall outlet exposes one interface for getting power.
The adapter converts one interface into another.

Object-oriented adapters

You don't want to solve the problem by changing your existing code and you can't change the vendor's code. So what do you do?

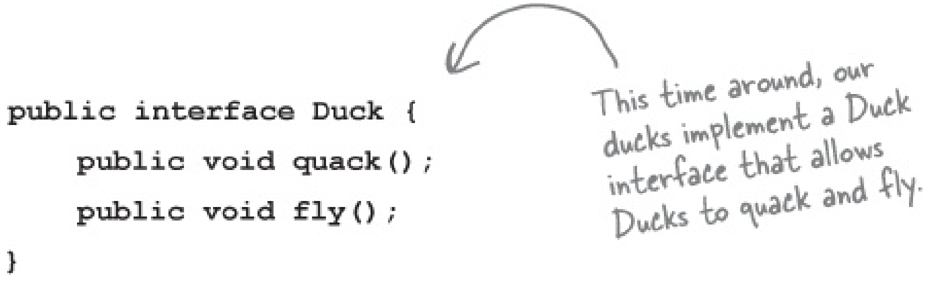

Your Existing System
Vendor Class
Their interface doesn't match the one you've written your code against. This isn't going to work!

The adapter acts as the middleman by receiving requests from the client and converting them into requests that make sense on the vendor classes.



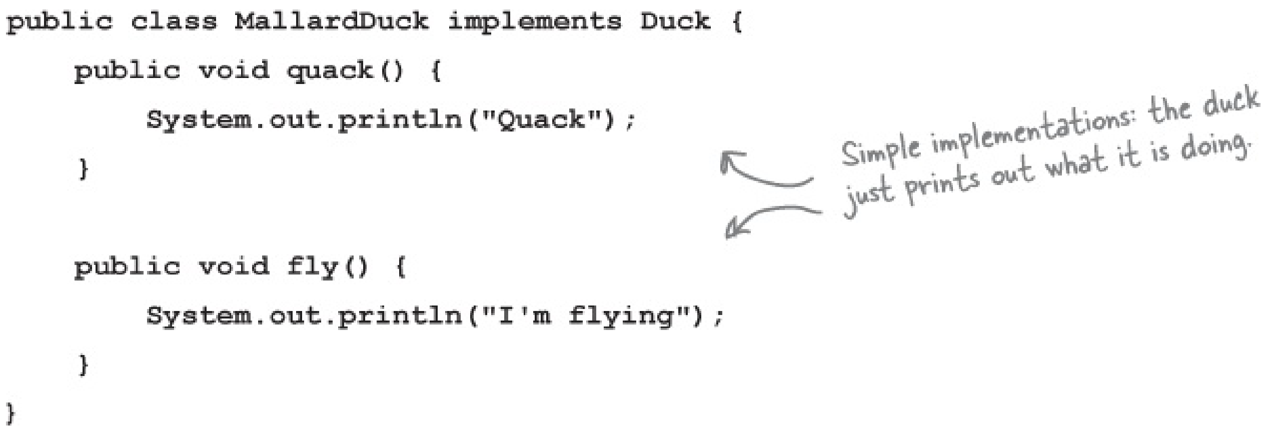
Let’s revisit the Duck example …

If it walks like a duck and quacks like a duck, then it ~~must~~ might be a ~~duck~~ turkey wrapped with a duck adapter...



```
public interface Duck {
    public void quack();
    public void fly();
}
```

This time around, our ducks implement a Duck interface that allows Ducks to quack and fly.

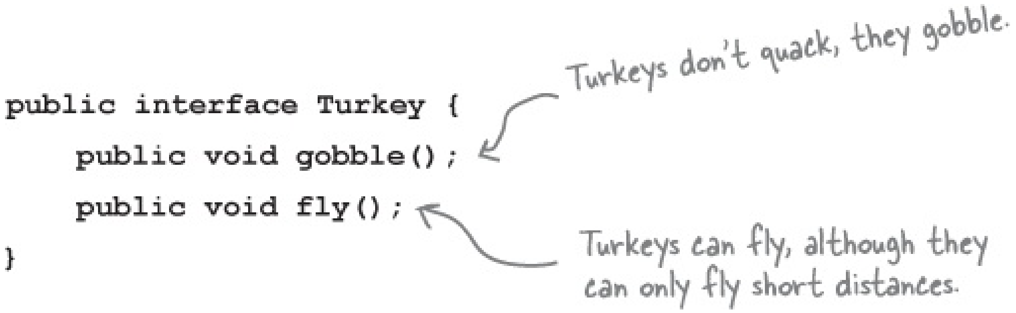


```
public class MallardDuck implements Duck {
    public void quack() {
        System.out.println("Quack");
    }

    public void fly() {
        System.out.println("I'm flying");
    }
}
```

Simple implementations: the duck just prints out what it is doing.

