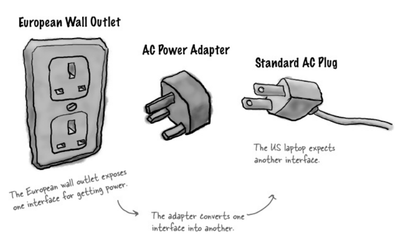
PREV
6. The Command Pattern: Encapsulating Invocation
emplate Method Patter

Chapter 7. The Adapter and Facade Patterns: Being Adaptive

In this chapter we're going to attempt such impossible feats as putting a square peg in a round hole. Sound impossible? Not when we have Design Patterns. Remember the Decorator Pattern? We **wrapped objects** to give them new responsibilities. Now we're going to wrap some objects with a different purpose: to make their interfaces look like something they're not. Why would we do that? So we can adapt a design expecting one interface to a class that implements a different interface. That's not all; while we're at it, we're going to look at another pattern that wraps objects to simplify their interface.

Adapters all around us

You'll have no trouble understanding what an OO adapter is because the real world is full of them. How's this for an example: Have you ever needed to use a US-made laptop in a European country? Then you've probably needed an AC power adapter...



You know what the adapter does: it sits in between the plug of your laptop and the European AC outlet; its job is to adapt the European outlet so that you can plug your laptop into it and receive power. Or look at it this way: the adapter changes the interface of the outlet into one that your laptop expects.

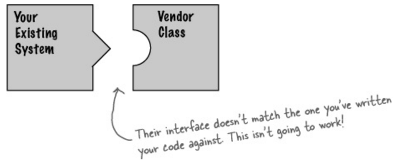
> NOTE
>
> How many other real world adapters can you think of?

Some AC adapters are simple – they only change the shape of the outlet so that it matches your plug, and they pass the AC current straight through – but other adapters are more complex internally and may need to step the power up or down to match your devices' needs.

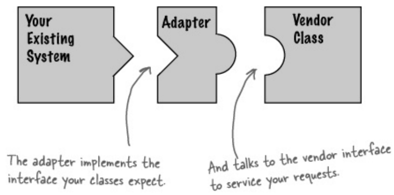
Okay, that's the real world, what about object oriented adapters? Well, our OO adapters play the same role as their real world counterparts: they take an interface and adapt it to one that a client is expecting.

Object oriented adapters

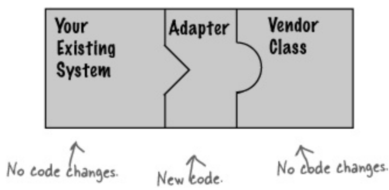
Say you've got an existing software system that you need to work a new vendor class library into, but the new vendor designed their interfaces differently than the last vendor:



Okay, 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? Well, you can write a class that adapts the new vendor interface into the one you're expecting.



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



NOTE

Can you think of a solution that doesn't require YOU to write ANY additional code to integrate the new vendor classes? How about making the vendor supply the adapter class.

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

It's time to see an adapter in action. Remember our ducks from Chapter 1? Let's review a slightly simplified version of the Duck interfaces and classes:

```
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.

Here's a subclass of Duck, the MallardDuck.

```
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.

Now it's time to meet 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. Obviously we can't use the turkeys outright because they have a different interface.

So, let's write an Adapter:

CODE UP CLOSE

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

Now we just need some code to test drive our 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.

Test run

```
File Edit Window Help Don'tForgetToDuck
%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
```

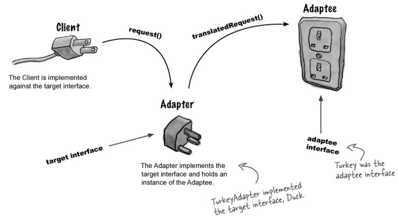
The Turkey gobbles and flies a short distance.

The Duck quacks and flies just like you'd expect.

And the adapter gobbles when quack() is called and flies a few times when fly() is called. The testDuck() method never knows it has a turkey disguised as a duck!

The Adapter Pattern explained

Now that we have an idea of what an Adapter is, let's step back and look at all the pieces again.



Here's how the Client uses the Adapter

1. **The client makes a request to the adapter by calling a method on it using the target interface.**
2. **The adapter translates the request into one or more calls on the adaptee using the adaptee interface.**

NOTE

Note that the Client and Adaptee are decoupled – neither knows about the other.

3. **The client receives the results of the call and never knows there is an adapter doing the translation.**

SHARPEN YOUR PENCIL

Let's say we also need an Adapter that converts a Duck to a Turkey. Let's call it DuckAdapter. Write that class:

How did you handle the fly method (after all we know ducks fly longer than turkeys)? Check the answers at the end of the chapter for our solution. Did you think of a better way?

