CHAPTER 20

Clipboard and Data Transfer

Nearly every modern windowing system includes support for a clipboard—a rendezvous point for data of various kinds; data can be transferred to and retrieved from the clipboard.

Although initial versions of the AWT did not explicitly support clipboard and data transfer, the current AWT comes complete with data transfer and clipboard support. In fact, the AWT provides access to two kinds of clipboards: the *system* clipboard, and *local* clipboards. As you might suspect, manipulation of the system clipboard actually manipulates the native windowing system clipboard. We mention this because copying data onto the system clipboard in a Java applet or application makes it available to other programs, not just to other AWT components. Alternatively, applets can also create, for their own internal use, local clipboards that do not involve the native windowing clipboard.

In this chapter, we'll take a look at the fundamental concepts behind data transfer and the clipboard, followed by examples that place data on both the system clipboard and local clipboards and retrieve data from the clipboards in some fashion. As of this writing, the `java.awt.datatransfer` package only provides explicit support for placing Java strings onto the clipboard, but we'll take a look at the steps involved in placing other kinds of data, such as images and custom AWT components, on the clipboard.

The java.awt.datatransfer Package

The clipboard and data transfer mechanism are implemented with a handful of classes and interfaces, all of which reside in the `java.awt.datatransfer` package. The classes and interfaces from `java.awt.datatransfer` are listed in Table 20-1.

Table 20-1 **java.awt.datatransfer Classes and Interfaces**

Class/Interface	Class or Interface	Purpose
`Clipboard`	Class	`Transferable`s can be copied to and retrieved from the clipboard
`ClipboardOwner`	Interface	An interface for classes that copy data to the clipboard
`DataFlavor`	Class	Data flavors (formats) supported by a transferable
`StringSelection`	Class	A transferable that encapsulates textual data
`Transferable`	Interface	Interface for items that can be placed on the clipboard
`UnsupportedFlavorException`	Class	Thrown by a transferable when a requested data flavor is not supported

The concepts behind the clipboard and the data transfer mechanism are quite simple. First of all, only a `ClipboardOwner` can copy data onto the clipboard. By copying data onto the clipboard, the clipboard owner becomes the current owner of the data on the clipboard. Furthermore, only one type of object can be copied to or retrieved from the clipboard—a *transferable* object. Transferables encapsulate data of some sort and are able to provide their data in one or more different *data flavors*. For instance, a transferable that contains an image may choose to provide the image in two different flavors—either as a reference to an instance of `java.awt.Image` or as an array of pixels.

The Clipboard Class

`java.awt.datatransfer.Clipboard` is an extremely simple class. There are only three things you can do to a clipboard, other than construct one, as you can see from Table 20-2: get the name of the clipboard, set the contents of the clipboard, and retrieve the current contents of the clipboard. Notice that there is

no `setName` method—setting the name of the clipboard is done at construction time. Furthermore, you can either create a local clipboard via the `Clipboard` constructor or you can obtain a reference to the system clipboard by invoking `Toolkit.getSystemClipboard()`.

Table 20-2 `java.awt.datatransfer.Clipboard` **Public Methods**

Method	Description
`String getName()`	Returns the name of the clipboard
`void setContents(Transferable, ClipboardOwner)`	Sets the contents of the clipboard to the transferable passed in; also sets the owner of the clipboard
`Transferable getContents(Object)`	Returns the contents of the clipboard; the `Object` parameter is the requestor of the clipboard contents

Copying Data to and Retrieving Data from a Clipboard

Before we launch into our first example, let's enumerate the steps involved in copying data to a clipboard and subsequently retrieving it.

To Copy Data to a Clipboard:

- Either instantiate a clipboard or obtain a reference to an existing clipboard.
- Wrap the data in a transferable object (which could involve implementing an extension of `Transferable`).
- Copy the transferable to the clipboard, specifying both the transferable and the owner of the clipboard.

To Retrieve Data from the Clipboard:

- Obtain a reference to the clipboard that contains the data you are interested in.
- (Optional.) Determine if the clipboard contents (a transferable) provides its data in a palatable flavor.
- If the transferable currently on the clipboard provides its data in a flavor acceptable to you, ask the transferable to produce its data in the flavor in question.

The ClipboardOwner Class

Depending upon the native facilities available for data transfer, the contents of the clipboard may not actually be copied to the clipboard until the data is requested from the clipboard. Such a scenario is known as a *lazy data mode* and is the reason

for the `ClipboardOwner` interface. The clipboard owner is the object that puts the data on the clipboard, and it must make sure that the data placed on the clipboard is accessible up until the time its `lostOwnership` method is invoked. `lostOwnership()` is the only method defined by the `ClipboardOwner` interface and is invoked when another object places data on the clipboard. If, for some reason, a clipboard owner needs to release the resources associated with an item it placed on the clipboard, it is only safe to do so after the `lostOwnership` method is invoked. In practice, the `lostOwnership` method is almost always implemented as a no-op.

Ok, enough discussion about data transfer concepts and the steps involved in using the clipboard. Let's get on with some examples.

The System Clipboard

As you can see in Figure 20-1, our first applet contains a textfield, a textarea, and two buttons. After text is typed into the textfield, activating the Copy To System Clipboard button places the textfield's text onto the system clipboard. Subsequently, activating the "Paste From System Clipboard" button retrieves the text currently on the clipboard and places it in the textarea.

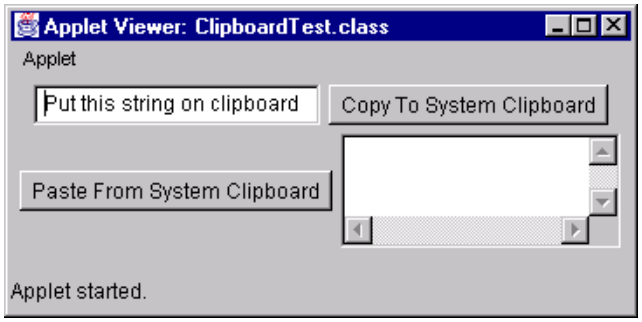


Figure 20-1 Copying a String to the System Clipboard
Activating the Copy To System Clipboard button copies the text from the textfield and places it on the system clipboard.