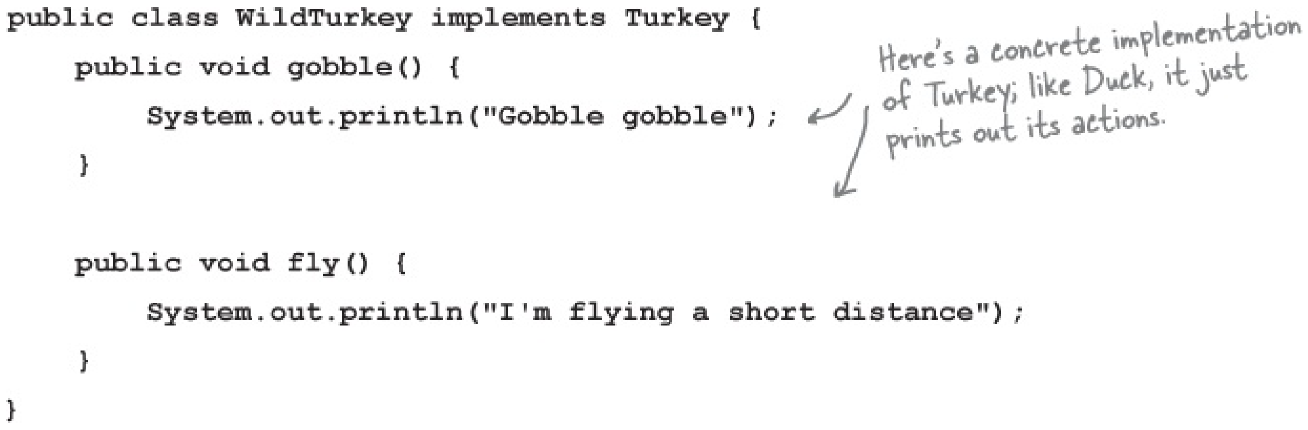
The newest fowl on the block …



```
public interface Turkey {
    public void gobble();
    public void fly();
}
```

Turkeys don't quack, they gobble.

Turkeys can fly, although they can only fly short distances.



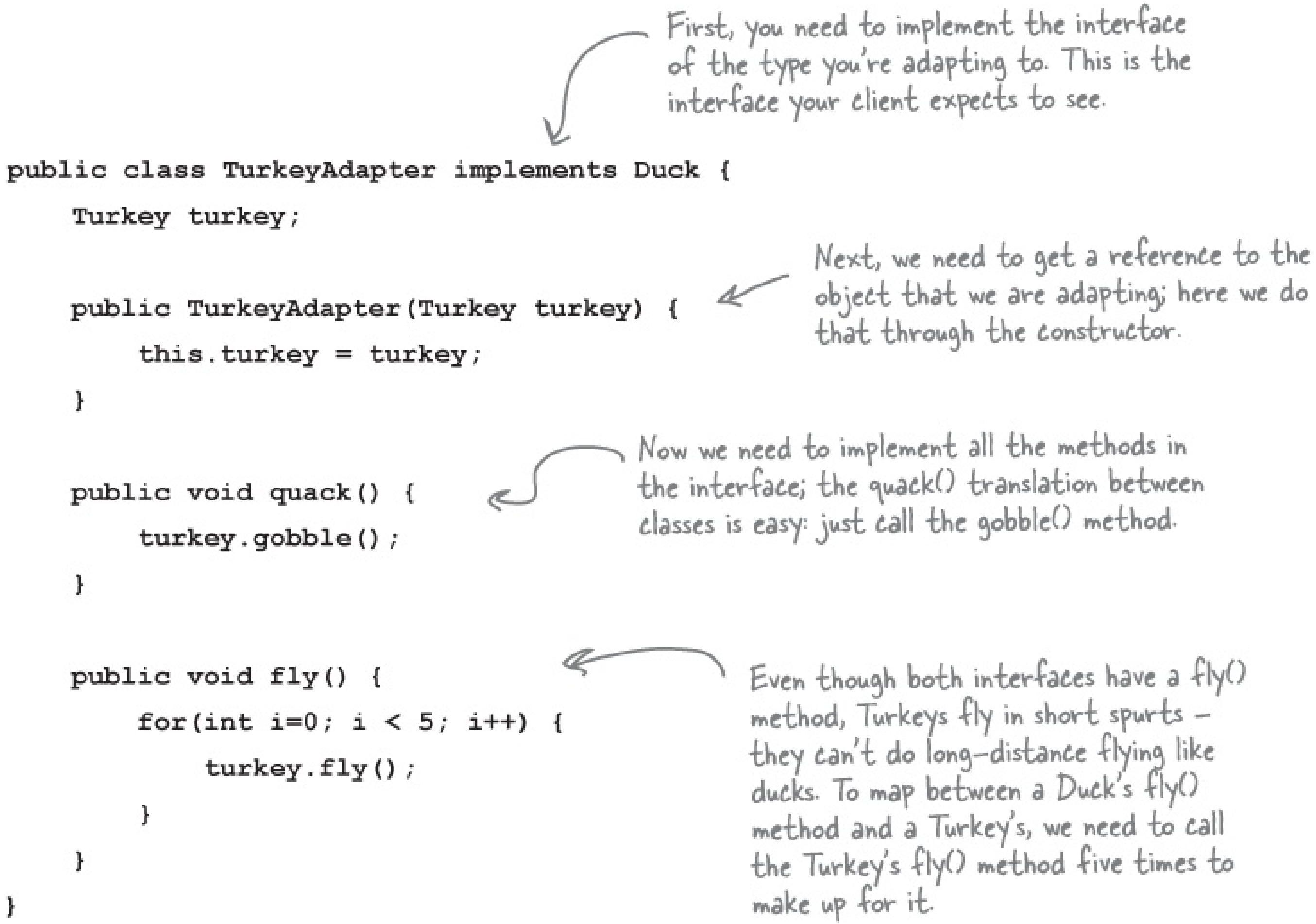
```
public class WildTurkey implements Turkey {
    public void gobble() {
        System.out.println("Gobble gobble");
    }

    public void fly() {
        System.out.println("I'm flying a short distance");
    }
}
```

