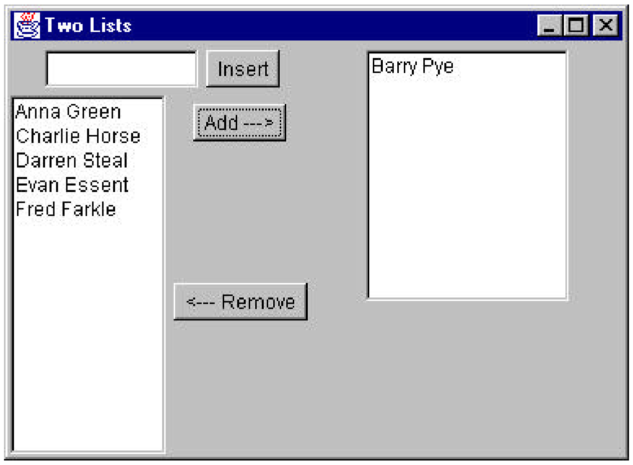
THE ADAPTER PATTERN

The Adapter pattern is used to convert the programming interface of one class into that of another. We use adapters whenever we want unrelated classes to work together in a single program. The concept of an adapter is thus pretty simple; we write a class that has the desired interface and then make it communicate with the class that has a different interface.

There are two ways to do this: by inheritance, and by object composition. In the first case, we derive a new class from the nonconforming one and add the methods we need to make the new derived class match the desired interface. The other way is to include the original class inside the new one and create the methods to translate calls within the new class. These two approaches, termed class adapters and object adapters are both fairly easy to implement in Java.

Moving Data between Lists

Let's consider a simple Java program that allows you to enter names into a list, and then select some of those names to be transferred to another list. Our initial list consists of a class roster and the second list, those who will be doing advanced work.



In this simple program, you enter names into the top entry field and click on Insert to move the names into the left-hand list box. Then, to move names to the right-hand list box, you click on them, and then click on Add. To remove a name from the right hand list box, click on it and then on Remove. This moves the name back to the left-hand list.

This is a very simple program to write in Java 1.1. It consists of a GUI creation constructor and an actionListener routine for the three buttons:

```
public void actionPerformed(ActionEvent e)
   {
      Button b = (Button)e.getSource();
      if(b == Add)
         addName();
      if(b == MoveRight)
         moveNameRight();
      if(b == MoveLeft)
         moveNameLeft();
   }
```

The button action routines are then simply

```
   private void addName()
   {
      if (txt.getText().length() > 0)
         {
         leftList.add(txt.getText());
         txt.setText("");
         }
   }
   //---------------------------------------------
   private void moveNameRight()
   {
     String sel[] = leftList.getSelectedItems();
     if (sel != null)
     {
     rightList.add(sel[0]);
     leftList.remove(sel[0]);
     }
   }
   //---------------------------------------------
   public void moveNameLeft()
   {
    String sel[] = rightList.getSelectedItems();
     if (sel != null)
     {
     leftList.add(sel[0]);
     rightList.remove(sel[0]);
     }
   }
```

This program is called TwoList.java on your CD-ROM.

Using the JFC JList Class

This is all quite straightforward, but suppose you would like to rewrite the program using the Java Foundation Classes (JFC or “Swing”). Most of the methods you use for creating and manipulating the user interface remain the same. However, the JFC JList class is markedly different than the AWT List class. In fact, because the JList class was designed to represent far more complex kinds of lists, there are virtually no methods in common between the classes:

awt List class	**JFC JList class**
add(String);	---
remove(String)	---
String[] getSelectedItems()	Object[] getSelectedValues()

Both classes have quite a number of other methods and almost none of them are closely correlated. However, since we have already written the program once, and make use of two different list boxes, writing an adapter to make the JList class look like the List class seems a sensible solution to our problem.

The JList class is a window container which has an array, vector or other ListModel class associated with it. It is this ListModel that actually contains and manipulates the data. Further, the JList class does not contain a scroll bar, but instead relies on being inserted in the viewport of the JScrollPane class. Data in the JList class and its associated ListModel are not limited to strings, but may be almost any kind of objects, as long as you provide the cell drawing routine for them. This makes it possible to have list boxes with pictures illustrating each choice in the list.