Before listing the applet in its entirety, let's take a look at the code that implements the steps outlined previously for placing data on and retrieving data from a clipboard.

First, we obtain a reference to the system clipboard:

```
public class ClipboardTest extends    Applet
                           implements ClipboardOwner {
private Clipboard clipboard;
private TextField copyFrom;
private TextArea  copyTo;
private Button    copy, paste;

public void init() {
   // Obtain a reference to the system clipboard
   clipboard = getToolkit().getSystemClipboard();
          ...
   }
}
```

As we noted earlier, a reference to the system clipboard is obtained by invoking the `getSystemClipboard()`[1] static method supplied by the `Toolkit` class. Next, we wrap the text contained in the textfield in a transferable and place the transferable on the system clipboard:

```
class CopyListener implements ActionListener {
   public void actionPerformed(ActionEvent event) {

      // Wrap the data in a transferable object
      StringSelection contents =
                   new StringSelection(copyFrom.getText());

      // Place the transferable onto the clipboard
      clipboard.setContents(contents, ClipboardTest.this);
   }
}
```

`CopyListener` is the action listener for the "Copy To System Clipboard" button. Note that the transferable is an instance of `StringSelection`, which is provided by the AWT for wrapping textual data.

Finally, we retrieve the text from the system clipboard when the Paste From System Clipboard button is activated:

```
class PasteListener implements ActionListener {
   public void actionPerformed(ActionEvent event) {
      Transferable contents = clipboard.getContents(this);

      // Determine if data is available in string flavor
      if(contents != null &&
            contents.isDataFlavorSupported(
                        DataFlavor.stringFlavor)) {
         try {
```

1. Note that `Toolkit.getSystemClipboard()` is subject to security restrictions.

```
                String string;
                // Have contents cough up string
                string = (String) contents.getTransferData(
                                    DataFlavor.stringFlavor);
                copyTo.append(string);
            }
            catch(Exception e) {
                e.printStackTrace();
            }
        }
    }
}
```

`PasteListener` is the action listener for the Paste From System Clipboard button, and the first thing its `actionPerformed` method does is to obtain the contents of the clipboard via the clipboard's `getContents` method.

`Clipboard.getContents()` requires us to pass a reference to the object that is requesting the data (the `PasteListener`). Once we have a reference to the transferable currently on the clipboard, we ask the transferable if the data it contains is available in the particular data flavor we are interested in. If there was indeed a transferable on the clipboard and it supports `DataFlavor.stringFlavor`, then we invoke the clipboard's `getTransferData()`, telling the transferable the type of data flavor we'd like returned. Finally, we append the string obtained from the transferable to the text area.

The fruits of our labor are shown in Figure 20-2, and the entire applet is listed in Example 20-1.

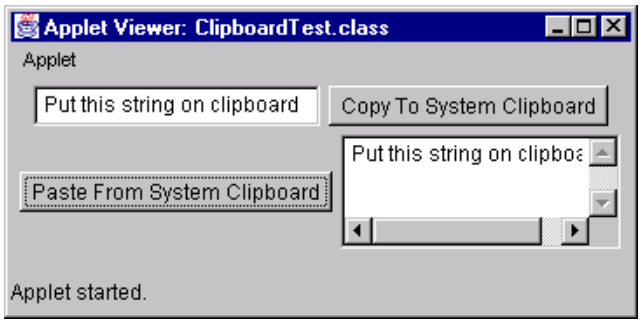
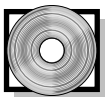


Figure 20-2 Retrieving a String from the System Clipboard Activating the Paste From System Clipboard button retrieves the text from the system clipboard and appends it to the textarea.



Example 20-1 ClipboardTest Applet

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
import java.awt.datatransfer.*;

public class ClipboardTest extends    Applet
                           implements ClipboardOwner {
    private Clipboard clipboard;
    private TextField copyFrom;
    private TextArea  copyTo;
    private Button    copy, paste;

    public void init() {
        // Obtain a reference to the system clipboard
        clipboard = getToolkit().getSystemClipboard();

        copyFrom  = new TextField(20);
        copyTo    = new TextArea(3, 20);
        copy      = new Button("Copy To System Clipboard");
        paste     = new Button("Paste From System Clipboard");

        add(copyFrom);
        add(copy);
        add(paste);
        add(copyTo);

        copy.addActionListener (new CopyListener());
        paste.addActionListener(new PasteListener());
    }
    class CopyListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            // Wrap the data in a transferable object
            StringSelection contents =
                      new StringSelection(copyFrom.getText());

            // Place the transferable onto the clipboard
            clipboard.setContents(contents, ClipboardTest.this);
        }
    }
    class PasteListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            Transferable contents = clipboard.getContents(this);

            // Determine if data is available in string flavor
            if(contents != null &&
               contents.isDataFlavorSupported(
                         DataFlavor.stringFlavor)) {
                try {
                    String string;
```