Here's a concrete implementation of Turkey; like Duck, it just prints out its actions.

Now, let's say you're short on Duck objects and you'd like to use some Turkey objects in their place

Use an Adapter, easy!



First, you need to implement the interface of the type you're adapting to. This is the interface your client expects to see.

```
public class TurkeyAdapter implements Duck {
    Turkey turkey;

    public TurkeyAdapter(Turkey turkey) {
        this.turkey = turkey;
    }

    public void quack() {
        turkey.gobble();
    }

    public void fly() {
        for(int i=0; i < 5; i++) {
            turkey.fly();
        }
    }
}
```

Next, we need to get a reference to the object that we are adapting; here we do that through the constructor.

Now we need to implement all the methods in the interface; the quack() translation between classes is easy: just call the gobble() method.

Even though both interfaces have a fly() method, Turkeys fly in short spurts – they can't do long-distance flying like ducks. To map between a Duck's fly() method and a Turkey's, we need to call the Turkey's fly() method five times to make up for it.

Test drive the adapter

```
public class DuckTestDrive {
    public static void main(String[] args) {
        MallardDuck duck = new MallardDuck();

        WildTurkey turkey = new WildTurkey();
        Duck turkeyAdapter = new TurkeyAdapter(turkey);

        System.out.println("The Turkey says...");
        turkey.gobble();
        turkey.fly();

        System.out.println("\nThe Duck says...");
        testDuck(duck);

        System.out.println("\nThe TurkeyAdapter says...");
        testDuck(turkeyAdapter);
    }

    static void testDuck(Duck duck) {
        duck.quack();
        duck.fly();
    }
}
```

Let's create a Duck...

...and a Turkey.

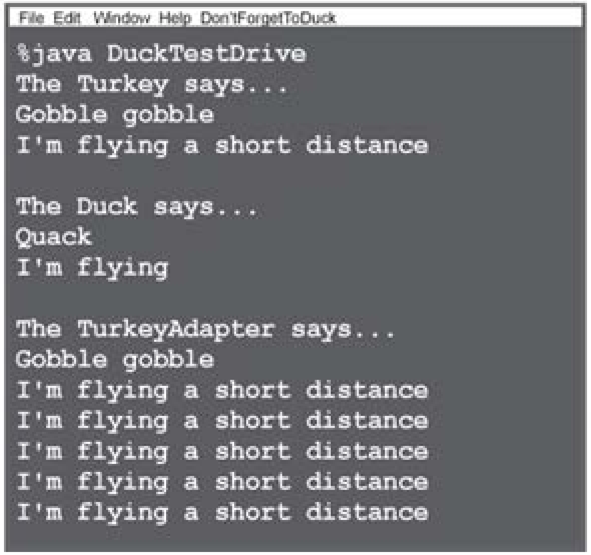
And then wrap the turkey in a TurkeyAdapter, which makes it look like a Duck.