In our case, we are only going to create a class that emulates the List class, and that in this simple case, needs only the three methods we showed in the table above.

We can define the needed methods as an interface and then make sure that the class we create implements those methods:

```
public interface awtList  {
     public void add(String s);
     public void remove(String s);
     public String[] getSelectedItems()
}
```

Interfaces are important in Java, because Java does not allow multiple inheritance as C++ does. Thus, by using the *implements* keyword, the class can take on methods and the appearance of being a class of either type.

The Object Adapter

In the object adapter approach, we create a class that *contains* a JList class but which implements the methods of the awtList interface above. This is a pretty good choice here, because the outer container for a JList is not the list element at all, but the JScrollPane that encloses it.

So, our basic JawtList class looks like this:

```
public class JawtList extends JScrollPane
   implements awtList
{
   private JList listWindow;
   private JListData listContents;
//-----------------------------------------
   public JawtList(int rows)    {
      listContents = new JListData();
      listWindow = new JList(listContents);
       getViewport().add(listWindow);

   }
//-----------------------------------------
   public void add(String s)    {
      listContents.addElement(s);
   }
//-----------------------------------------
   public void remove(String s)    {
      listContents.removeElement(s);
   }
//-----------------------------------------
   public String[] getSelectedItems()    {
      Object[] obj = listWindow.getSelectedValues();
      String[] s = new String[obj.length];
      for (int i =0; i<obj.length; i++)
            s[i] = obj[i].toString();
      return s;
   }
}
```

Note, however, that the actual data handling takes place in the JlistData class. This class is derived from the AbstractListModel, which defines the following methods:

addListDataListener(l)	Add a listener for changes in the data.

removeListDataListener(l)	Remove a listener
fireContentsChanged(obj, min,max)	Call this after any change occurs between the two indexes min and max
fireIntervalAdded(obj,min,max)	Call this after any data has been added between min and max.
fireIntervalRemoved(obj, min, max)	Call this after any data has been removed between min and max.

The three *fire* methods are the communication path between the data stored in the ListModel and the actual displayed list data. Firing them causes the displayed list to be updated.

In this case, the addElement, removeElement methods are all that are needed, although you could imagine a number of other useful methods. Each time we add data to the *data* vector, we call the *fireIntervalAdded* method to tell the list display to refresh that area of the displayed list.

```
class JListData extends AbstractListModel
{
   private Vector data;
//-------------------------------------------
   public JListData()    {
      data = new Vector();
   }
//-------------------------------------------
   public void addElement(String s)
   {
      data.addElement(s);
      fireIntervalAdded(this, data.size()-1,
                   data.size());
   }
//-------------------------------------------
   public void removeElement(String s)    {
      data.removeElement(s);
      fireIntervalRemoved(this, 0, data.size());
   }
}
```

The Class Adapter

In Java, the class adapter approach isn’t all that different. If we create a class JawtClassList that is derived from JList, then we have to create a JScrollPane in our main program’s constructor:

```
 leftList = new JclassAwtList(15);
JScrollPane lsp = new JScrollPane();
pLeft.add("Center", lsp);
lsp.getViewport().add(leftList);
```

and so forth.

The class-based adapter is much the same, except that some of the methods now refer to the enclosing class instead of an encapsulated class:

```
public class JclassAwtList extends JList
   implements awtList
{
   private JListData listContents;
//-----------------------------------------
   public JclassAwtList(int rows)
   {
      listContents = new JListData();
      setModel(listContents);
      setPrototypeCellValue("Abcdefg Hijkmnop");
   }
```

There are some differences between the List and the adapted JList class that are not so easy to adapt, however. The List class constructor allows you to specify the length of the list in lines. There is no way to specify this directly in the JList class. You can compute the preferred size of the enclosing JScrollPane class based on the font size of the JList, but depending on the layout manager, this may not be honored exactly.

You will find the example class JawtClassList, called by JTwoClassList on your example CD-ROM.

There are also some differences between the class and the object adapter approaches, although they are less significant than in C++.

- The Class adapter
    - Won’t work when we want to adapt a class and all of its subclasses, since you define the class it derives from when you create it.
    - Lets the adapter change some of the adapted class’s methods but still allows the others to be used unchanged.
- An Object adapter
    - Could allow subclasses to be adapted by simply passing them in as part of a constructor.