```
                    // Have contents cough up string
                    string = (String) contents.getTransferData(
                                    DataFlavor.stringFlavor);
                    copyTo.append(string);
                }
                catch(Exception e) {
                    e.printStackTrace();
                }
            }
        }
    }
    public void lostOwnership(Clipboard    clip,
                              Transferable transferable) {
        System.out.println("Lost ownership");
    }
}
```

Pasting from the System Clipboard to Other Applications

As we alluded to earlier, copying data to the system clipboard makes it available to other programs in addition to other AWT components. After we copied the text onto the system clipboard, we opened a document in FrameMaker® and did a paste operation. The result is shown in Figure 20-3.

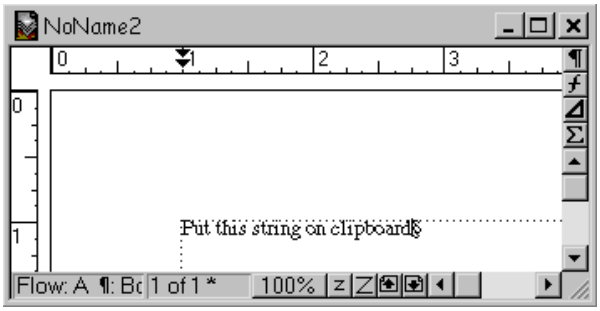


Figure 20-3 Pasting Clipboard Contents into Another Application
Text placed on the clipboard from an applet is pasted into a FrameMaker document.

Local Clipboards

In Example 20-1 on page 675, we manipulated the system clipboard. You can also create local clipboards by invoking the lone `Clipboard` constructor that takes a string as an argument. The string specifies the name of the clipboard, so you can

keep track of several clipboards if need be. As we saw in Table 20-2 on page 671, the `Clipboard` class comes with a `getName` method so that you can determine which clipboard you currently have in hand.

Data Transfer Mechanism

Now that we've introduced the basics of data transfer and manipulating a clipboard and provided a simple example of cutting and pasting text to/from the system clipboard, let's take a closer look at the data transfer mechanism.

Data Flavors

No, you cannot taste data; flavor, in this case, can be thought of as a synonym for format. One data flavor for an image, for instance, might be a `java.awt.Image`; another flavor for an image might be an array of pixels representing the image in question.

Data flavors fall into two general categories:

- A Java class
- MIME (Multipurpose Internet Mail Extensions) Type Representation[2]

All this means is that you can either construct a `DataFlavor` by specifying the particular Java class that the data flavor represents, or you can construct a data flavor by specifying a string that represents a data type. If you choose to construct a data flavor with a string representing the data type, the string should conform to a standard MIME type. If you construct a `DataFlavor` by specifying a Java class, it will be assigned the MIME type "application/x-javaserializedobject". For instance, if you look at the source for `java.awt.datatransfer.DataFlavor`, you will see that the `DataFlavor` class provides two constructors:

```
public DataFlavor(Class representationClass,
                  String humanPresentableName)

public DataFlavor(String mimeType, String humanPresentableName)
```

Regardless of which constructor you use, you must provide a human presentable name for your particular data flavor, but you probably figured that out on your own!

The `DataFlavor` class contains two static public instances of `DataFlavor`, each representing a different flavor of text. One flavor is represented by `java.lang.String`; the other is represented by a MIME type for plain text:

2. If you want to read the gory details concerning MIME types, check out http://206.21.31.20/notes/rfcs1/30ee_1e2.htm.

```
static { // This is from the DataFlavor class
    try {
     stringFlavor =
           new DataFlavor(Class.forName("java.lang.String"),
                          "Unicode String");
     plainTextFlavor =
              new DataFlavor("text/plain; charset=unicode",
                          "Plain Text");
...
```

So, don't get hung up on MIME types. MIME types are simply a standard way to describe certain flavors of data, and since there is already a standard established, it makes perfect sense to use it instead of defining another standard. Table 20-3 lists the public methods provided by the `DataFlavor` class.

Table 20-3 `java.awt.datatransfer.DataFlavor` **Public Methods**

Method	Description
`Object clone() throws CloneNotSupportedException`	Returns a clone of the data flavor
`String getParameter(String)`	Returns the value of the named parameter
`String getPrimaryType()`	Returns the primary MIME type
`String getSubType()`	Returns the MIME subtype
`String getMimeType()`	Returns MIME type for data flavor
`Class getRepresentationClass()`	Returns the Java class the data flavor represents
`String getHumanPresentableName()`	Returns the human presentable name of the data flavor
`void setHumanPresentableName( String)`	Sets the human presentable name of the data flavor
`boolean equals(DataFlavor)`	Determines if another data flavor is equal to the data flavor on whose behalf the method is called
`boolean equals(MimeType)`	Determines if the data flavor is equal to the specified MIME type
`boolean equals(Object)`	Determines if the data flavor is equal to the specified object
`boolean equals(String)`	Determines if another data flavor is equal to the data flavor on whose behalf the method is called

Table 20-3 `java.awt.datatransfer.DataFlavor` **Public Methods (Continued)**

Method	Description
`boolean isMimeTypeEqual(DataFlavor)`	Determines if the data flavor has a MIME type equivalent to the MIME type of the data flavor passed in
`boolean isMimeTypeEqual(String)`	Determines if the data flavor has a MIME type equivalent to the `String` passed in
`boolean isMimeTypeEqual(DataFlavor)`	Convenience method implemented as: `return isMimeTypeEqual( flavor.getMimeType())`
`boolean isMimeTypeSerializedObject()`	Signifies whether the data flavor represents a serializable object
`boolean isRepresentationClassInput-Stream()`	Signifies whether the data flavor represents an input stream
`String normalizeMimeTypeParameter( String, String)`	Allows `DataFlavor` subclasses to modify how MIME types are normalized
`String normalizeMimeTypeParameter( String, String)`	Allows `DataFlavor` subclasses to handle special parameters
`void readObject( ObjectInputStream)`	Restores a serialized data flavor
`void writeObject( ObjectOutputStream)`	Serializes the data flavor

Transferables and Data Flavors

As we mentioned previously, there's only one type of object that can be placed on or retrieved from a clipboard—a `Transferable`, which is an interface that defines the three methods listed in Table 20-4.

A transferable then, is simply a wrapper around some piece of data. Since a given type of data can be packaged in different flavors, a transferable is a data wrapper that manages the flavors of the particular type of data it encapsulates.

Notice that `getTransferData()` can throw either an `UnsupportedFlavorException` if the requested data flavor is not supported or an `IOException` if an object representing the given data flavor cannot be created. Of course, you can avoid having an `UnsupportedFlavorException`

Table 20-4 `java.awt.datatransfer.Transferable` **Interface Public Methods**

Method	Intent
`DataFlavor[] getTransferDataFlavors()`	Provides an array of data flavors supported by the transferable
`boolean isDataFlavorSupported(DataFlavor)`	Determine if a particular data flavor is supported by the transferable
`Object getTransferData(DataFlavor) throws UnsupportedFlavorException, IOException;`	Returns an `Object` representing the specified flavor

thrown by calling `isDataFlavorSupported()` to find out if a particular data flavor is supported before you invoke `getTransferData()`, as we did in Example 20-1 on page 675.

StringSelection

The `java.awt.datatransfer` package comes with one implementation of `Transferable` that encapsulates text: `StringSelection`. `StringSelection` can cough up its string in one of two different flavors: as a `java.lang.String` or as plain text. As you might guess, `StringSelection` uses the two (public) static instances of `DataFlavor` provided by the `DataFlavor` class for representing its string—see "Data Flavors" on page 677.

In Example 20-1 on page 675, we asked an instance of `StringSelection` obtained from the clipboard for its text in the form of a `java.lang.String`:

```
if(contents != null &&
        contents.isDataFlavorSupported(DataFlavor.stringFlavor))
{
       try {
           String string;

           // Have contents cough up string
           string = (String) contents.getTransferData(
                                    DataFlavor.stringFlavor);
       ...
```

We could also have asked the `StringSelection` object for its text represented as plain text:

```
class PasteListener implements ActionListener {
    public void actionPerformed(ActionEvent event) {
        Transferable contents = clipboard.getContents(this);
        StringBuffer sb = new StringBuffer();

        // Determine if data is available in plain text flavor
        if(contents != null &&
```

```
            contents.isDataFlavorSupported(
                        DataFlavor.plainTextFlavor)) {
            try {
                int i;
                StringReader s = (StringReader)
                            contents.getTransferData(
                            DataFlavor.plainTextFlavor);

                while( (i = s.read()) != -1) {
                    sb.append((char)i);
                }
            }
            catch(Exception e) {
                e.printStackTrace();
            }
            catch(java.util.mime.MimeTypeParseException e) {
                e.printStackTrace();
            }
            copyTo.setText(sb.toString());
        }
    }
}
```

`StringSelection.getTransferData()` returns a `StringReader` when asked for its data in plain text flavor.

Copying Images to a Clipboard

As we have seen, manipulating clipboards is relatively straightforward. Perhaps the most difficult aspect of the clipboard / data transfer mechanism is copying data other than text to a clipboard, and even that is not rocket science, especially once you get the hang of it.

ImageSelection—A Transferable for Encapsulating Images

The first order of business is to develop a class that implements `Transferable` and encapsulates an image. Our first version of such a class will offer a lone data flavor for representing the image—`java.awt.Image`. A little later on, we'll add another data flavor for good measure.

We'll follow the naming convention established by `StringSelection` and name our class `ImageSelection`. Additionally, as with `StringSelection`, we'll also implement the `ClipboardOwner` interface[3] as a convenience, so that the contents and the owner of the clipboard can be specified as the same object when the contents of the clipboard are set.

```
public class ImageSelection implements Transferable,
```

3. See The ClipboardOwner Class on page 671. It is often the case that classes implementing `Transferable` also implement the `ClipboardOwner` interface.

```
                                                ClipboardOwner {
    static public DataFlavor ImageFlavor;

    private DataFlavor[] flavors = {ImageFlavor};
    private Image image;
    ...
```

`ImageSelection` contains a `static public` instance of `DataFlavor` that clients can use to specify the data flavor in which they'd like the image to be produced. Since `ImageSelection` has only one data flavor for its image, it maintains an array of data flavors that has only one entry. The array, of course, will be returned from the `getTransferDataFlavors` method. Finally, `ImageSelection` maintains a reference to the image that it currently represents.

`ImageSelection` implements a static block that creates the `ImageFlavor` instance:

```
static {
    try {
        ImageFlavor = new DataFlavor(
                  Class.forName("java.awt.Image"),"AWT Image");
    }
    catch(ClassNotFoundException e) {
        e.printStackTrace();
    }
}
```

Notice that the data flavor is constructed with a Java class instead of a MIME type, and with a human presentable name of `"AWT Image"`.

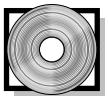
`ImageSelection` provides a lone constructor that takes a reference to an image, and implements the three methods required by the `Transferable` interface:

```
public ImageSelection(Image image) {
    this.image = image;
}
public synchronized DataFlavor[] getTransferDataFlavors() {
    return flavors;
}
public boolean isDataFlavorSupported(DataFlavor flavor) {
    return flavor.equals(ImageFlavor);
}
public synchronized Object getTransferData(DataFlavor flavor)
                  throws UnsupportedFlavorException, IOException {
    if(flavor.equals(ImageFlavor)) {
        return image;
    }
    else {
        throw new UnsupportedFlavorException(flavor);
    }
}
public void lostOwnership(Clipboard c, Transferable t) {
}
```

If the data flavor requested in `getTransferData()` is equal to the `ImageFlavor` instance, the image is returned; otherwise, an `UnsupportedFlavorException` is thrown. Also, as is typically the case, since `ImageSelection` never releases the resources associated with the image it encapsulates, its `lostOwnership` method is implemented as a no-op.

That's all there is to implementing a transferable that encapsulates an image. For completeness, `ImageSelection` is shown in its entirety in Example 20-2.

Example 20-2 ImageSelection Class Listing



