Introduction to Adapters

- We may have an application that needs to use libraries/a different application/you-name-it, but the thing we want to call has a different interface than our caller.

- Alternative 1: Re-write the caller
  - Ugly, messy, error-prone (Equivalent to changing the power cable in our electrical plug when we go abroad)

- Alternative 2: Re-write the called libraries/classes
  - May not have the source code
  - As ugly and error-prone as Alternative 1

- Alternative 3: Write an adapter

- The adapter converts all requests to a language the adaptee understands

See http://mypages.valdosta.edu/dgibson/courses/cs4322/Lessons/Adapter/AdapterNotes.pdf
Another example

Client class programmed against a Vendor class. Thus, the Client is strongly coupled with the Vendor class.

Later, it is decided to change vendors. Vendor2 is selected which has different method names.
The Client must be modified in order to adapt to the new vendor.

- What was the problem?
  - The Client class encapsulates some portion of application logic, which is intertwined with the Vendor class.
  - This causes the strong coupling/dependence.
  - Thus, modifying the Client code to adapt to the new vendor could result in errors being injected into the application logic.

- How do we mitigate this situation where we have a Client coded against a Vendor, where the Vendor will change?
Two helpful design principles

• Identify the aspects of your application that vary and separate them from what stays the same.
• Program to an interface, not an implementation.
• The first principle suggests that we separate the application logic from the code tied to the vendor.
• The second design principle indicates how we may achieve that.
• We program against an abstraction of all Vendor classes and then use an Adapter to encapsulate vendor specific code required to implement the abstraction (interface).
• We can even require that Vendors supply the adapter.
Idea of 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.
Suppose method $x()$ corresponds to method $a()$ for Vendor 1. Then $Adapter1$’s implementation of $x()$ would just delegate the call to $Vendor1$ with a line of code like: `vendor.a()`. 

Object Adapters
Adapter pattern

As mentioned earlier, an adapter converts the interface of a class into another interface clients expect.

• The client and the adapted object remain independent.

• An adapter adds an extra level of indirection.

• Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces.
Adapter pattern

You have an existing **client** (application) that uses an **old interface** to an existing package.

To design a **new interface** to a new package you need to produce an **adapter** so that:

The client can use the new interface instead of the old one (without changing the client)
Illustration of Adapter Pattern

Client

Old package

Old interface

New package

New interface

Adapter
Recap: Implementing Adapter Pattern Using Object Composition

Client

Old Interface
Request()

Adaptee

Adapter
Request()

New Interface
SpecificRequest()

inherit

adaptee
Adapter in a nutshell

• How to use a class that allows you to get the right functionalities but does not have the signatures of the operations that the client wants?

• An adapter converts the operations in the class so that the client gets the functionalities in the interfaces s/he wants.

• Client and the adapted objects remain independent.

• Some people call this “wrapper”.
Object & Class Adapters

- The Adapter discussed above is known as object adapters which result in 2 objects (Adapter and Vendor) at runtime.
Another solution is a class adapter where the Adapter extends the Vendor class.

This results in only 1 object at runtime.
Consequences of Using an Adapter

Class Adapter:
- Adapts Adaptee to Target by committing to a concrete Adapter class
- Can’t adapt a class and all its subclasses
- Lets Adapter override some of Adaptee’s behavior
- No extra objects due to adapter usage

Object Adapter:
- Lets a single Adapter work with many Adaptees
- Can add functionality to all Adaptees
When to use an object adapter?

- An object adapter deals with subclasses better than a class adapter. A Vendor subclass can simply be composed with the Adapter when the Adapter is created.
A class adapter might not make sense unless we have an Adapter for each subclass.
A simple example with two legacy classes

```java
public class LegacyLine {
    public void draw(int x1, int y1, int x2, int y2) {
        System.out.println("line from (" + x1 + "," + y1 + ") to 
                         (" + x2 + "," + y2 + ")");
    }
}

public class LegacyRectangle {
    public void draw(int x, int y, int w, int h) {
        System.out.println("rectangle at (" + x + "," + y + ") with width 
                           + w + " and height " + h);
    }
}
```
Before using adapter we must check what kind of object the client program has and apply specific calls to the appropriate classes.
public class BeforeAdapterDemo {
    public static void main(String[] args) {
        Object[] shapes = {
            new LegacyLine(), new LegacyRectangle()
        };
        // A begin and end point from a graphical editor
        int x1 = 10, y1 = 20;
        int x2 = 30, y2 = 60;
        for (int i = 0; i < shapes.length; ++i) {
            if (shapes[i].getClass().getName().equals("LegacyLine")) {
                ((LegacyLine)shapes[i]).draw(x1, y1, x2, y2);
            } else if (shapes[i].getClass().getName().equals("LegacyRectangle")) {
                ((LegacyRectangle)shapes[i]).draw(Math.min(x1, x2), Math.min(y1, y2),
                                                    Math.abs(x2 - x1), Math.abs(y2 - y1));
            }
        }
    }
}
Approach using Adapter class

• Define an interface called, say, Shape.

```java
public interface Shape {
    void draw(int x1, int y1, int x2, int y2);
}
```

• Define two classes, called, say, Line and Rectangle, which implement the interface Shape.

