The Adapter Pattern

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.

Toni Sellarès
Universitat de Girona

Wrapping Objects to Unify Interfaces

Sometimes, an existing class may provide the functionality required by a client, but its interface may not be what the client expects.

In such cases, the existing interface needs to be converted into another interface, which the client expects, preserving the reusability of the existing class.

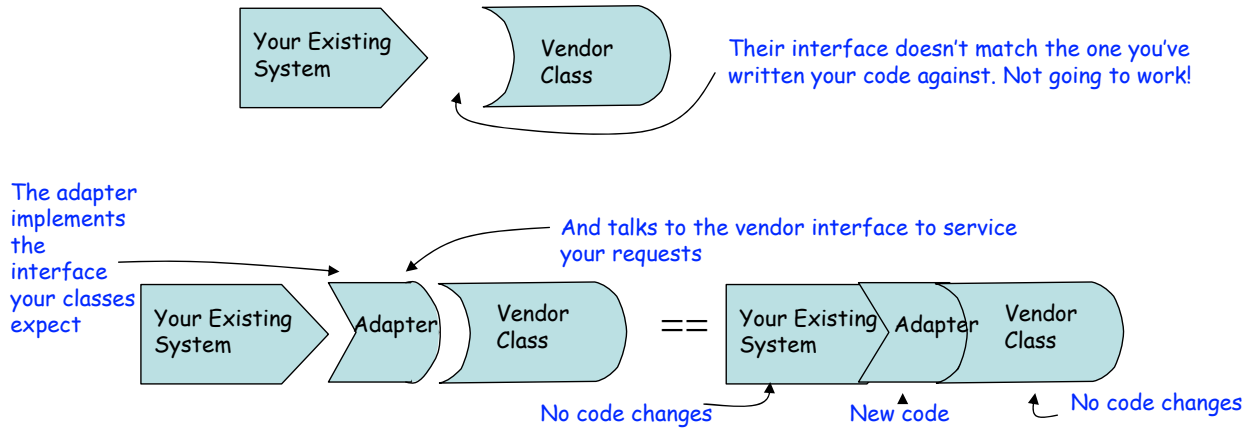
The Adapter pattern suggests defining a wrapper class around the object with the incompatible interface.

This wrapper object is referred as an *adapter* and the object it wraps is referred to as an *adaptee*.

Adapter Example

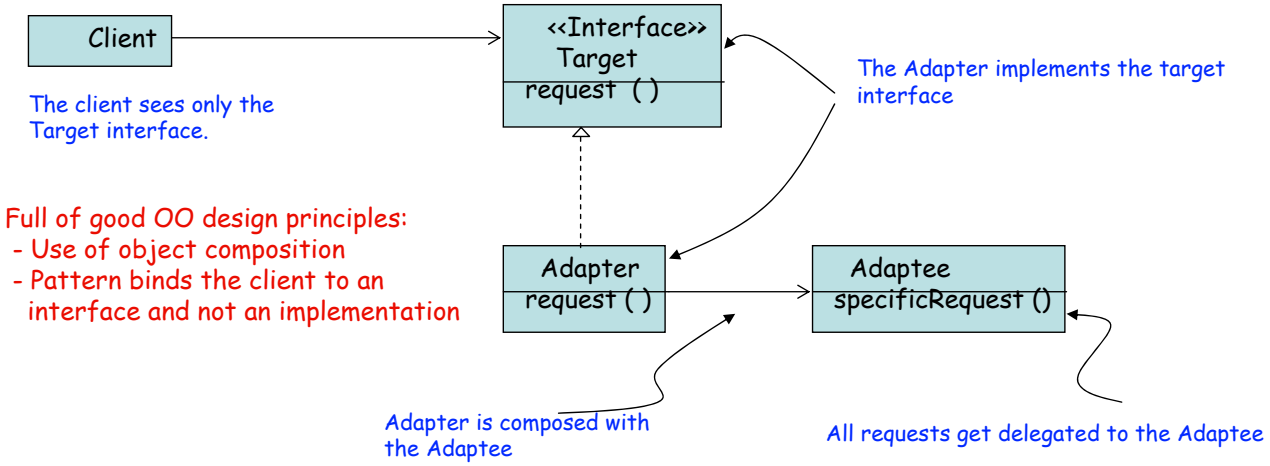
You have an existing software system that you need to work a new vendor library into, but the new vendor designed their interfaces differently than the last vendor.

What to do? Write a class that adapts the new vendor interface into the one you're expecting.



The Adapter Pattern Defined

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.