```
import java.awt.*;
import java.awt.datatransfer.*;
import java.io.*;

public class ImageSelection implements Transferable,
                                       ClipboardOwner {
    static public DataFlavor ImageFlavor;

    private DataFlavor[] flavors = {ImageFlavor};
    private Image image;

    static {
        try {
            ImageFlavor = new DataFlavor(
                Class.forName("java.awt.Image"), "AWT Image");
        }
        catch(ClassNotFoundException e) {
            e.printStackTrace();
        }
        catch(java.util.mime.MimeTypeParseException e) {
            e.printStackTrace();
        }
    }
    public ImageSelection(Image image) {
        this.image = image;
    }
    public synchronized DataFlavor[] getTransferDataFlavors() {
        return flavors;
    }
    public boolean isDataFlavorSupported(DataFlavor flavor) {
        return flavor.equals(ImageFlavor);
    }
    public synchronized Object getTransferData(DataFlavor flavor)
        throws UnsupportedFlavorException, IOException {
        if(flavor.equals(ImageFlavor)) {
            return image;
        }
        else {
            throw new UnsupportedFlavorException(flavor);
        }
    }
    public void lostOwnership(Clipboard c, Transferable t) {
    }
}
```

Now let's put our `ImageSelection` class to use by copying an image to a local clipboard and subsequently retrieving it.

Using the ImageSelection Class

As you can see from Figure 20-4, our applet contains two image canvases, one of which initially contains an image, along with a copy button and a paste button. The Copy button, of course, copies the image to a clipboard—this time, we'll copy the image to a local clipboard instead of the system clipboard. Activating the Paste button retrieves the image from the local clipboard and puts it into the right-hand image canvas.

Figure 20-4 Copying and Retrieving an Image to/from a Clipboard
The top picture shows the applet in its initial state. After activating the copy and paste buttons, the applet looks like the bottom picture—the image has been copied to a clipboard and then pasted from the clipboard into the right-hand image canvas.

Our applet creates a local clipboard, with the name "image clipboard," along with the various AWT components used in the applet:

```
public class ClipboardTest2 extends    Applet
                          implements ClipboardOwner {
    private Clipboard   clipboard;
    private ImageCanvas copyFrom = new ImageCanvas();
    private ImageCanvas copyTo   = new ImageCanvas();
    private Button copy  = new Button("Copy");
    private Button paste = new Button("Paste");

    public void init() {
        clipboard = new Clipboard("image clipboard");

        copyFrom.setImage(getImage(getCodeBase(),"skelly.gif"));
        add(copyFrom);
        add(copyTo);
        add(copy);
        add(paste);
        copy.addActionListener (new CopyListener());
        paste.addActionListener(new PasteListener());
    }
...
}
```

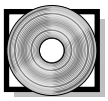
Next, event listeners are implemented for the two buttons:

```
class CopyListener implements ActionListener {
    public void actionPerformed(ActionEvent event) {
        ImageSelection contents =
                      new ImageSelection(copyFrom.getImage());

        clipboard.setContents(contents, ClipboardTest2.this);
    }
}
class PasteListener implements ActionListener {
    public void actionPerformed(ActionEvent event) {
        Transferable contents = clipboard.getContents(this);

        if(contents != null &&
            contents.isDataFlavorSupported(
                                ImageSelection.ImageFlavor)) {
            try {
                Image image;
                image = (Image) contents.getTransferData(
                                ImageSelection.ImageFlavor);
                copyTo.setImage(image);
            }
            catch(Exception e) {
                e.printStackTrace();
            }
            catch(java.util.mime.MimeTypeParseException e) {
                e.printStackTrace();
            }
        }
    }
}
```

`CopyListener.actionPerformed()` creates an instance of `ImageSelection`, which it places on the clipboard. Notice that the clipboard owner (the second argument to `setContents()`) is specified as `ClipboardTest2.this` because `CopyListener` is an inner class of `ClipboardTest2` and it is `ClipboardTest2`, not `CopyListener`, that implements the `ClipboardOwner` interface.

`PasteListener.actionPerformed()`, after obtaining the contents of the clipboard, checks to see if the transferable currently on the clipboard supports the `ImageSelection.ImageFlavor` data flavor. Obviously, we are engaging in a bit of paranoia here—we know who put the data on the clipboard, and so we know that `ImageSelection.ImageFlavor` is supported, but such tight coupling between the object that places data on the clipboard and objects that retrieve the data is not always the case, so we make the check for the sake of illustration. Finally, we retrieve the image from the transferable and place it in the right-hand image canvas.

The entire applet is shown in Example 20-3. Note that the implementation of `ImageCanvas` has no bearing on the concepts we are stressing but is included for completeness.



Example 20-3 `ClipboardTest2 Applet`

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
import java.awt.datatransfer.*;

public class ClipboardTest2 extends    Applet
                            implements ClipboardOwner {
    private Clipboard   clipboard;
    private ImageCanvas copyFrom = new ImageCanvas();
    private ImageCanvas copyTo   = new ImageCanvas();
    private Button copy  = new Button("Copy");
    private Button paste = new Button("Paste");

    public void init() {
        clipboard = new Clipboard("image clipboard");

        copyFrom.setImage(getImage(getCodeBase(),"skelly.gif"));
        add(copyFrom);
        add(copyTo);
        add(copy);
        add(paste);

        copy.addActionListener (new CopyListener());
        paste.addActionListener(new PasteListener());
    }
```

```
    class CopyListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            ImageSelection contents =
                       new ImageSelection(copyFrom.getImage());

            clipboard.setContents(contents, ClipboardTest2.this);
        }
    }
    class PasteListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            Transferable contents = clipboard.getContents(this);

            if(contents != null &&
               contents.isDataFlavorSupported(
                   ImageSelection.ImageFlavor)) {
               try {
                   Image image;
                   image = (Image) contents.getTransferData(
                           ImageSelection.ImageFlavor);
                   copyTo.setImage(image);
               }
               catch(Exception e) {
                   e.printStackTrace();
               }
            }
        }
    }
    public void lostOwnership(Clipboard    clip,
                              Transferable transferable) {
        System.out.println("Lost ownership");
    }
}
class ImageCanvas extends Panel {
    private Image image;

    public ImageCanvas() {
        this(null);
    }
    public ImageCanvas(Image image) {
        if(image != null)
            setImage(image);
    }
    public void paint(Graphics g) {
        g.setColor(Color.lightGray);

        g.draw3DRect(0,0,getSize().width-1,
                         getSize().height-1,true);

        if(image != null) {
         g.drawImage(image, 1, 1, this);
        }
    }
```

```
        public void update(Graphics g) {
            paint(g);
        }
        public void setImage(Image image) {
            this.image = image;
            try {
                MediaTracker tracker = new MediaTracker(this);
                tracker.addImage(image, 0);
             tracker.waitForID(0);
            }
            catch(Exception e) { e.printStackTrace(); }

            if(isShowing()) {
                repaint();
            }
        }
        public Image getImage() {
            return image;
        }
        public Dimension getPreferredSize() {
            return new Dimension(100,100);
        }
    }
```

