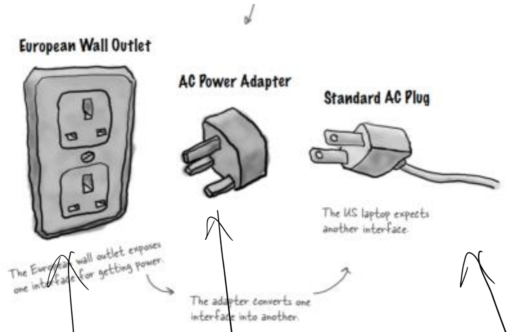
ADAPTER DESIGN PATTERN

Tuesday, September 12, 2023 2:25 PM

Need of adaptor design pattern?

→ when situation like this appear's, i.e., we have data in one form but we need it in other.

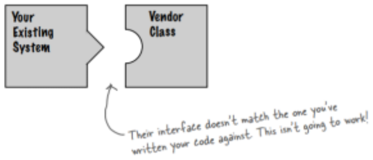

European Wall Outlet
AC Power Adapter
Standard AC Plug
The European wall outlet exposes one interface for getting power.
The adapter converts one interface into another.
The US laptop expects another interface.

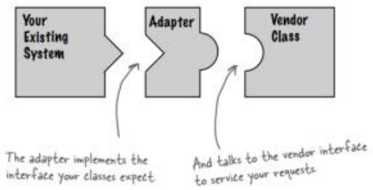
Input data

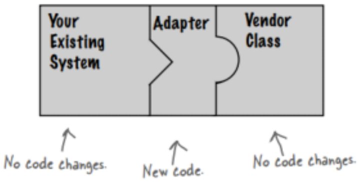
Adaptor

Required output

↳ which converts Input to desired output.


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


Your Existing System
Adapter
Vendor Class
The adapter implements the interface your classes expect.
And talks to the vendor interface to service your requests.



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


public class MallardDuck implements Duck {
 public void quack() {
 System.out.println("Quack");
 }
 public void fly() {
 System.out.println("I'm flying");
 }
}
```

Required output

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


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");
 }
}
```

Input

```
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();
 }
 }
}
```

adaptor to convert Input → Required output.

```
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();
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();
}
}
```

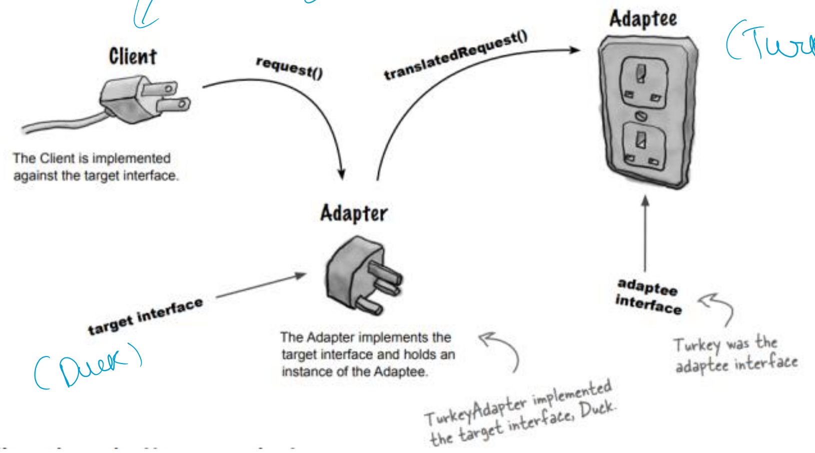
Client

Testing adapter functionality.

CONSOLE OUTPUT:

```
%java RemoteControlTest
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
```

It understands Target interface only(Duck)



(Turkey)

(Duck)

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

Here's how the Client uses the Adapter

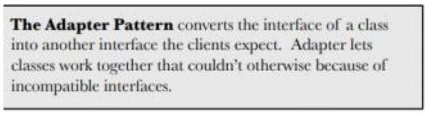
1. **The client makes a request to the adapter by calling a method on it using the target interface.**
2. **The adapter translates the request into one or more calls on the adaptee using the adaptee interface.**
3. **The client receives the results of the call and never knows there is an adapter doing the translation.**

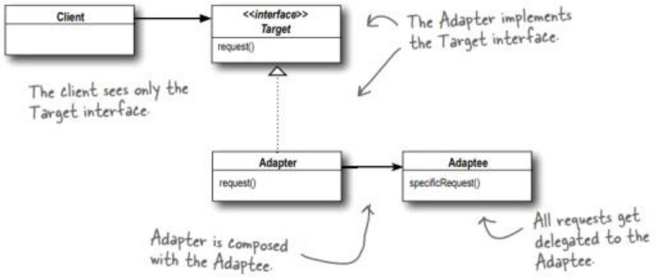
Note that the Client and Adaptee are decoupled – neither knows about the other.

Can we write two way adapter?

↳ Yes we can, just implement both duck & Turkey. and use duck & Turkey as component too

and use duck & Turkey as component too
& write implementation of both.