Then, let's test the Turkey: make it gobble, make it fly.

Now let's test the duck by calling the testDuck() method, which expects a Duck object.

Now the big test: we try to pass off the turkey as a duck...

Here's our testDuck() method; it gets a duck and calls its quack() and fly() methods.



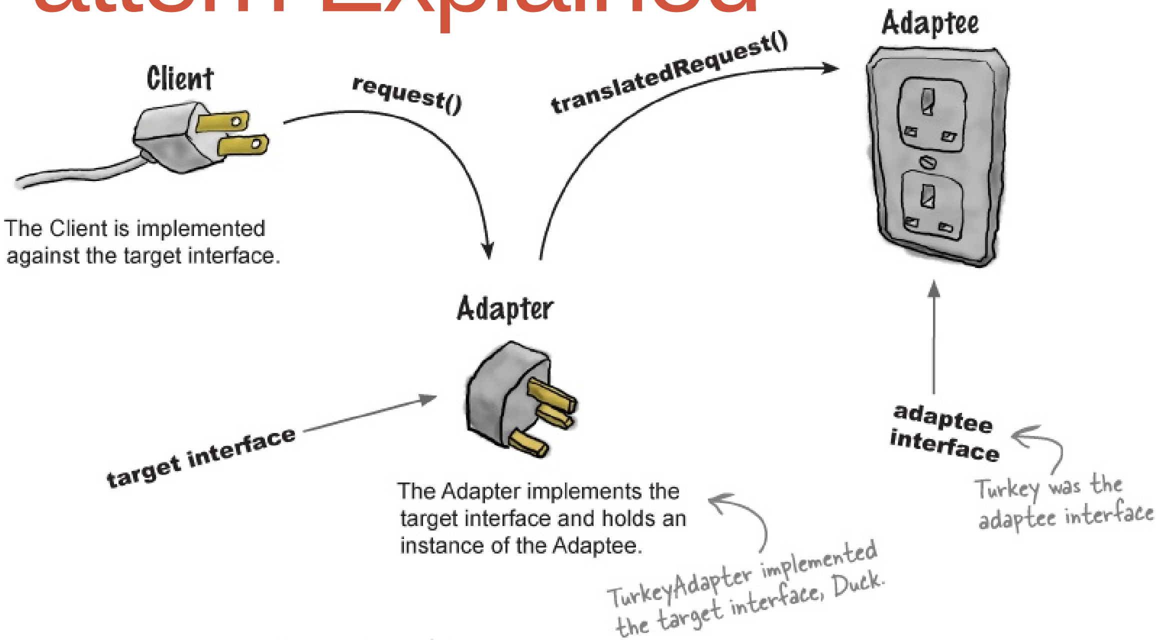
```
%java DuckTestDrive
The Turkey says...
Gobble gobble
I'm flying a short distance

The Duck says...
Quack
I'm flying

The TurkeyAdapter says...
Gobble gobble
I'm flying a short distance
I'm flying a short distance
I'm flying a short distance
I'm flying a short distance
I'm flying a short distance
```

The Adapter Pattern Explained

The Adapter Pattern converts the interface of a class into another interface the clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