Adding an Additional Flavor

Now we'll take our `ImageSelection` class and add support for another data flavor.[4] Our additional flavor will take the form of an array of bits that represents the image. The `ImageSelection2` class contains two `static public DataFlavor` objects, which are placed in the `flavors` array. Additionally, the static block now constructs both flavors:

```
    public class ImageSelection2 implements Transferable,
                                            ClipboardOwner {
        static public DataFlavor ImageFlavor;
        static public DataFlavor ImageArrayFlavor;
        private DataFlavor[] flavors = {ImageFlavor,
                                                ImageArrayFlavor};
        private Image image;
        private int   width, height;

        static {
            try {
                ImageFlavor = new DataFlavor(
                    Class.forName("java.awt.Image"), "AWT Image");

                ImageArrayFlavor = new DataFlavor("image/gif",
                                                   "GIF Image");
            }
```

4. Note that we have resisted bad jokes about Baskin-Robbins.

```
            catch(ClassNotFoundException e) {
                e.printStackTrace();
            }
            catch(java.util.mime.MimeTypeParseException e) {
                e.printStackTrace();
            }
        }
    ...
    }
```

The `ImageSelection2` constructor is passed the width and height of the image, which are needed to return an array of pixels representing the image. Once again, the methods defined in the `Transferable` interface are implemented.

```
public ImageSelection2(Image image, int width, int height) {
    this.image  = image;
    this.width  = width;
    this.height = height;
}
public synchronized DataFlavor[] getTransferDataFlavors() {
    return flavors;
}
public boolean isDataFlavorSupported(DataFlavor flavor) {
    return flavor.equals(ImageFlavor) ||
               flavor.equals(ImageArrayFlavor);
}
public synchronized Object getTransferData(
                                       DataFlavor flavor)
    throws UnsupportedFlavorException, IOException {
    if(flavor.equals(ImageFlavor)) {
        return image;
    }
    else if(flavor.equals(ImageArrayFlavor)) {
        return imageToArray();
    }
    else
        throw new UnsupportedFlavorException(flavor);
}
```

By now, the methods listed above should need little commentary. Notice that `ImageSelection2.getTransferData()` returns the image as either a `java.awt.Image` or an array of pixels, depending upon the requested data flavor. Of course, the only mystery left to uncover involves the inner workings of the `imageToArray` method, which employs an instance of `PixelGrabber` for obtaining the pixels associated with the image:[5]

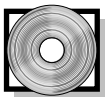
```
private int[] imageToArray() {
    int[] pixels = new int[width*height];
```

5. See “Grabbing Pixels” on page 190 for more information on `PixelGrabber` and image manipulation.

```
        PixelGrabber pg = new PixelGrabber(image,0,0,
                                width,height,pixels,0,width);
        try { pg.grabPixels(); }
        catch(InterruptedException e) { e.printStackTrace(); }
        return pixels;
    }
```

`ImageSelection2` is listed in Example 20-4.

Example 20-4 `ImageSelection2` Class Listing



```
    import java.awt.*;
    import java.awt.image.*;
    import java.awt.datatransfer.*;
    import java.io.*;

    public class ImageSelection2 implements Transferable,
                                            ClipboardOwner {
        static public DataFlavor ImageFlavor;
        static public DataFlavor ImageArrayFlavor;

        private DataFlavor[] flavors = {ImageFlavor,
                                        ImageArrayFlavor};
        private Image image;
        private int   width, height;

        static {
            try {
                ImageFlavor = new DataFlavor(
                    Class.forName("java.awt.Image"),
                    "AWT Image");

                ImageArrayFlavor = new DataFlavor("image/gif",
                                                  "GIF Image");
            }
            catch(ClassNotFoundException e) {
                e.printStackTrace();
            }
            catch(java.util.mime.MimeTypeParseException e) {
                e.printStackTrace();
            }
        }
        public ImageSelection2(Image image, int width, int height) {
            this.image  = image;
            this.width  = width;
            this.height = height;
        }
        public synchronized DataFlavor[] getTransferDataFlavors() {
            return flavors;
        }
        public boolean isDataFlavorSupported(DataFlavor flavor) {
            return flavor.equals(ImageFlavor) ||
                   flavor.equals(ImageArrayFlavor);
```

```
    }
    public synchronized Object getTransferData(
                                        DataFlavor flavor)
        throws UnsupportedFlavorException, IOException {
        if(flavor.equals(ImageFlavor)) {
            return image;
        }
        else if(flavor.equals(ImageArrayFlavor)) {
            return imageToArray();
        }
        else
            throw new UnsupportedFlavorException(flavor);
    }
    public void lostOwnership(Clipboard c, Transferable t) {
    }
    private int[] imageToArray() {
        int[] pixels = new int[width*height];
        PixelGrabber pg = new PixelGrabber(image,0,0,
                             width,height,pixels,0,width);

        try { pg.grabPixels(); }
        catch(InterruptedException e) { e.printStackTrace(); }

        return pixels;
    }
}
```