- Requires that you specifically bring any of the adapted object's methods to the surface that you wish to make available.

Two Way Adapters

The two-way adapter is a clever concept that allows an object to be viewed by different classes as being either of type awtList or a type JList. This is most easily carried out using a class adapter, since all of the methods of the base class are automatically available to the derived class. However, this can only work if you do not override any of the base class's methods with ones that behave differently. As it happens, our JawtClassList class is an ideal two-way adapter, because the two classes have no methods in common. You can refer to the awtList methods or to the JList methods equally conveniently.

Pluggable Adapters

A pluggable adapter is one that adapts dynamically to one of several classes. Of course, the adapter can only adapt to classes it can recognize, and usually the adapter decides which class it is adapting based on differing constructors or setParameter methods.

Java has yet another way for adapters to recognize which of several classes it must adapt to: reflection. You can use reflection to discover the names of public methods and their parameters for any class. For example, for any arbitrary object you can use the *getClass()* method to obtain its class and the *getMethods()* method to obtain an array of the method names.

```
JList list = new JList();
      Method[] methods = list.getClass().getMethods();
       //print out methods
      for (int i = 0; i < methods.length; i++)       {
         System.out.println(methods[i].getName());
          //print out parameter types
         Class cl[] = methods[i].getParameterTypes();
         for(int j=0; j < cl.length; j++)
            System.out.println(cl[j].toString());
      }
```

A "method dump" like the one produced by the code shown above can generate a very large list of methods, and it is easier if you know the name of the method you are looking for and simply want to find out which arguments that method requires. From that method signature, you can then deduce the adapting you need to carry out.

However, since Java is a strongly typed language, it is more likely that you would simply invoke the adapter using one of several constructors, where each constructor is tailored for a specific class that needs adapting.

Adapters in Java

In a broad sense, there are already a number of adapters built into the Java language. In this case, the Java adapters serve to simplify an unnecessarily complicated event interface. One of the most commonly used of these Java adapters is the WindowAdapter class.

One of the inconveniences of Java is that windows do not close automatically when you click on the Close button or window Exit menu item. The general solution to this problem is to have your main Frame window implement the WindowListener interface, leaving all of the Window events empty except for windowClosing.

```
public void mainFrame extends Frame
       implements WindowListener
{
       public void mainFrame()    {
       addWindowListener(this);   //frame listens
                                         //for window events
       }

   public void windowClosing(WindowEvent wEvt)    {
    System.exit(0);   //exit on System exit box clicked
   }
   public void windowClosed(WindowEvent wEvt){}
   public void windowOpened(WindowEvent wEvt){}
   public void windowIconified(WindowEvent wEvt){}
   public void windowDeiconified(WindowEvent wEvt){}
   public void windowActivated(WindowEvent wEvt){}
   public void windowDeactivated(WindowEvent wEvt){}
}
```

As you can see, this is awkward and hard to read. The WindowAdapter class is provided to simplify this procedure. This class contains empty implementations of all seven of the above WindowEvents. You need then only override the windowClosing event and insert the appropriate exit code.

One such simple program is shown below:

```
//illustrates using the WindowAdapter class
public class Closer extends Frame {
  public Closer()   {
     WindAp windap = new WindAp();
     addWindowListener(windap);
     setSize(new Dimension(100,100));
     setVisible(true);
  }
  static public void main(String argv[])   {
     new Closer();
  }
}
//make an extended window adapter which
//closes the frame when the closing event is received
class WindAp extends WindowAdapter {
   public void windowClosing(WindowEvent e)    {
      System.exit(0);
   }
}
```

You can, however, make a much more compact, but less readable version of the same code by using an anonymous inner class:

```
//create window listener for window close click
      addWindowListener(new WindowAdapter()
       {
           public void windowClosing(WindowEvent e)
                    {System.exit(0);}
       });
```

Adapters like these are common in Java when a simple class can be used to encapsulate a number of events. They include ComponentAdapter, ContainerAdapter, FocusAdapter, KeyAdapter, MouseAdapter, and MouseMotionAdapter.