THERE ARE NO DUMB QUESTIONS

Q: **Q: How much "adapting" does an adapter need to do? It seems like if I need to implement a large target interface, I could have a LOT of work on my hands.**

A: **A:** *You certainly could. The job of implementing an adapter really is proportional to the size of the interface you need to support as your target interface. Think about your options, however. You could rework all your client-side calls to the interface, which would result in a lot of investigative work and code changes. Or, you can cleanly provide one class that encapsulates all the changes in one class.*

Q: **Q: Does an adapter always wrap one and only one class?**

A: **A:** *The Adapter Pattern's role is to convert one interface into another. While most examples of the adapter pattern show an adapter wrapping one adaptee, we both know the world is often a bit more messy. So, you may well have situations where an adapter holds two or more adaptees that are needed to implement the target interface.*

This relates to another pattern called the Facade Pattern; people often confuse the two. Remind us to revisit this point when we talk about facades later in this chapter.

Q: **Q: What if I have old and new parts of my system, the old parts expect the old vendor interface, but we've already written the new parts to use the new vendor interface? It is going to get confusing using an adapter here and the unwrapped interface there. Wouldn't I be better off just writing my older code and forgetting the adapter?**

A: **A:** *Not necessarily. One thing you can do is create a Two Way Adapter that supports both interfaces. To create a Two Way Adapter, just implement both interfaces involved, so the adapter can act as an old interface or a new interface.*

Adapter Pattern defined

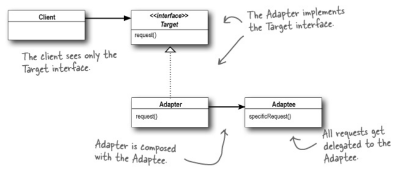
Enough ducks, turkeys and AC power adapters; let's get real and look at the official definition of the Adapter Pattern:

NOTE

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.

Now, we know this pattern allows us to use a client with an incompatible interface by creating an Adapter that does the conversion. This acts to decouple the client from the implemented interface, and if we expect the interface to change over time, the adapter encapsulates that change so that the client doesn't have to be modified each time it needs to operate against a different interface.

We've taken a look at the runtime behavior of the pattern; let's take a look at its class diagram as well:



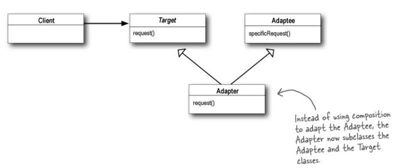
The Adapter Pattern is full of good OO design principles: check out the use of object composition to wrap the adaptee with an altered interface. This approach has the added advantage that we can use an adapter with any subclass of the adaptee.

Also check out how the pattern binds the client to an interface, not an implementation; we could use several adapters, each converting a different backend set of classes. Or, we could add new implementations after the fact, as long as they adhere to the Target interface.

Object and class adapters

Now despite having defined the pattern, we haven't told you the whole story yet. There are actually *two* kinds of adapters: *object* adapters and *class* adapters. This chapter has covered object adapters and the class diagram on the previous page is a diagram of an object adapter.

So what's a *class* adapter and why haven't we told you about it? Because you need multiple inheritance to implement it, which isn't possible in Java. But, that doesn't mean you might not encounter a need for class adapters down the road when using your favorite multiple inheritance language! Let's look at the class diagram for multiple inheritance.



Look familiar? That's right – the only difference is that with class adapter we subclass the Target and the Adaptee, while with object adapter we use composition to pass requests to an Adaptee.

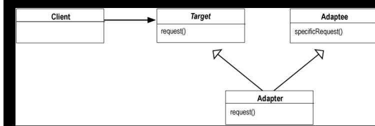
BRAIN POWER

Object adapters and class adapters use two different means of adapting the adaptee (composition versus inheritance). How do these implementation differences affect the flexibility of the adapter?

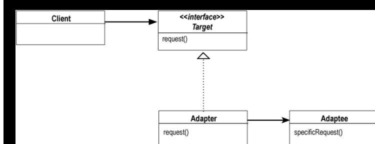
DUCK MAGNETS

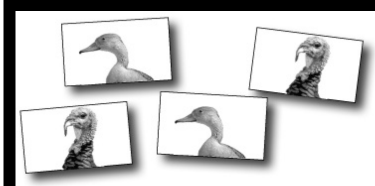
Your job is to take the duck and turkey magnets and drag them over the part of the diagram that describes the role played by that bird, in our earlier example. (Try not to flip back through the pages.) Then add your own annotations to describe how it works.

Class Adapter



Object Adapter

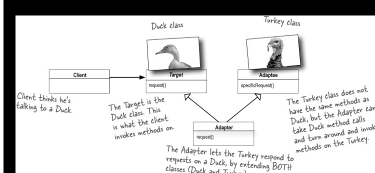




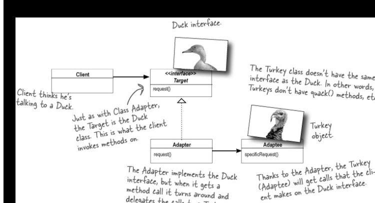
DUCK MAGNETS ANSWER

> **NOTE**
>
> Note: the class adapter uses multiple inheritance, so you can't do it in Java...