All that's left is to modify our applet to ask for the new data flavor when retrieving the image from the clipboard:

```
class PasteListener implements ActionListener {
    public void actionPerformed(ActionEvent event) {
        Transferable contents = clipboard.getContents(this);

        if(contents != null) {
            try {
                int[] array = (int[])
                          contents.getTransferData(
                           ImageSelection2.ImageArrayFlavor);

                copyTo.setImage(waveThis(array,width,height));
            }
            catch(Exception e) {
                e.printStackTrace();
            }
        }
    }
}
```

We obtain the array of bits from the `ImageSelection2` instance by asking for the `ImageArrayFlavor`, and then we pass the array to the `waveThis` method, which, purely for entertainment purposes, runs the array through a sine wave

filter and returns a new image, which we set as the image for the right-hand image canvas. You can see the results of our shenanigans in Figure 20-5. Since the `waveThis` method has no bearing on clipboard and data transfer, we won't bother to list it; the code, of course, is on the CD in the back of the book.[6]

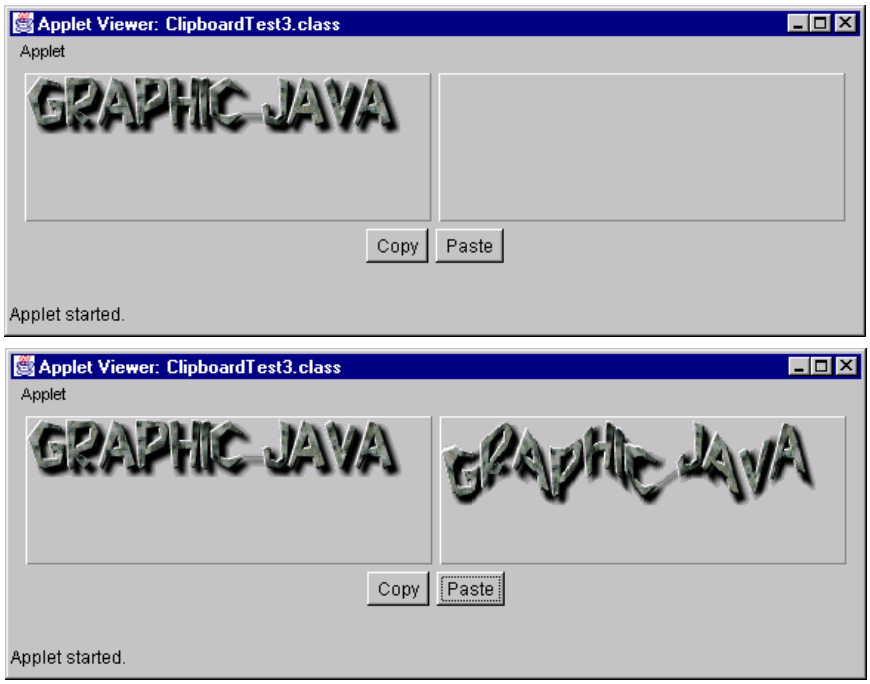


Figure 20-5 Retrieving an Image off the Clipboard as an Array of Bits
The picture on the left is copied to the clipboard and retrieved as an array of bits. The array of bits is run through a sine wave function, the result of which is displayed in the right-hand image canvas.

6. A more extensive `WaveFilter` class is discussed in "Wave Filter" on page 160.

Transferring Custom AWT Components

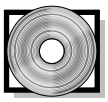
Our final example, at the risk of beating data transfer to death, is to implement a transferable that encapsulates a custom component.

Note: A bug under the 1.2 release of the AWT causes the following applet to throw an exception under Windows NT (it works fine under Solaris).

A Transferable for Encapsulating a Custom Component

Our custom component will be a very simplistic image button, whose implementation we won't bother to discuss. Instead, we will simply show the implementation of `ImageButtonSelection` and the applet that exercises it. The complete implementation of `ImageButtonSelection` is listed in Example 20-5.

Example 20-5 `ImageButtonSelection` Class Listing



```
import java.awt.*;
import java.awt.datatransfer.*;
import java.io.*;

public class ImageButtonSelection implements Transferable,
                                             ClipboardOwner {
    public static DataFlavor ImageButtonFlavor;
    private DataFlavor[] flavors = {ImageButtonFlavor};
    private ImageButton imageButton;

    static {
        try {
            ImageButtonFlavor = new DataFlavor(
                                  Class.forName("ImageButton"),
                                  "ImageButton");
        }
        catch(ClassNotFoundException e) {
            e.printStackTrace();
        }
        catch(java.util.mime.MimeTypeParseException e) {
            e.printStackTrace();
        }
    }
    public ImageButtonSelection(ImageButton imageButton) {
        this.imageButton = imageButton;
    }
    public synchronized DataFlavor[] getTransferDataFlavors() {
        return flavors;
    }
    public boolean isDataFlavorSupported(DataFlavor flavor) {
        return flavor.equals(ImageButtonFlavor);
    }
    public synchronized Object getTransferData(
```

```
                                              DataFlavor flavor)
            throws UnsupportedFlavorException, IOException {
            if(flavor.equals(ImageButtonFlavor)) {
                return imageButton;
            }
            else
                throw new UnsupportedFlavorException(flavor);
        }
        public void lostOwnership(Clipboard c, Transferable t) {
        }
    }
```

We have but one lone data flavor—`ImageButtonFlavor`. By now, the implementation of `ImageButtonSelection` should require no explanation, so without further ado, we'll move on to an applet that copies image buttons to and from a clipboard.

