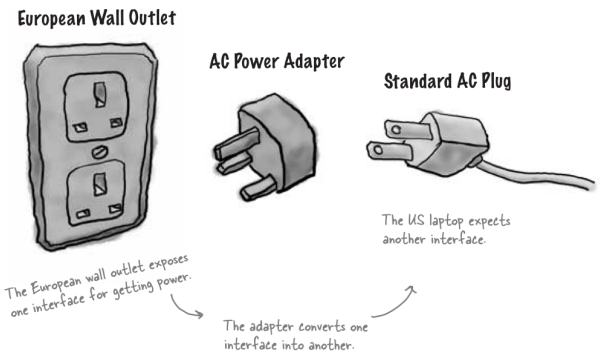
# The Adapter Analogy




# Writing an Adapter

- Back to Ducks… but with interfaces this time:

```
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!");
  }
}
```

# A New Fowl

```
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!");
  }
}
```

# The TurkeyAdapter

## The TurkeyAdapter

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

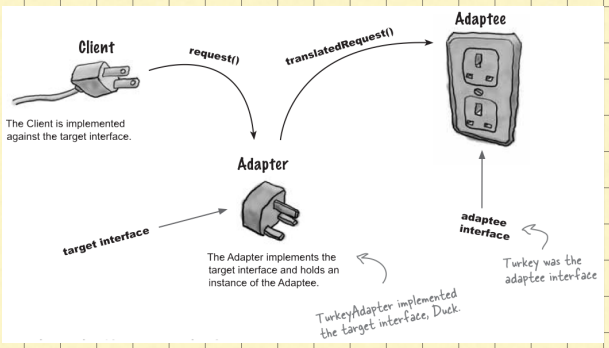
```
public class TurkeyAdapter implements Duck {
  Turkey turkey;
  public TurkeyAdapter(Turkey turkey) {
    this.turkey = turkey;
  }
  public void quack() {
    turkey.gobble();
  }
  public void fly() {
    for(int i = 0; i < 5; i++) {
      turkey.fly();
    }
  }
}
```

We need to get a reference to the object that we're adapting; here we just do that through the constructor.

We have to implement all of the methods in the interface

Even though both interfaces have the fly() method, the turkey's fly is different than the duck's. So we have to implement that logic.

# The Adapter Pieces



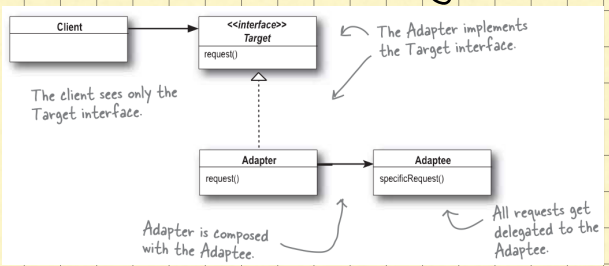

# Questions

- How much adapting can be done in an adapter?
  - That really depends on the particular situation and the particular interfaces. It could be just basic translation or massive amounts of work
- Does an adapter always wrap only one class?
  - The real world can be messier; an adapter could wrap two or more adaptees needed to implement the target interface
  - However, the Adapter *always* converts one interface to another (the point is just that the definition of "interface" may not be limited to a single class)

# The Adapter Pattern

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

# The Adapter Class Diagram

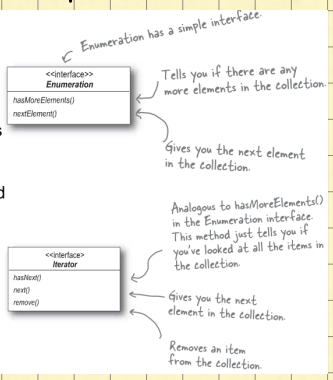



# Adapter Pattern and Good OO Design

- Favor composition
  - The adaptee is wrapped with an altered interface using composition
- Programming to an interface not an implementation
  - The client is only aware of the target interface

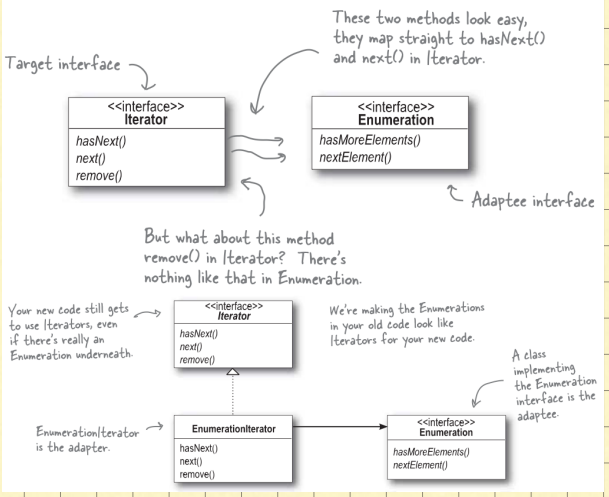
# An Example Adapter in the Wild

- Old World Enumerators…
  - Early Java collections types implemented an `elements()` method that returned an `Enumeration` of the elements that you could then step through without knowing how the collection was implemented
- New World Iterators…
  - The Collections classes use an `Iterator` interface that implements a similar capability but also allows item removal
- Never the twain shall meet
  - Oh, wait…




# Backwards Compatibility

- You know what that means, right?
- We often have to work with legacy code that does it the old way.
  - But our clients insist (and they should) on using the new way.




# How to Handle the remove() method

- Enumeration simply doesn't support remove; it's "read only"
- We can, however, throw a runtime exception if someone tries to call `remove()` on an `EnumerationIterator`
  - The Iterator class supports this; it's remove method supports throwing an `UnsupportedOperationException`
- In the end, the adapter isn't perfect; clients will still have to deal with potential exceptions

# The EnumerationIterator Adapter

```
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();
  }
}
```

Our adapter has to implement the Iterator interface

Use composition to stash the enum

The Iterator's `hasNext()` method is a direct map to the Enumeration's `hasMoreElements()` method

The Iterator's `next()` method is a direct map to the Enumeration's `nextElement()` method

Throw the `UnsupportedOerationException` if someone tries to call remove