Class Adapter



Object Adapter



FIRESIDE CHATS

Tonight's talk: **The Object Adapter and Class Adapter meet face to face.**

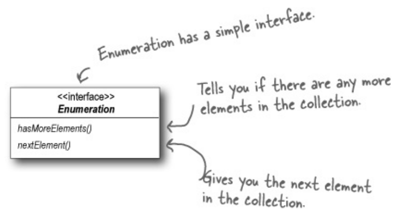
Object Adapter	Class Adapter
Because I use composition I've got a leg up. I can not only adapt an adaptee class, but any of its subclasses.	
	That's true, I do have trouble with that because I am committed to one specific adaptee class, but I have a huge advantage because I don't have to reimplement my entire adaptee. I can also override the behavior of my adaptee if I need to because I'm just subclassing.
In my part of the world, we like to use composition over inheritance; you may be saving a few lines of code, but all I'm doing is writing a little code to delegate to the adaptee. We like to keep things flexible.	
	Flexible maybe, efficient? No. Using a class adapter there is just one of me, not an adapter and an adaptee.
You're worried about one little object? You might be able to quickly override a method, but any behavior I add to my adapter code works with my adaptee class *and* all its subclasses.	
	Yeah, but what if a subclass of adaptee adds some new behavior. Then what?
Hey, come on, cut me a break, I just need to compose with the subclass to make that work.	
	Sounds messy...
You wanna see messy? Look in the mirror!	

Real world adapters

Let's take a look at the use of a simple Adapter in the real world (something more serious than Ducks at least)...

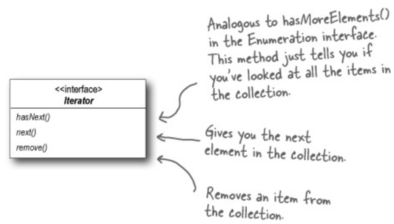
Old world Enumerators

If you've been around Java for a while you probably remember that the early collections types (Vector, Stack, Hashtable, and a few others) implement a method elements(), which returns an Enumeration. The Enumeration interface allows you to step through the elements of a collection without knowing the specifics of how they are managed in the collection.



New world Iterators

When Sun released their more recent Collections classes they began using an Iterator interface that, like Enumeration, allows you to iterate through a set of items in a collection, but also adds the ability to remove items.

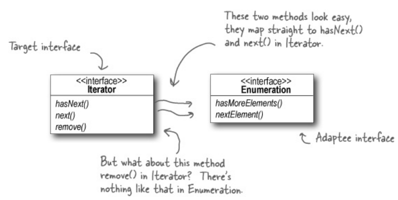


And today...

We are often faced with legacy code that exposes the Enumerator interface, yet we'd like for our new code to use only Iterators. It looks like we need to build an adapter.

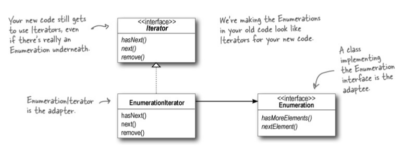
Adapting an Enumeration to an Iterator

First we'll look at the two interfaces to figure out how the methods map from one to the other. In other words, we'll figure out what to call on the adaptee when the client invokes a method on the target.



Designing the Adapter

Here's what the classes should look like: we need an adapter that implements the Target interface and that is composed with an adaptee. The hasNext() and next() methods are going to be straightforward to map from target to adaptee: we just pass them right through. But what do you do about remove()? Think about it for a moment (and we'll deal with it on the next page). For now, here's the class diagram:



Dealing with the remove() method

Well, we know Enumeration just doesn't support remove. It's a "read only" interface. There's no way to implement a fully functioning remove() method on the adapter. The best we can do is throw a runtime exception. Luckily, the designers of the Iterator interface foresaw this need and defined the remove() method so that it supports an UnsupportedOperationException.

This is a case where the adapter isn't perfect; clients will have to watch out for potential exceptions, but as long as the client is careful and the adapter is well documented this is a perfectly reasonable solution.

Writing the EnumerationIterator adapter

Here's simple but effective code for all those legacy classes still producing Enumerations:

```
public class EnumerationIterator implements Iterator
{
    Enumeration enum;

    public EnumerationIterator(Enumeration enum) {
        this.enum = enum;
    }

    public boolean hasNext() {
        return enum.hasMoreElements();
    }

    public Object next() {
        return enum.nextElement();
    }

    public void remove() {
        throw new UnsupportedOperationException();
    }
}
```

Since we're adapting Enumeration to Iterator, our Adapter implements the Iterator interface... it has to look like an Iterator.

The Enumeration we're adapting. We're using composition so we stash it in an instance variable.

The Iterator's hasNext() method is delegated to the Enumeration's hasMoreElements() method...