ImageButton Transfer Applet

First, let's take a look at the applet, both in its initial state and after we've copied a single image button three times from the clipboard, in Figure 20-6.

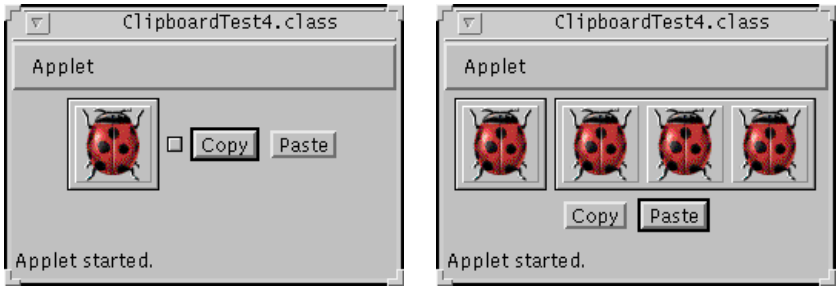


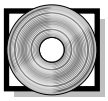
Figure 20-6 Copying and Retrieving an ImageButton to/from a Clipboard
The picture on the left shows the applet in its initial state. After we activate the Copy button once and the Paste button three times, the applet looks like the picture on the right.

Rather than bore you with the details of the inner workings of the applet, we'll have a brief discussion and show you the code. The applet is composed of four components: two instances of `ImageButtonCanvas` and two instances of `java.awt.Button`. An `ImageButtonCanvas` draws a 3D rectangle inside of a black rectangle and contains image buttons. `ImageButtonCanvas`, when it is laid out, expands to accommodate the image buttons it currently contains; thus, the original canvas on the left is just big enough to hold the single image button, while the image canvas on the right is initially at its minimum size.

Copying and retrieving image buttons to and from the clipboard is accomplished with the following (by now familiar) code:

```
class CopyListener implements ActionListener {
    public void actionPerformed(ActionEvent event) {
        ImageButton button = copyFrom.getImageButton();
        ImageButtonSelection contents =
                        new ImageButtonSelection(button);

        clipboard.setContents(contents, ClipboardTest4.this);
    }
}
class PasteListener implements ActionListener {
    public void actionPerformed(ActionEvent event) {
        Transferable contents = clipboard.getContents(this);

        if(contents != null) {
            try {
                ImageButton imageButton =
                    (ImageButton) contents.getTransferData(
                    ImageButtonSelection.ImageButtonFlavor);

                copyTo.setImageButton(imageButton);
                copyTo.invalidate();
                validate();
            }
            catch(Exception e) {
                e.printStackTrace();
            }
        }
    }
}
```

The `copyTo ImageButtonCanvas` is invalidated, and then `validate()` is invoked for its container—the applet—to force the image button canvas to be laid out whenever an image button is pasted to it. The entire applet is listed in Example 20-6.



Example 20-6 ClipboardTest4 Applet

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
import java.awt.datatransfer.*;

public class ClipboardTest4 extends    Applet
                            implements ClipboardOwner {
    private Clipboard   clipboard;
    private ImageButtonCanvas copyFrom = new ImageButtonCanvas();
    private ImageButtonCanvas copyTo   = new ImageButtonCanvas();
    private Button copy  = new Button("Copy");
    private Button paste = new Button("Paste");

    public void init() {
        clipboard = getToolkit().getSystemClipboard();

        copyFrom.setImageButton(
          new ImageButton(getImage(getCodeBase(),
                          "ladybug.gif")));

        add(copyFrom);
        add(copyTo);
        add(copy);
        add(paste);

        copy.addActionListener (new CopyListener());
        paste.addActionListener(new PasteListener());
    }
    class CopyListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            ImageButton button = copyFrom.getImageButton();
            ImageButtonSelection contents =
                      new ImageButtonSelection(button);
            clipboard.setContents(contents, ClipboardTest4.this);
        }
    }
    class PasteListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            Transferable contents = clipboard.getContents(this);

            if(contents != null) {
                try {
                    ImageButton imageButton =
                        (ImageButton) contents.getTransferData(
                          ImageButtonSelection.ImageButtonFlavor);
                    copyTo.setImageButton(imageButton);
                    copyTo.invalidate();
                    validate();
                }
                catch(Exception e) {
                    e.printStackTrace();
                }
            }
        }
```

```
    }
    public void lostOwnership(Clipboard    clip,
                              Transferable transferable) {
        System.out.println("Lost ownership");
    }
}
class ImageButtonCanvas extends Panel {
    private ImageButton imageButton;

    public ImageButtonCanvas() {
        this(null);
    }
    public ImageButtonCanvas(ImageButton imageButton) {
        if(imageButton != null)
            setImageButton(imageButton);
    }
    public void paint(Graphics g) {
        g.setColor  (Color.black);
        g.drawRect  (0,0,getSize().width-1,getSize().height-1);

        g.setColor  (Color.lightGray);
        g.draw3DRect(1,1,getSize().width-3,
                     getSize().height-3,true);

    }
    public void setImageButton(ImageButton button) {
        imageButton = button;
        add(new ImageButton(imageButton.getImage()));

        if(isShowing()) {
            repaint();
        }
    }
    public ImageButton getImageButton() {
        return imageButton;
    }
}
```

Summary

The AWT includes support for clipboards, both local clipboards and access to the system clipboard. Copying data to the system clipboard makes the data available to other programs in addition to other AWT components.

The AWT's data transfer mechanism consists of transferables and data flavors. Transferables encapsulate data and can provide access to their data in one or more data flavors.

Transferring textual data is supported by the `java.awt.datatransfer` package; however, you are on your own when transferring data of other types. To illustrate transferring nontextual data types, we've demonstrated how to transfer both images and AWT custom components to and from clipboards.