Adapter Pattern

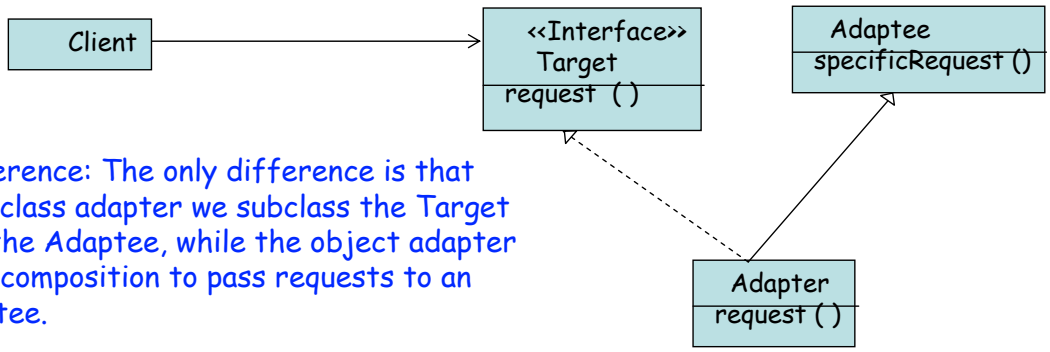
Delegation is used to bind an **Adapter** and an **Adaptee**.

Interface inheritance is use to specify the interface of the **Adapter** class.

Target and **Adaptee** (usually called legacy system) pre-exist the **Adapter.**

Object and Class Adapters

- There are two types of Adapters
  - *Object* Adapter: what we have seen so far.
  - *Class* Adapter: not as common as it uses multiple inheritance, which isn't possible in Java.



Difference: The only difference is that with class adapter we subclass the Target and the Adaptee, while the object adapter uses composition to pass requests to an adaptee.

Object Adapters and Class Adapters use two different means of adapting the adaptee: composition versus inheritance.

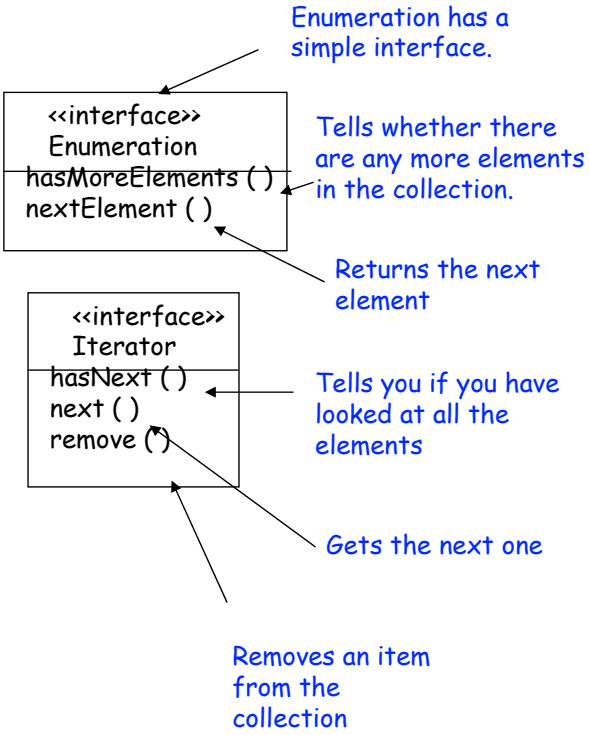
Adapter: Structural Example

```
/**
 * This is the interface the client expects.
 */
public interface Target
{
          public abstract void request();
}

/**
 * This adapts the Adaptee so the client
 * may use it.
 */
public class Adapter implements Target
{
          private Adaptee delegate;
          public Adapter() {
                    delegate = new Adaptee();
          }
          public void request() {
                    delegate.delegatedRequest();
          }
}
```

```
public class Adaptee
{
         public void delegatedRequest() {
                  System.out.println("This is the delegated method.");
         }
}

public class TestAdapter
{
         public static void main(String[] args) {
                  Target client = new Adapter();
                  client.request();
         }
}
```

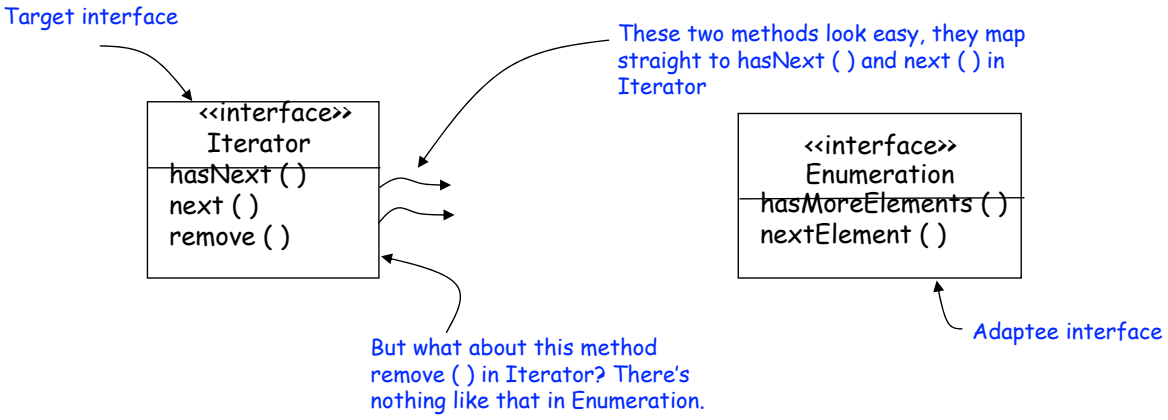
Adapter: Example 1

- Old world **Enumerators**
- New world **Iterators**
- And today…legacy code that exposes the **Enumerator** interface. Yet we want new code to use **Iterators**. Need an adapter.