... and the Iterator's next() method is delegated to the Enumerations's nextElement() method.

Unfortunately, we can't support Iterator's remove() method, so we have to punt (in other words, we give up!). Here we just throw an exception.

EXERCISE

While Java has gone in the direction of the Iterator, there is nevertheless a lot of legacy **client code** that depends on the Enumeration interface, so an Adapter that converts an Iterator to an Enumeration is also quite useful.

Write an Adapter that adapts an Iterator to an Enumeration. You can test your code by adapting an ArrayList. The ArrayList class supports the Iterator interface but doesn't support Enumerations (well, not yet anyway).

BRAIN POWER

Some AC adapters do more than just change the interface – they add other features like surge protection, indicator lights and other bells and whistles.

If you were going to implement these kinds of features, what pattern would you use?

FIRESIDE CHATS

Tonight's talk: **The Decorator Pattern and the Adapter Pattern discuss their differences.**

Decorator	Adapter
I'm important. My job is all about *responsibility* – you know that when a Decorator is involved there's going to be some new responsibilities or behaviors added to your design.	
	You guys want all the glory while us adapters are down in the trenches doing the dirty work: converting interfaces. Our jobs may not be glamorous, but our clients sure do appreciate us making their lives simpler.
That may be true, but don't think we don't work hard. When we have to decorate a big interface, whoa, that can take a lot of code.	
	Try being an adapter when you've got to bring several classes together to provide the interface your client is expecting. Now that's tough. But we have a saying: "an uncoupled client is a happy client."
Cute. Don't think we get all the glory; sometimes I'm just one decorator that is being wrapped by who knows how many other decorators. When a method call gets delegated to you, you have no idea how many other decorators have already dealt with it and you don't know that you'll ever get noticed for your efforts servicing the request.	
	Hey, if adapters are doing their job, our clients never even know we're there. It can be a thankless job. But, the great thing about us adapters is that we allow clients to make use of new libraries and subsets without changing *any* code, they just rely on us to do the conversion for them. Hey, it's a niche, but we're good at it.
Well us decorators do that as well, only we allow *new behavior* to be added to classes without altering existing code. I still say that adapters are just fancy decorators – I mean, just like us, you wrap an object.	
	No, no, no, not at all. We *always* convert the interface of what we wrap, you *never* do. I'd say a decorator is like an adapter; it is just that you don't change the interface!
Uh, no. Our job in life is to extend the behaviors or responsibilities of the objects we wrap, we aren't a *simple*	

pass through.	
	Hey, who are you calling a simple pass through? Come on down and we'll see how long *you* last converting a few interfaces!
Maybe we should agree to disagree. We seem to look somewhat similar on paper, but clearly we are *miles* apart in our *intent*.	
	Oh yeah, I'm with you there.

And now for something different...

There's another pattern in this chapter.

You've seen how the Adapter Pattern converts the interface of a class into one that a client is expecting. You also know we achieve this in Java by wrapping the object that has an incompatible interface with an object that implements the correct one.

We're going to look at a pattern now that alters an interface, but for a different reason: to simplify the interface. It's aptly named the Facade Pattern because this pattern hides all the complexity of one or more classes behind a clean, well-lit facade.

WHO DOES WHAT?

Match each pattern with its intent:

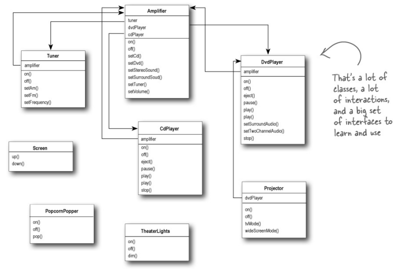
Decorator	Converts one interface to another
Adapter	Doesn't alter the interface, but adds responsibility
Facade	Makes an interface simpler

Home Sweet Home Theater

Before we dive into the details of the Facade Pattern, let's take a look at a growing national obsession: building your own home theater.

You've done your research and you've assembled a killer system complete with a DVD player, a projection video system, an automated screen, surround sound and even a popcorn popper.

Check out all the components you've put together:



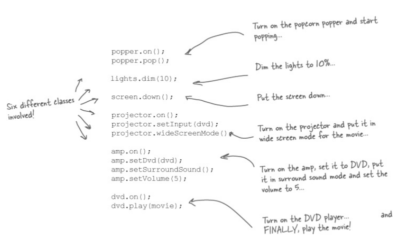
You've spent weeks running wire, mounting the projector, making all the connections and fine tuning. Now it's time to put it all in motion and enjoy a movie...

Watching a movie (the hard way)

Pick out a DVD, relax, and get ready for movie magic. Oh, there's just one thing – to watch the movie, you need to perform a few tasks:

1. **Turn on the popcorn popper**
2. **Start the popper popping**
3. **Dim the lights**
4. **Put the screen down**
5. **Turn the projector on**
6. **Set the projector input to DVD**
7. **Put the projector on wide-screen mode**
8. **Turn the sound amplifier on**
9. **Set the amplifier to DVD input**
10. **Set the amplifier to surround sound**
11. **Set the amplifier volume to medium (5)**
12. **Turn the DVD Player on**
13. **Start the DVD Player playing**

Let's check out those same tasks in terms of the classes and the method calls needed to perform them:



But there's more...