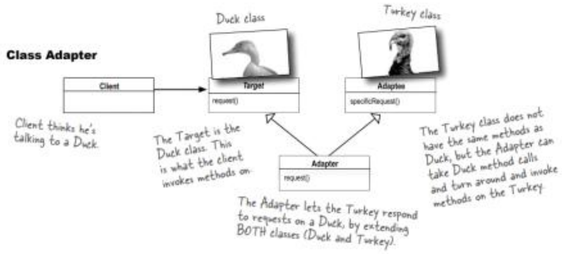

The Adapter Pattern converts the interface of a class into another interface the clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

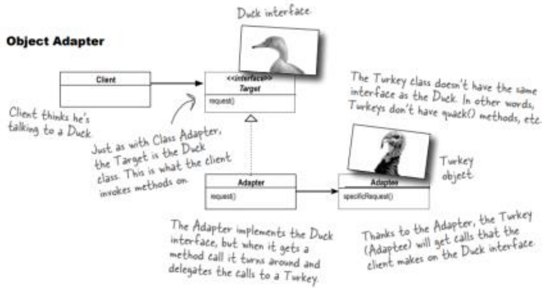

Client
<<interface>>
Target
request()
The Adapter implements the Target interface.
The client sees only the Target interface.
Adapter
request()
Adaptee
specificRequest()
Adapter is composed with the Adaptee.
All requests get delegated to the Adaptee.

we have learnt about object adapter till now.
we don't deal with class adapter in Java
because multiple inheritance not allowed in Java

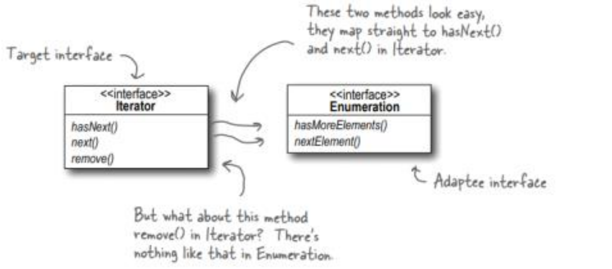
Duck Magnets Answer

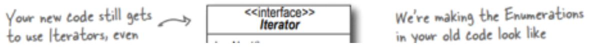
Note: the class adapter uses multiple inheritance, so you can't do it in Java...


Class Adapter
Duck class
Turkey class
Client
Target
request()
Adaptee
specificRequest()
Adapter
request()
Client thinks he's talking to a Duck
The Target is the Duck class. This is what the client invokes methods on.
The Turkey class does not have the same methods as Duck, but the Adapter can take Duck method calls and turn around and invoke methods on the Turkey
The Adapter lets the Turkey respond to requests on a Duck, by extending BOTH classes (Duck and Turkey).

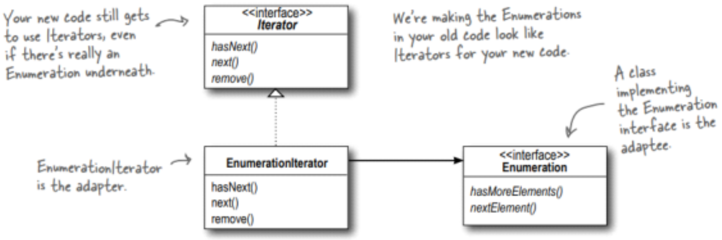

Object Adapter
Duck interface
Client
<<interface>>
Target
request()
Client thinks he's talking to a Duck
Just as with Class Adapter, the Target is the Duck class. This is what the client invokes methods on.
The Turkey class doesn't have the same interface as the Duck. In other words, Turkeys don't have quack() methods, etc.
Turkey object
Adapter
request()
Adaptee
specificRequest()
The Adapter implements the Duck interface, but when it gets a method call it turns around and delegates the calls to a Turkey.
Thanks to the Adapter, the Turkey (Adaptee) will get calls that the client makes on the Duck interface.

Real world Example to convert
Enumeration -> Iterator

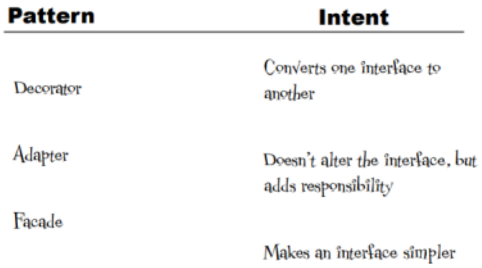

Target interface
These two methods look easy, they map straight to hasNext() and next() in Iterator.
<<interface>>
Iterator
hasNext()
next()
remove()
<<interface>>
Enumeration
hasMoreElements()
nextElement()
Adaptee interface
But what about this method remove() in Iterator? There's nothing like that in Enumeration.


Your new code still gets to use Iterators, even
<<interface>>
Iterator
We're making the Enumerations in your old code look like

```
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() {
```


Your new code still gets to use Iterators, even if there's really an Enumeration underneath.
<<interface>>
Iterator
hasNext()
next()
remove()
We're making the Enumerations in your old code look like Iterators for your new code.
A class implementing the Enumeration interface is the adaptee.
EnumerationIterator is the adapter.
EnumerationIterator
hasNext()
next()
remove()
<<interface>>
Enumeration
hasMoreElements()
nextElement()

```
public Object next() {
return enum.nextElement();
}
public void remove() {
throw new UnsupportedOperationException();
}
```



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

Youtube channel : Shubham Hardtesh