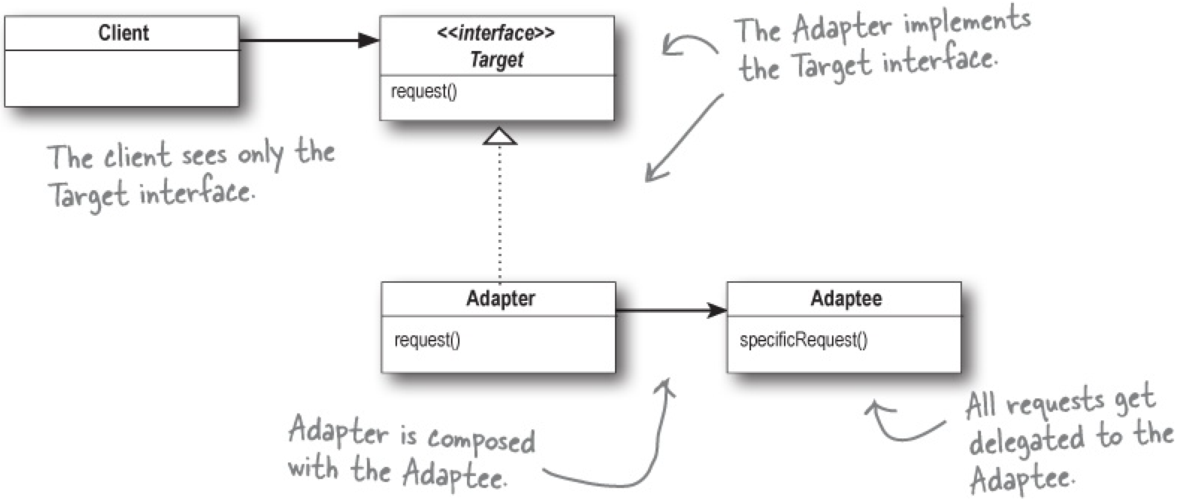


The Client is implemented against the target interface.

The Adapter implements the target interface and holds an instance of the Adaptee.

TurkeyAdapter implemented the target interface, Duck.

Turkey was the adaptee interface



The Adapter implements the Target interface.

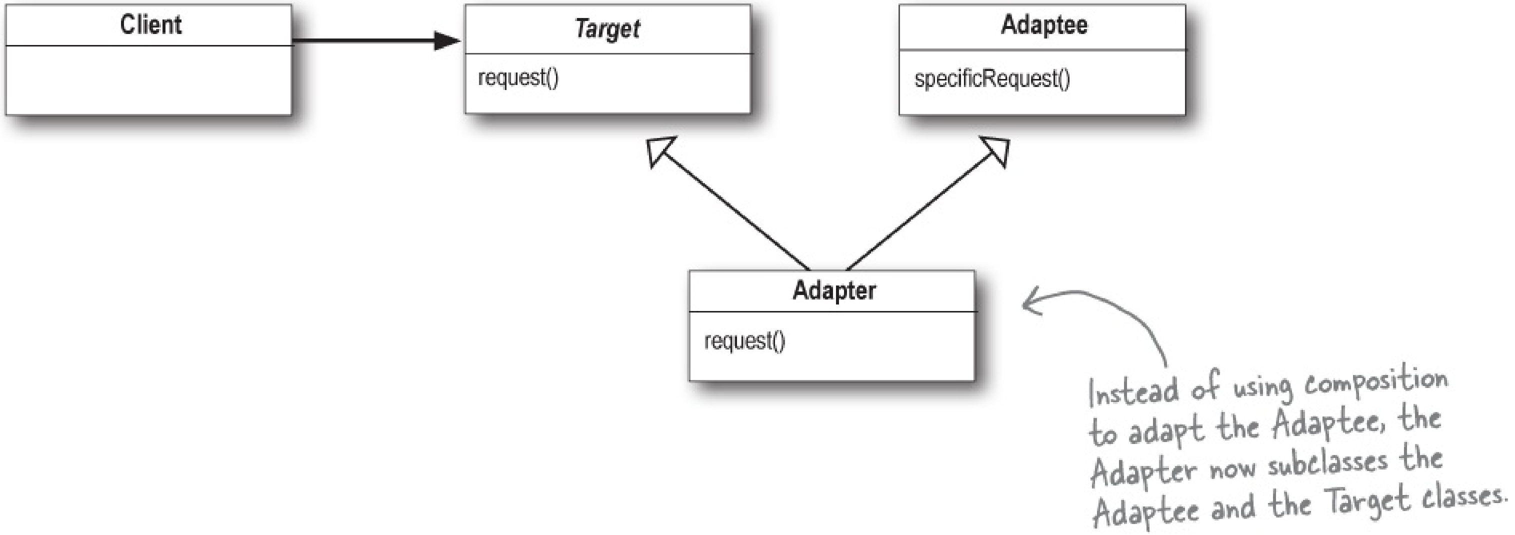
The client sees only the Target interface.

Adapter is composed with the Adaptee.

All requests get delegated to the Adaptee.

Q1

Object and class adapters



Instead of using composition to adapt the Adaptee, the Adapter now subclasses the Adaptee and the Target classes.

Class adapters use inheritance over composition. Not possible in Java but possible in languages that support multiple inheritance such as C++.

Recap

An adapter changes an interface into one a client expects.

When you need to use an existing class and its interface is not the one you need, use an adapter.

There are two forms of the Adapter Pattern: object and class adapters. Class adapters require multiple inheritance.