- When the movie is over, how do you turn everything off?

  Wouldn't you have to do all of this over again, in reverse?
- Wouldn't it be as complex to listen to a CD or the radio?
- If you decide to upgrade your system, you're probably going to have to learn a slightly different procedure.

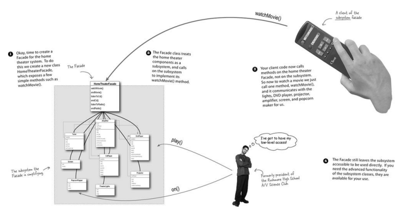
So what to do? The complexity of using your home theater is becoming apparent!

Let's see how the Facade Pattern can get us out of this mess so we can enjoy the movie...

Lights, Camera, Facade!

A Facade is just what you need: with the Facade Pattern you can take a complex subsystem and make it easier to use by implementing a Facade class that provides one, more reasonable interface. Don't worry; if you need the power of the complex subsystem, it's still there for you to use, but if all you need is a straightforward interface, the Facade is there for you.

Let's take a look at how the Facade operates:



THERE ARE NO DUMB QUESTIONS

Q: **Q: If the Facade encapsulates the subsystem classes, how does a client that needs lower-level functionality gain access to them?**

A: **A:** *Facades don't "encapsulate" the subsystem classes; they merely provide a simplified interface to their functionality. The subsystem classes still remain available for direct use by clients that need to use more specific interfaces. This is a nice property of the Facade Pattern: it provides a simplified interface while still exposing the full functionality of the system to those who may need it.*

Q: **Q: Does the facade add any functionality or does it just pass through each request to the subsystem?**

A: **A:** *A facade is free to add its own "smarts" in addition to making use of the subsystem. For instance, while our home theater facade doesn't implement any new behavior, it is smart enough to know that the popcorn popper has to be turned on before it can pop (as well as the details of how to turn on and stage a movie showing).*

Q: **Q: Does each subsystem have only one facade?**

A: **A:** *Not necessarily. The pattern certainly allows for any number of facades to be created for a given subsystem.*

Q: **Q: What is the benefit of the facade other than the fact that I now have a simpler interface?**

A: **A:** *The Facade Pattern also allows you to decouple your client implementation from any one subsystem. Let's say for instance that you get a big raise and decide to upgrade your home theater to all new components that have different interfaces. Well, if you coded your client to the facade rather than the subsystem, your client code doesn't need to change, just the facade (and hopefully the manufacturer is supplying that!).*

Q: **Q: So the way to tell the difference between the Adapter Pattern and the Facade Pattern is that the adapter wraps one class and the facade may represent many classes?**

A: **A:** *No! Remember, the Adapter Pattern changes the interface of one or more classes into one interface that a client is expecting. While most textbook examples show the adapter adapting one class, you may need to adapt many classes to provide the interface a client is coded to. Likewise, a Facade may provide a simplified interface to a single class with a very complex interface.*

The difference between the two is not in terms of how many classes they "wrap," it is in their ***intent****. The intent of the Adapter Pattern is to* ***alter*** *an interface so that it matches one a client is expecting. The intent of the Facade Pattern is to provide a* ***simplified*** *interface to a subsystem.*

A facade not only simpl es an interface, it decouples a client from a subsystem of components.

Facades and adapters may wrap multiple classes, but a facade's intent is to simplify, while an adapter's is to convert the interface to something different.

Constructing your home theater facade

Let's step through the construction of the HomeTheaterFacade: The first step is to use composition so that the facade has access to all the components of the subsystem:

```java
public class HomeTheaterFacade {
    Amplifier amp;
    Tuner tuner;
    DvdPlayer dvd;
    CdPlayer cd;
    Projector projector;
    TheaterLights lights;
    Screen screen;
    PopcornPopper popper;

    public HomeTheaterFacade(Amplifier amp,
                 Tuner tuner,
                 DvdPlayer dvd,
                 CdPlayer cd,
                 Projector projector,
                 Screen screen,
                 TheaterLights lights,
                 PopcornPopper popper) {

        this.amp = amp;
        this.tuner = tuner;
        this.dvd = dvd;
        this.cd = cd;
        this.projector = projector;
        this.screen = screen;
        this.lights = lights;
        this.popper = popper;
    }

        // other methods here

}
```

Here's the composition; these are all the components of the subsystem we are going to use.

The facade is passed a reference to each component of the subsystem in its constructor. The facade then assigns each to the corresponding instance variable.

We're just about to fill these in...