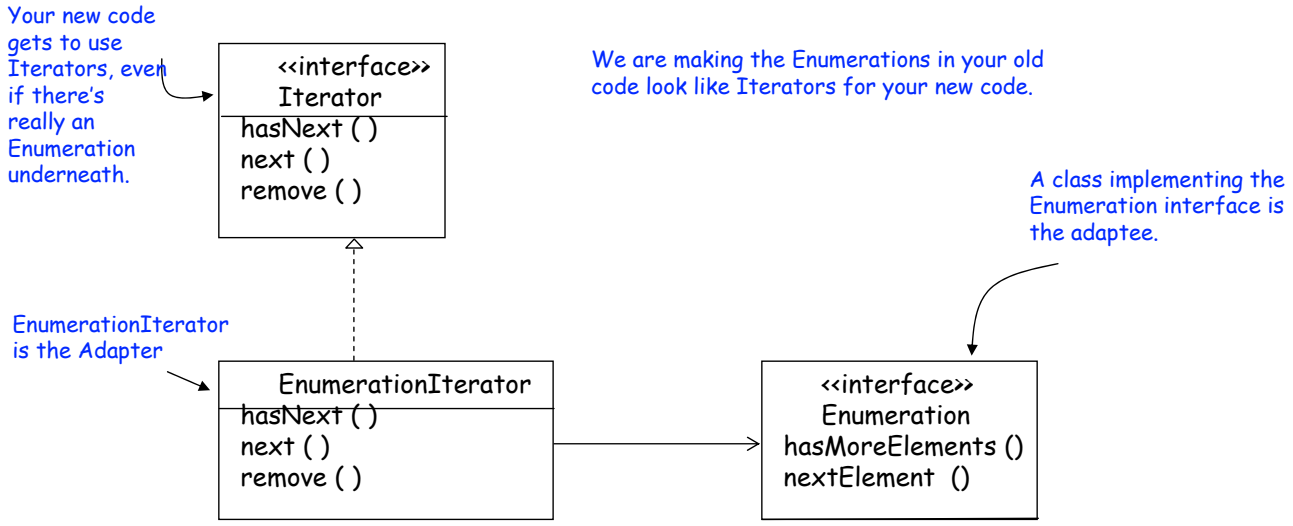


Adapting an Enumeration to an Iterator

First step: examine the two interfaces



Designing the Adapter



Dealing with the remove ( ) method

- Enumeration is a "read only" interface - it does not support the `remove` ( ) method.
  - Implies there is no real way to implement a fully functioning `remove` () method.
  - The best that can be done is to throw a runtime exception.
  - Iterator designers foresaw the need and have implemented an `UnsupportedOperationException`.
- Here the adapter is not perfect but is a reasonable solution as long as the client is careful and the adapter is well-documented.

EnumerationIterator - The Code

```
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 are adapting Enumeration to Iterator, the EnumerationIterator must implement the Iterator interface -- it has to look like the Iterator.

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

hasNext ( ) and next () are implemented by delegating to the appropriate methods in the Enumeration.

For the remove ( ) we simply throw an exception.

Adapter: Example 2

Adapt from integer Set to integer Priority Queue

- Original
  - Integer set does not support Priority Queue.
- Using Adapter pattern
  - Adapter provides interface for using Set as Priority Queue.
  - Add needed functionality in Adapter methods.

```
public interface PriorityQueue {      // Priority Queue
      void add(Object o);
      int size();
      Object removeSmallest();
}


public class PriorityQueueAdapter implements PriorityQueue {
      Set s;
      PriorityQueueAdapter(Set s){ this.s = s; }
      public void add(Object o){ s.add(o); }
      int size(){ return s.size(); }
      public Integer removeSmallest() {
         Integer smallest = Integer.MAX_VALUE;
          Iterator it = s.iterator();
          while ( it.hasNext() ) {
              Integer i = it.next();
              if (i.compareTo(smallest) < 0)
                  smallest = i;
          }
          s.remove(smallest);
          return smallest;
      }
}
```

Summary

- When you need to use an existing class and its interface is not the one you need, use an adapter: allows collaboration between classes with incompatible interfaces.
- An adapter changes an interface into one a client expects.
- Implementing an adapter may require little work or a great deal of work depending on the size and complexity of the target interface.
- There are two forms of adapter patterns: object and class adapters.
- Class adapters require multiple inheritance.