• The client program now works without worrying about which legacy class we have
public class Line implements Shape{
    private LegacyLine adaptee = new LegacyLine();

    public void draw(int x1, int y1, int x2, int y2)
    {
        adaptee.draw(x1, y1, x2, y2);
    }
}

public class Rectangle implements Shape{
    private LegacyRectangle adaptee = new LegacyRectangle();

    /*
    * To draw a rectangle, give two diagonally opposite points
    */
    public void draw(int x1, int y1, int x2, int y2){
        adaptee.draw(Math.min(x1, x2), Math.min(y1, y2), Math.abs(x2 - x1),
                     Math.abs(y2 - y1));
    }
}
public class AdapterDemo{
public static void main(String[] args){
    Shape[] shapes = {
        new Line(), new Rectangle()
    }
    // A begin and end point from a graphical editor
    int x1 = 10, y1 = 20;
    int x2 = 30, y2 = 60;
    for (int i = 0; i < shapes.length; ++i)
        shapes[i].draw(x1, y1, x2, y2);
}
}
A familiar example- Java Adapter class

• The Java abstract class WindowAdapter is an adapter of sorts.
• The JFrame class has a WindowListener which responds to various window events.
• The WindowListener interface specifies 7 methods which may not all be necessary.
• Often, only one or two are required.
• The WindowAdapter is a convenience class as it implements all 7 WindowListener methods with “do-nothing” (e.g. {} ) implementations.
• Thus, a WindowAdapter subclass need only to implement the methods that it needs.
A familiar example- Java Adapter class

```java
public interface Windowlistener {
    public void windowClosed(WindowEvent e);
    public void windowOpened(WindowEvent e);
    public void windowIconified(WindowEvent e);
    public void windowDeiconified(WindowEvent e);
    public void windowActivated(WindowEvent e);
    public void windowDeactivated(WindowEvent e);
    public void windowClosing(WindowEvent e);
}
```
Windowadapter class

public class WindowAdapter implements WindowListner{
    public void windowClosed(WindowEvent e)
    {
    }
    public void windowOpened(WindowEvent e)
    {
    }
    public void windowIconified(WindowEvent e)
    {
    }
    public void windowDeiconified(WindowEvent e)
    {
    }
    public void windowActivated(WindowEvent e)
    {
    }
    public void windowDeactivated(WindowEvent e)
    {
    }
    public void windowClosing(WindowEvent e)
    {
    }
}
A familiar example-Java Adapter class

```
Window
    Frame
    JFrame

<<interface>>
WindowListener
windowActivated()
windowClosed()
windowClosing()
windowDeactivated()
windowDeiconified()
windowIconified()
windowOpened()

WindowAdapter
windowActivated() {}
windowClosed() {}
windowClosing() {}
windowDeactivated() {}
windowDeiconified() {}
windowIconified() {}
windowOpened() {}
```
Part of the main method

```java
JFrame frame = new JFrame();
frame.addWindowListener(
    new WindowAdapter()
    {
        public void windowClosing(WindowEvent e)
        {
            System.exit(); // We want the application to terminate
        }
    });
```
Example of Using an Adapter in Java

- Collection classes in Java, e.g. Vector, Stack implement a method `elements()` which returns the elements of the structure.
- Older versions of Java return this as an `Enumeration`.
- Later versions return the elements as an `Iterator`.
- Both `Enumeration` and `Iterator` are interfaces.
- An adapter will be needed to convert from one interface type to another.
Enumeration vs. Iterator

- **Enumeration** interface methods:
  - `hasMoreElements()`
  - `nextElement()`

- **Iterator** interface methods:
  - `hasNext()`
  - `next()`
  - `remove()`
How do we Implement the Adapter Pattern in this Case

• Implement an \textit{EnumerationAdapter} class.
• This class will implement the \textit{Iterator} interface.
• Redefine the \textit{hasNext()} method.
• Redefine the \textit{next()} method.
• Redefine the \textit{remove()} method?
import java.util.*;

public class EnumerationIterator implements Iterator {
    private Enumeration oldEnum;
    public EnumerationIterator(Enumeration old) {
        oldEnum = old;
    }

    /*
     * Fulfils the general contract of Iterator.hasNext(), that is, return
     * True as long as there is at least one more item in the Iterator.
     */

    public boolean hasNext() {
        return oldEnum.hasMoreElements();
    }
}
public Object next() {
    return oldEnum.nextElement();
}

public void remove() {
    throw new UnsupportedOperationException("remove “
        + “not supported");
}