Implementing the simplified interface

Now it's time to bring the components of the subsystem together into a unified interface. Let's implement the watchMovie() and endMovie() methods:

```java
public void watchMovie(String movie) {
    System.out.println("Get ready to watch a movie...");
    popper.on();
    popper.pop();
    lights.dim(10);
    screen.down();
    projector.on();
    projector.wideScreenMode();
    amp.on();
    amp.setDvd(dvd);
    amp.setSurroundSound();
    amp.setVolume(5);
    dvd.on();
    dvd.play(movie);
}

public void endMovie() {
    System.out.println("Shutting movie theater down...");
    popper.off();
    lights.on();
    screen.up();
    projector.off();
    amp.off();
    dvd.stop();
    dvd.eject();
    dvd.off();
}
```

watchMovie() follows the same sequence we had to do by hand before, but wraps it up in a handy method that does all the work. Notice that for each task we are delegating the responsibility to the corresponding component in the subsystem.

And endMovie() takes care of shutting everything down for us. Again, each task is delegated to the appropriate component in the subsystem.

BRAIN POWER

Think about the facades you've encountered in the Java API. Where would you like to have a few new ones?

Time to watch a movie (the easy way)

It's SHOWTIME!

```java
public class HomeTheaterTestDrive {
    public static void main(String[] args) {
        // instantiate components here

        HomeTheaterFacade homeTheater =
                new HomeTheaterFacade(amp, tuner, dvd, cd,
                        projector, screen, lights, popper);

        homeTheater.watchMovie("Raiders of the Lost Ark");
        homeTheater.endMovie();
    }
```

Here we're creating the components right in the test drive. Normally the client is given a facade, it doesn't have to construct one itself.

First you instantiate the Facade with all the components in the subsystem.

Use the simplified interface to first start the movie up, and then shut it down.

Here's the output.

Calling the Facade's watchMovie() does all this work for us...

...and here, we're done watching the movie, so calling endMovie() turns everything off.

```
File Edit Window Help SnakesWhy'dItHaveToBeSnakes?
%java HomeTheaterTestDrive

Get ready to watch a movie...
Popcorn Popper on
Popcorn Popper popping popcorn!
Theater Ceiling Lights dimming to 10%
Theater Screen going down
Top-O-Line Projector on
Top-O-Line Projector in widescreen mode (16x9 aspect ratio)
Top-O-Line Amplifier on
Top-O-Line Amplifier setting DVD player to Top-O-Line DVD Player
Top-O-Line Amplifier surround sound on (5 speakers, 1 subwoofer)
Top-O-Line Amplifier setting volume to 5
Top-O-Line DVD Player on
Top-O-Line DVD Player playing "Raiders of the Lost Ark"
Shutting movie theater down...
Popcorn Popper off
Theater Ceiling Lights on
Theater Screen going up
Top-O-Line Projector off
Top-O-Line Amplifier off
Top-O-Line DVD Player stopped "Raiders of the Lost Ark"
Top-O-Line DVD Player eject
Top-O-Line DVD Player off
%
```

Facade Pattern defined

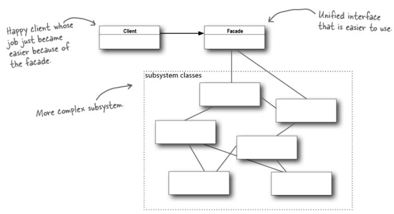
To use the Facade Pattern, we create a class that simplifies and unifies a set of more complex classes that belong to some subsystem. Unlike a lot of patterns, Facade is fairly straightforward; there are no mind bending abstractions to get your head around. But that doesn't make it any less powerful: the Facade

Pattern allows us to avoid tight coupling between clients and subsystems, and, as you will see shortly, also helps us adhere to a new object oriented principle.

Before we introduce that new principle, let's take a look at the official definition of the pattern:

> NOTE
>
> **The Facade Pattern** provides a unified interface to a set of interfaces in a subsytem. Facade defines a higher-level interface that makes the subsystem easier to use.

There isn't a lot here that you don't already know, but one of the most important things to remember about a pattern is its intent. This definition tells us loud and clear that the purpose of the facade it to make a subsystem easier to use through a simplified interface. You can see this in the pattern's class diagram:



That's it; you've got another pattern under your belt! Now, it's time for that new OO principle. Watch out, this one can challenge some assumptions!

The Principle of Least Knowledge

The Principle of Least Knowledge guides us to reduce the interactions between objects to just a few close "friends." The principle is usually stated as:

> NOTE
>
> *Design Principle*
>
> *Principle of Least Knowledge - talk only to your immediate friends.*

But what does this mean in real terms? It means when you are designing a system, for any object, be careful of the number of classes it interacts with and also how it comes to interact with those classes.

This principle prevents us from creating designs that have a large number of classes coupled together so that changes in one part of the system cascade to other parts. When you build a lot of dependencies between many classes, you are building a fragile system that will be costly to maintain and complex for others to understand.

> BRAIN POWER
>
> How many classes is this code coupled to?

```
public float getTemp() {
     return station.getThermometer().getTemperature();
}
```

How NOT to Win Friends and Influence Objects

Okay, but how do you keep from doing this? The principle provides some guidelines: take any object; now from any method in that object, the principle tells us that we should only invoke methods that belong to:

- The object itself
- Objects passed in as a parameter to the method
- Any object the method creates or instantiates
- Any components of the object

Notice that these guidelines tell us not to call methods on objects that were returned from calling other methods!!

Think of a "component" as any object that is referenced by an instance variable. In other words think of this as a HAS-A relationship.

This sounds kind of stringent doesn't it? What's the harm in calling the method of an object we get back from another call? Well, if we were to do that, then we'd be making a request of another object's subpart (and increasing the number of objects we directly know). In such cases, the principle forces us to ask the object to make the request for us; that way we don't have to know about its component objects (and we keep our circle of friends small). For example:

Without the Principle

```
public float getTemp() {
    Thermometer thermometer = station.getThermometer();
    return thermometer.getTemperature();
}
```

Here we get the thermometer object from the station and then call the getTemperature() method ourselves.

With the Principle

```
public float getTemp() {
    return station.getTemperature();
}
```

When we apply the principle, we add a method to the Station class that makes the request to the thermometer for us. This reduces the number of classes we're dependent on.

Keeping your method calls in bounds...

Here's a Car class that demonstrates all the ways you can call methods and still adhere to the Principle of Least Knowledge:

```
public class Car {
      Engine engine;
      // other instance variables

      public Car() {
            // initialize engine, etc.
      }

      public void start(Key key) {
            Doors doors = new Doors();

            boolean authorized = key.turns();

            if (authorized) {
                  engine.start();
                  updateDashboardDisplay();
                  doors.lock();
            }
      }

      public void updateDashboardDisplay() {
            // update display
      }
```

Here's a component of this class. We can call its methods.

Here we're creating a new object, its methods are legal.

You can call a method on an object passed as a parameter.

You can call a method on a component of the object.

You can call a local method within the object.

You can call a method on an object you create or instantiate.

THERE ARE NO DUMB QUESTIONS

Q: **Q: There is another principle called the Law of Demeter; how are they related?**

A: ***A:** The two are one and the same and you'll encounter these terms being intermixed. We prefer to use the Principle of Least Knowledge for a couple of reasons: (1) the name is more intuitive and (2) the use of the word "Law" implies we always have to apply this principle. In fact, no principle is a law, all principles should be used when and where they are helpful. All design involves tradeoffs (abstractions versus speed, space versus time, and so on) and while principles provide guidance, all factors should be taken into account before applying them.*

Q: **Q: Are there any disadvantages to applying the Principle of Least Knowledge?**

A: ***A:** Yes; while the principle reduces the dependencies between objects and studies have shown this reduces software maintenance, it is also the case that applying this principle results in more "wrapper" classes being written to handle method calls to other components. This can result in increased complexity and development time as well as decreased runtime performance.*

SHARPEN YOUR PENCIL

Do either of these classes violate the Principle of Least Knowledge? Why or why not?

```
public House {
    WeatherStation station;

    // other methods and constructor

    public float getTemp() {
         return station.getThermometer().getTemperature();
    }
}
```

```
public House {
     WeatherStation station;

    // other methods and constructor

    public float getTemp() {
         Thermometer thermometer = station.getThermometer();
         return getTempHelper(thermometer);
    }

    public float getTempHelper(Thermometer thermometer) {
        return thermometer.getTemperature();
    }
}
```

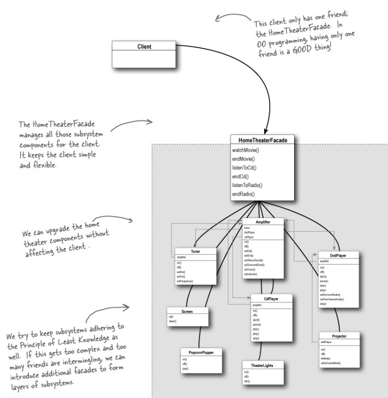
BRAIN POWER

Q: *Can you think of a common use of Java that violates the Principle of Least Knowledge?*

Should you care?

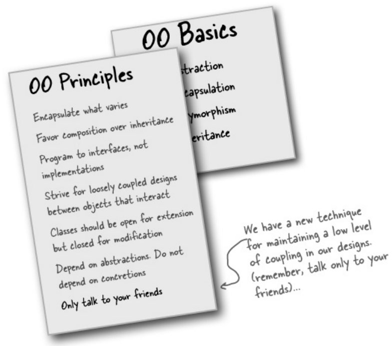
A: *Answer: How about System.out.println()?*

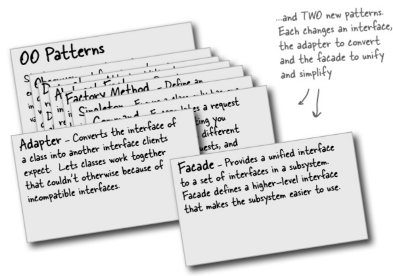
The Facade and the Principle of Least Knowledge



Tools for your Design Toolbox

Your toolbox is starting to get heavy! In this chapter we've added a couple of patterns that allow us to alter interfaces and reduce coupling between clients and the systems they use.

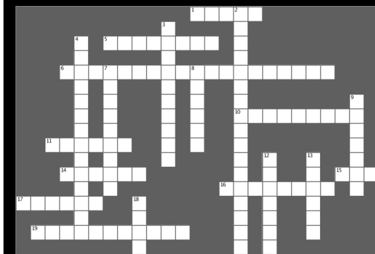




BULLET POINTS

- When you need to use an existing class and its interface is not the one you need, use an adapter.
- When you need to simplify and unify a large interface or complex set of interfaces, use a facade.
- An adapter changes an interface into one a client expects.
- A facade decouples a client from a complex subsystem.
- Implementing an adapter may require little work or a great deal of work depending on the size and complexity of the target interface.
- Implementing a facade requires that we compose the facade with its subsystem and use delegation to perform the work of the facade.
- There are two forms of the Adapter Pattern: object and class adapters. Class adapters require multiple inheritance.
- You can implement more than one facade for a subsystem.
- An adapter wraps an object to change its interface, a decorator wraps an object to add new behaviors and responsibilities, and a facade "wraps" a set of objects to simplify.

Yes, it's another crossword. All of the solution words are from this chapter.

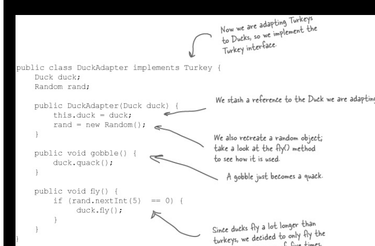


Across	Down
1. True or false, Adapters can only wrap one object 5. An Adapter _____ an interface 6. Movie we watched (5 words) 10. If in Europe you might need one of these (two words) 11. Adapter with two roles (two words) 14. Facade still _____ low level access 15. Ducks do it better than Turkeys 16. Disadvantage of the Principle of Least Knowledge: too many ___________ 17. A ___________ simplifies an interface 19. New American dream (two words)	2. Decorator called Adapter this (3 words) 3. One advantage of Facade 4. Principle that wasn't as easy as it sounded (two words) 7. A ______ adds new behavior 8. Masquerading as a Duck 9. Example that violates the Principle of Least Knowledge: System.out.______ 12. No movie is complete without this 13. Adapter client uses the _________ interface 18. An Adapter and a Decorator can be said to ______ an object

Exercise Solutions

SHARPEN YOUR PENCIL

Let's say we also need an adapter that converts a Duck to a Turkey. Let's call it DuckAdapter. Write that class:



Now we are adapting Turkeys to Ducks, so we implement the Turkey interface.

```
public class DuckAdapter implements Turkey {
    Duck duck;
    Random rand;

    public DuckAdapter(Duck duck) {
        this.duck = duck;
        rand = new Random();
    }

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

    public void fly() {
        if (rand.nextInt(5)  == 0) {
             duck.fly();
        }
    }
}
```

We stash a reference to the Duck we are adapting.

We also recreate a random object; take a look at the fly() method to see how it is used.

A gobble just becomes a quack.

Since ducks fly a lot longer than turkeys, we decided to only fly the duck on average one of five times.

SHARPEN YOUR PENCIL

Do either of these classes violate the Principle of Least Knowledge? For each, why or why not?

```
public House {
    WeatherStation station;

    // other methods and constructor

    public float getTemp() {
         return station.getThermometer().getTemperature();
    }
}
```

Violates the Principle of Least Knowledge! You are calling the method of an object returned from another call.

```
public House {
     WeatherStation station;

    // other methods and constructor

    public float getTemp() {
         Thermometer thermometer = station.getThermometer();
         return getTempHelper(thermometer);
    }

    public float getTempHelper(Thermometer thermometer) {
        return thermometer.getTemperature();
    }
}
```

Doesn't violate Principle of Least Knowledge! This seems like hacking our way around the principle. Has anything really changed since we just moved out the call to another method?

You've seen how to implement an adapter that adapts an Enumeration to an Iterator; now write an adapter that adapts an Iterator to an Enumeration.

```
public class IteratorEnumeration implements Enumeration {
    Iterator iterator;

    public IteratorEnumeration(Iterator iterator) {
        this.iterator = iterator;
    }

    public boolean hasMoreElements() {
        return iterator.hasNext();
    }

    public Object nextElement() {
        return iterator.next();
    }
}
```

WHO DOES WHAT?

Match each pattern with its intent: