

17

Images and Animation

In this chapter we will introduce some of the basic techniques for reading images from a file and displaying them, and creating animation effects in your applications and applets. In this chapter you will learn:

- ☐ How to read an image into an application or an applet.
- ☐ How to display an image.
- ☐ How to create animation effects.
- ☐ How to draw on an image.
- ☐ How to create an image internally to your program.
- ☐ What alpha compositing is.
- ☐ How to set the transparency of an image and create fading effects.

We will use applets as the primary vehicle for working examples in this chapter, but don't forget that everything you will learn here can also be applied to your applications.

Applet Operations

We have already seen a few simple examples of applets, but we haven't really gone into how they are structured in any detail. Since we will be writing several rather more complicated applets in this chapter, we will remedy this right now.

To recap, an applet is defined by a class that has the `Applet` class as a base. We will use the Swing class `JApplet` which is derived from `Applet` to define our applets because it provides more capabilities. As you know, an applet is a program that is embedded in a web page, so it executes under the control of a web browser, or the `appletviewer` program that comes with the Java Development Kit. As we discussed in Chapter 1, you are likely to need to have the Java plug-in installed if Java 2 applets are to run with your browser. If you want to be sure that your applet will run with Netscape or Microsoft browsers without the plug-in being installed, then you must forgo the added capabilities of the `JApplet` class and limit yourself to using the `Applet` class as the base for your applet.

Whether an applet runs or not, and when it starts execution and when it stops, are all controlled by the browser, not by code in the applet. Of course, whether things work as they should still depends on the applet being implemented properly. When a browser loads a web page containing an applet, it will create an object of the class defining the applet. It will then use four methods for the applet class object to control the operation of the applet. All four methods are `public`, have a `void` return type and require no arguments. These methods are:

Method	Description
<code>init()</code>	To start your applet, the browser always calls the default constructor for your applet class to create the object, and then calls its <code>init()</code> method to do any initialization that is necessary. When <code>init()</code> is called, the applet is loaded and any parameters specified for the applet are available. You typically create any objects that are needed within the applet in this method and initialize any data members of the applet class. You will also implement the <code>init()</code> method to create any threads that the applet requires. Of course, your applet class can still have data members that you supply initial values for just like any other class. You can also implement the default constructor and do the initialization of data members there, too.
<code>start()</code>	This method is called by the browser to start or restart the operation of the applet. When the applet is started initially, the <code>start()</code> method will be called after the <code>init()</code> method has executed. The <code>start()</code> method is typically used to start any additional threads required in the applet. You don't always have to implement this method, but we will be using it a lot in this chapter to start animations.
<code>stop()</code>	This method is called when an applet should stop what it is doing temporarily – when the applet is 'hidden', for instance – because the user has scrolled the web page so the applet region is no longer visible. You would typically stop any continuously running operations that are in separate threads in this method – ones that might display an animation, for instance. This avoids wasting processor time on preparing images that can't be seen. To restart the applet the browser will call the <code>start()</code> method.
<code>destroy()</code>	This is called when the applet is being terminated so you should use it to free up any system resources that your applet uses. The <code>stop()</code> method is always called prior to this method being called.

The default implementations of these methods are inherited in your applet class, but they all do nothing. It is up to you to implement your versions of these methods so that they do what is necessary in the context of your applet. One other method that your applets should implement is `getAppletInfo()`. This method is expected to return a `String` object that contains information about the author, the version and copyright for the applet suitable for displaying in an "About" dialog.

The browser finds out about your applet through the HTML code that appears in the web page, so let's take a quick look at the bits that are likely to be involved.

HTML for Applets

In this section we will show HTML tags and attributes in upper case to highlight them, although they are not case sensitive. The basic HTML tags you need for embedding an applet in a web page are as follows:

```
<APPLET      CODE = "AppletName.class"
              WIDTH = "Applet_width_in_pixels"
              HEIGHT = "Applet_height_in_pixels" >
Optional alternate text displayed if the APPLETTAG is not supported
</APPLET>
```

The optional default text after the `APPLET` tag is useful to signal when a browser does not support the `APPLET` tag – when the applet will not be executed. Other than that, the HTML tags above are the minimum required to include an applet in a page. The value for the `CODE` attribute in the `APPLET` tag identifies the `.class` file for the applet. This presumes the applet is in the same directory as the page containing the applet. If this is not the case you must not put the path as part of the `CODE` value. Instead, you should add a `CODEBASE` attribute that has a URL as a value that identifies the source for the applet – the URL can be absolute (which would start with `"http://"`) or it can be relative to the directory containing the web page. While the `WIDTH` and `HEIGHT` attributes are mandatory, they only determine the initial space allocated to the applet. The applet can adjust these itself, as we shall see in this chapter. The double quotes around the parameter values can be omitted if the parameter value does not contain spaces or other characters that might cause confusion.

You can also specify parameter values to be read by your applet by placing `<PARAM>` tags between the `<APPLET>` and `</APPLET>` entries, one for each parameter. For example:

```
<APPLET      CODE=MyApplet.class WIDTH=200 HEIGHT=100>
<PARAM      NAME="frameRate"  VALUE="10">
<PARAM      NAME="description" VALUE="Some text">

</APPLET>
```

These tags will execute the `MyApplet` applet from the directory containing the HTML file, and make a parameter with the name `frameRate` available with the value `"10"`, plus a second parameter, `description`, with the value `"Some text"`. These values can be retrieved programmatically from within the applet – usually from within the `init()` method – by calling the `getParameter()` method with a `String` defining the parameter name for which the value is requested as the argument. If it is available, the parameter value is returned as a `String`, which you can then convert to numeric form where necessary. For example, to retrieve the value of the parameter with the name, `frameRate`, we could write:

```
int frameRate = 5; // Default frame rate
String value = getParameter("frameRate"); // Get the value of the parameter
frameRate
if(value != null)
    frameRate = Integer.parseInt(value);
```

It is important to verify that the `String` returned by the `getParameter()` method is not `null`. It will be if there is no `PARAM` tag specifying the value. Note that the parameter name is not case sensitive, so `"FRAMERATE"` would specify the same parameter as `"framerate"`.

When your applet expects to read parameter values, it is a good idea to override the default `getParameterInfo()` method in your applet class to identify their names, their types, and their description in an array of strings. The version of the method that your applet class will inherit from the `Applet` class returns `null`. This method can be used by the applet context (the web page) to find out about your applet. For the applet `MyApplet`, that expects values for the parameters `frameRate` and `description`, you would implement the `getParameterInfo()` method as:

```
public String[][] getParameterInfo()
{
    String[][] pInfo = {
        { "frameRate", "integer", "The frames per second" },
        { "description", "String", "Description of the animation" }
    };
    return pInfo;
}
```

There must be three elements in each row of the array providing information about a particular parameter. The three elements in a row describe the parameter name, the type of value and the purpose of the parameter respectively.

*Remember to use the **OBJECT** and **EMBED** tags as shown in Chapter 1 to add Java 2 support to browsers other than **appletviewer**.*

If you are using this book in non-linear fashion you will find more on deploying applets at the end of Chapter 12 where I also briefly discuss the security issues relating to applets. This is an extensive topic and a thorough discussion of it is beyond the scope of this book. For the most up-to-date documentation regarding the security architecture in Java 2, refer to the web pages at: <http://java.sun.com/security/>

Let's now turn to the fundamentals of dealing with images, and then look at how we can implement an applet from the ground up – this time to handle an image.

Obtaining an Image

We have already seen one way to obtain an image from a file using the `ImageIcon` class back in Chapter 13. The `ImageIcon` class doesn't really care about the size of the image, that is, it is not constrained to be a typical icon size, so an image represented by an object of this class can be anything. We only considered one `ImageIcon` constructor when we read icons for our toolbars buttons from a file. That constructor happened to accept a `String` argument that specified the file name. There are a number of other `ImageIcon` constructors, so here's the complete set:

Constructor	Description
<code>ImageIcon()</code>	Creates an uninitialized object that you must initialize with an <code>Image</code> object before use. You would do this by passing a reference to an <code>Image</code> object to the <code>setImage()</code> method for the <code>ImageIcon</code> object.
<code>ImageIcon(String filename)</code>	Creates an object from the file specified by <code>filename</code> , which represents either an absolute path for the file, or a path relative to the current directory.
<code>ImageIcon(String filename, String description)</code>	As the constructor above, but stores a description of the image specified by the second argument, and which can be retrieved by calling the <code>getDescription()</code> method for the <code>ImageIcon</code> object.
<code>ImageIcon(URL location)</code>	Creates an object specified by the file that is located at the source specified by the argument. The <code>URL</code> class represents a uniform resource locator specification that identifies a source on the World Wide Web. We will come back to this class a little later in the chapter.
<code>ImageIcon(URL location, String description)</code>	Same as the previous constructor but adds a description of the image that can be retrieved by calling the <code>getDescription()</code> method for the <code>ImageIcon</code> object.
<code>ImageIcon(Image image)</code>	Creates an <code>ImageIcon</code> object from the <code>Image</code> object supplied as the argument. We will discuss the <code>Image</code> class later in this section.
<code>ImageIcon(Image image, String description)</code>	Creates an <code>ImageIcon</code> object from the <code>Image</code> object supplied as the first argument. The second argument provides a description of the image.
<code>ImageIcon(byte[] imageData)</code>	Creates an object from the byte array that must contain data containing an image in a supported format – which can be GIF, PNG, or JPEG at the present time. The data can be read from a file or created programmatically.
<code>ImageIcon(byte[] imageData, String description)</code>	As the previous constructor but also stores a description of the image.

An `ImageIcon` object contains a reference to an object of type `Image` as a member. The `Image` class is an abstract class that is a superclass of all classes that represent images in Java, so a reference of type `Image` can refer to any instance of a graphical image being used. You can obtain a reference to the `Image` member of an `ImageIcon` object by calling its `getImage()` method. An object of type `Image` can be created from data in GIF (Graphics Interchange Format), PNG (Portable Network Graphics), or JPEG (Joint Photographic Experts Group) format. So, whether the image data is passed to the constructor as an array – as is the case for two of the `ImageIcon` constructors, or is obtained from an external source such as a file or a `URL` object, it must be a GIF, PNG, or JPEG representation of an image.

While the GIF format for images is very common, it has some limitations. A GIF image can contain a maximum of 256 different colors, albeit selected from a palette of 16 million colors. The JPEG image format has been designed to store photographs, and is a much more sophisticated way of representing a color image. The sophistication comes at a price though. The complexity of the format and the data compression technique used means that it takes much longer to process than a GIF image, but it does make it feasible to transmit good quality photographic images over the net. The JPEG compression technique is also 'lossy', so the quality is not as good as the original image. The PNG image format is also much more flexible than GIF images and it stores images in a lossless form. It is designed to be a portable image storage form for computer-originated images. You can represent grayscale, indexed-color and true color images in PNG format, and you can also include an alpha channel that determines the transparency of the image when it is combined with others.

The constructors that create the `ImageIcon` object by referencing a `String` object specifying a file name, or by referencing a `URL` object defining an Internet source, always construct the internal `Image` object before returning. When you specify a `URL` as the source of the image data, there can be a considerable delay before the `ImageIcon` object creation is completed, depending on how long it takes to retrieve the image data from the source. Being able to create an `ImageIcon` object from a `URL` object is particularly relevant to applets. You can retrieve any images, or indeed any other external data that you use in your applet from an Internet source. Since `URLs` are so important, let's take a brief detour into how `URLs` and the `URL` class work.

Identifying Sources on a Network

Data sources on a network are identified by a **Uniform Resource Locator**, or **URL**. A source identified by a `URL` can be a file, or can be some other facility that makes data available, such as a search engine for instance. A `URL` is a relatively straightforward extension of the normal way of identifying a data source file by the path. It just has some extra bits to identify the computer that contains the source, and how you should communicate with the computer to get at the file. A `URL` is made up of four pieces of information, some of which may be omitted entirely or take default values:

Protocol	Domain Name	Port Number	Path and File Name
http://	www.wrox.com	:80	/index.html
ftp://	java.sun.com	:21	

The protocol identifies the particular communications method that is to be used to get at the file. **HTTP** is the **H**yper**T**ext **T**ransfer **P**rotocol used on the World Wide Web, **FTP** is the **F**ile **T**ransfer **P**rotocol, and there are others (we'll see one in the next chapter).

The domain name identifies the data source uniquely. The last part of the name identifies the kind of organization that owns the computer. For instance, a domain name ending in `.com` is usually a commercial organization in the USA such as Wrox Press or Sun, `.co.uk` is a UK company, `.edu` is an educational establishment in the USA, `.ac.uk` is an academic or research organization in the UK and `.gov` is a government or other public body. These are just a few examples. There are lots of others.

A port is just a number identifying a particular service on a computer. A computer can have many ports – each providing a different service. You don't usually need to specify the port number in a URL because a standard port number, which is typically associated with a particular protocol, will be assigned by default. The port number for the `http` protocol is 80 for example, and the port number for `ftp` is 21.

Because the domain name is part of the reference to a data source, every data source in a network has a unique URL. As long as you know the URL for the source of the data you want on the Internet, and the machine containing it is connected and active, you can go straight to it. Assuming, of course, you have permission!

The URL Class

As you saw when we discussed `ImageIcon` constructors, the `URL` class encapsulates URLs. This class is defined in the `java.net` package, so you will need an `import` statement for the package or for the class when you refer to the `URL` class in your code. A `URL` object identifies a particular URL and provides you with the tools to access it and read it, wherever it is on the network. You can create a `URL` object from a string specifying the URL, for example:

```
URL sourceURL = new URL("http://www.wrox.com/");
```

This defines a `URL` object corresponding to the home (or default) page on the Wrox web site.

Perhaps the most commonly used `URL` constructor accepts two arguments – a `URL` object identifying an Internet source, plus a `String` argument which is a specification for a source within the context of the `URL` specified by the first argument. The popularity of this form is due to the definition within the `Applet` class of a method, `getCodeBase()`, which returns the `URL` for the source where the `.class` file for the applet is stored. This enables you to create the `URL` for a file, `picture.gif` say, which is stored in the same directory as the applet code, wherever it is, with a statement such as:

```
URL getThePicture = new URL(getCodeBase(), "picture.gif");
```

There is another method in the `Applet` class (and therefore inherited in the `JApplet` class) that provides a useful `URL` object. This is the `getDocumentBase()` method that returns a `URL` object corresponding to the `URL` for the document (the `.html` file in other words) that contains the applets. This may well be different from the location of the applet itself. You could call this method to obtain the `URL` like this:

```
URL doc = getDocumentBase();           // Get the document containing the applet
```

You can also define a `URL` object by supplying a constructor with separate values for the protocol, the host name, the port number and the source name. You specify the port number as type `int`, and the other arguments as type `String`. The web site defined by the domain name `"www.ncsa.uiuc.edu"` contains a page that explains what a `URL` is. The file path and name for the page is `"/demoweb/url-primer.html"`. You could define a `URL` object for this page with the statement:

```
URL sourceURL = new URL("http",           // Protocol
                        "www.ncsa.uiuc.edu", // Host name
                        "/demoweb/url-primer.html"); // File
```

This uses the default port number for the protocol. You just need to supply the protocol, the domain name and the file path as `String` objects. Note that URL paths are always specified using forward slashes as separators, regardless of the type of server hosting the data source.

There is an alternative constructor defined in the `URL` class:

```
URL sourceURL = new URL("http",           // Protocol
                        "www.ncsa.uiuc.edu", // Host name
                        80,                 // Port number
                        "/demoweb/url-primer.html"); // File
```

The value of 80 for the port number is the standard port number for `http`. The value of -1 for the port name selects the default port number for the protocol. If you specify a protocol that cannot be identified, a `MalformedURLException` will be thrown.

Once you have a `URL` object, you can get the components of the URL by calling the `getProtocol()`, `getHost()`, `getPort()` and `getFile()` methods for the object. The `getPort()` method returns the port number as type `int`, and the other methods return objects of type `String`.

The `URL` class also implements an `equals()` method that you can use to compare two `URL` objects. The expression `URL1.equals(URL2)` will return `true` if `URL1` and `URL2` specify the same file on the same host via the same port number and using the same protocol.

Perhaps the most useful method in the `URL` class is the `openStream()` method. This returns an object of type `InputStream`, which you can use to read the file represented by the URL. We can try this out by reading the file referenced by the `sourceURL` object we created in the code fragment above.

Try It Out – Reading a URL

This example assumes you have a live connection to the Internet. So when the program closes, the connection may still be open, in which case you will need to close it manually. Here's the program to read a URL:

The program assumes that the directory `JunkData` that we used in Chapter 8 is still on your `C:` drive. If it isn't, you can add it, or better still, add the code to check that the directory is there, and if it isn't, create it.

```
import java.net.*;
import java.io.*;

class ReadURL
{
```



```

public static void main(String[] args)
{
    try
    {
        // Define a URL
        URL sourceURL = new URL(
            "http://www.ncsa.uiuc.edu/demoweb/url-primer.html");

        // Get a character input stream for the URL
        BufferedReader in = new BufferedReader(
            new InputStreamReader(
                sourceURL.openStream()));

        // Create the stream for the output file
        PrintWriter out = new PrintWriter(
            new BufferedWriter(
                new FileWriter(
                    new File("C:/JunkData/netdata.html"))));

        System.out.println("Reading the file " + sourceURL.getFile() +
            " on the host " + sourceURL.getHost() +
            " using " + sourceURL.getProtocol());

        // Read the URL and write it to a file
        String buffer; // Buffer to store lines
        while(!(null==(buffer=in.readLine()))
            out.println(buffer);

        in.close(); // Close the input stream
        out.close(); // Close the output file
    }
    catch(MalformedURLException e)
    {
        System.out.println("Failed to create URL:\n" + e);
    }
    catch(IOException e)
    {
        System.out.println("File error:\n" + e);
    }
}
}

```

While we have used a URL that references a source on the Internet, you could also use a URL to specify a file on your local machine. Executing this as written produces the output:

Reading the file /demoweb/url-primer.html on the host www.ncsa.uiuc.edu using http

It will also write the text contents of the URL to the file netdata.html. Have a look at what you've captured. It's a good read.

How It Works

The object `sourceURL` is created directly from the URL string. Because the constructor can throw a `MalformedURLException` – if you type the URL string incorrectly, for example – we have a `catch` block for this exception following the `try` block. Note that this exception is derived from `IOException` so the `catch` block for this must precede the `catch` block for `IOException`. Depending on whether your dial-up-networking is working properly, another typical exception can be `java.net.UnknownHostException`.

By calling the `openStream()` method for the URL object, we obtain a stream that we can pass to the `InputStreamReader` constructor. A character reader will automatically take care of converting characters from the stream to Unicode. From the `InputStreamReader` we create a `BufferedReader` object so input from the stream will be buffered in memory.

The file is a sizable .html file, so rather than writing it to the screen, we write it to a file in the `JunkData` directory that we used in Chapter 8. You will then be able to view the file using your web browser. Once we have the `BufferedReader` object for the URL, the reading of the URL and writing the local file follows the sort of process you have already seen in Chapter 8. We just use the `readLine()` method to read a line at a time from the URL into the string buffer, and we use the `println()` method to transfer the buffer contents to the local file. You can now read more about URLs by reading the file contents at your leisure. In the meantime, let's get back to images.

Displaying an Image

You display an image in essentially the same way as you display anything else – by calling a method for a `Graphics` or `Graphics2D` object. The `drawImage()` method will draw an image in a graphics context, so from this you can immediately deduce that you can draw images on any components you like. Let's go straight away to trying this out in a working example using an `ImageIcon` object in the first instance.

Try It Out – Displaying an Image

We will implement an applet that creates an `ImageIcon` from a file in the same directory as the applet's .class file. The code here will refer to an image file `wrox_logo.gif`. If you want to use this file, you can download it from the Wrox web site. Alternatively you can use your own .gif or .jpg file. Here's the code for the applet:

```
import java.awt.*;
import javax.swing.*;
import java.net.*;

public class DisplayImage extends JApplet
{
    public void init()
    {
        ImageIcon icon = null;
        try
        {
            icon = new ImageIcon(new URL(getCodeBase(),"Images/wrox_logo.gif"));
        }
        catch(MalformedURLException e)
```

```

    {
        System.out.println("Failed to create URL:\n" + e);
        return;
    }

    int imageWidth = icon.getIconWidth();    // Get icon width
    int imageHeight = icon.getIconHeight();  // and its height
    resize(imageWidth,imageHeight);          // Set applet size to fit the image

    // Create panel a showing the image
    ImagePanel imagePanel = new ImagePanel(icon.getImage());

    getContentPane().add(imagePanel);        // Add the panel to the content pane
}

// Class representing a panel displaying an image
class ImagePanel extends JPanel
{
    public ImagePanel(Image image)
    {
        this.image = image;
    }

    public void paint(Graphics g)
    {
        g.drawImage(image, 0, 0, this);      // Display the image
    }

    Image image;                             // The image
}

```

You will need an HTML file to run the applet. You could call it `DisplayImage.html`, and make the contents:

```
<applet code="DisplayImage.class" width=50 height=50></applet>
```

This just sets the width and height parameters for the applet to arbitrary values. The applet will resize itself to accommodate the image. When I ran this with `appletviewer`, it displayed the window shown: (You may prefer to refer back to Chapter 1, which explains how to run the applet in a browser using the plug-in.)



How It Works

We retrieve the icon from the file `wrox_logo.gif` in the `Images` directory relative to the URL that holds the code for the applet. The `URL` class constructor can throw an exception, so we must arrange to catch exceptions of type `MalformedURLException` here in order to get the code to compile. The `ImageIcon` class constructor does everything necessary to obtain the data from the file, and uses it to create the image. The constructor will only return when this has been completed.

The `getIconWidth()` and `getIconHeight()` methods for the `ImageIcon` object return the width and height of the image respectively, and we use these as arguments to the `resize()` method for the `JApplet` object, to adjust the size of the applet to accommodate the image.

We have to remember that an applet based on the `JApplet` class is similar to a window based on the `JFrame` class in that anything we want to display in an applet must be added to the content pane for the applet object. For this reason we define an inner class, `ImagePanel`, that will draw the image for the `ImageIcon` object, and we can add an object of this class to the content pane for the applet. The `ImagePanel` class is quite simple. The constructor saves the reference to the `Image` object that is passed to it in a data member of the class, and the `paint()` method calls `drawImage()` to display the image.

Note that no cast to `Graphics2D` is necessary in the `paint()` method since the `drawImage()` method that we are using is defined in the `Graphics` class. There are several other overloaded versions of this method, including two that are defined in the `Graphics2D` class, so when you want to use these, a cast will be necessary. The arguments to the version of `drawImage()` that we use here are:

- ❑ A reference to the `Image` object to be drawn
- ❑ The coordinates of the position where the image is to be drawn
- ❑ A reference to an `ImageObserver` object

An image is defined by its top-left corner point, so in our applet we place the top-left corner of the image at the origin of the user coordinate system for our panel. An `ImageObserver` object is an object of a class that implements the `ImageObserver` interface, and since the `Component` class implements this interface, any component, including objects of our inner class `ImagePanel`, are `ImageObserver` objects. The `ImageObserver` interface handles the process of accessing image data from Internet sources where there may be a considerable time delay in loading the image. This interface provides the means to allow you to determine whether an image has been loaded or not, and we will come back to this on the next page.

Creating Image Objects

The `Image` class is at the heart of all images in Java. An image will always be an object of a class that has `Image` as a base. You can't directly create an `Image` object that contains an image because the class is abstract. While a reference of type `Image` is used to refer to an object that encapsulates the data for an image, the `Image` class does not provide you with the means of obtaining the image data. There has to be some other mechanism for creating an `Image` object corresponding to a particular image.

The `Applet` class defines two overloaded versions of a method, `getImage()`, that return a reference to an `Image` object. One version accepts an object of type `URL` as an argument that specifies the source of the image data. The other accepts two arguments, a `URL` object plus a `String` object, where the `String` object defines the specification of the source of the image data in the context of the `URL` object. In the previous example, we could have created an `Image` object directly with the following statement:

```
Image image = getImage(new URL(getCodeBase(),"Images/wrox_logo.gif"));
```

We still need to put this in a `try` block because of the possibility of a `MalformedURLException` (or others) being thrown by the `URL` class constructor. There is a significant difference between this statement obtaining a reference to an `Image` object for an image from a given source, and the statement in the example that created an `ImageIcon` object. When the `ImageIcon` class constructor returned to the `init()` method, the object was fully constructed and the image was available. In the statement above we get a reference to an `Image` object returned that *may* at some time in the future contain the data for the image. This gets over the problem implicit in using an `ImageIcon` object – it hangs up your applet until all the image data for the icon has been retrieved from the source. When you call `getImage()`, your applet code can continue to execute and get on with other things while the image data is being downloaded. This still leaves the question of determining when the image is actually here which is where the `ImageObserver` interface comes in.

Image Observers

The `Component` class implements the `ImageObserver` interface so all components will inherit this implementation. The `ImageObserver` interface declares one method, `imageUpdate()`, that is called automatically when information about an image that was previously requested becomes available. The `imageUpdate()` method will draw the image on the component. So how do you request information about an image?

One way is to call the `drawImage()` method for a `Graphics` object. By calling this method you implicitly request all the information necessary to draw the image in the graphics context – the width, the height, the pixel data and so on. As you saw, the last argument to the `drawImage()` method was a reference to an `ImageObserver` object that in our case was the panel on which the image was to be drawn. If any of the image data was not available at the time of the call, the `imageUpdate()` method for the `ImageObserver` object would be called as more information became available.

Another way you can request information about an image is to call methods for the `Image` object itself. The `Image` class defines methods `getWidth()` and `getHeight()` to access the width and height of the image. Both of these methods require a reference of type `ImageObserver` to be passed – normally a reference to the component on which the image is to be drawn – and that object's `imageUpdate()` method is called when the information becomes available.

The last part we need to understand in all this is exactly what the `imageUpdate()` method does when it is called. It simply draws the image incrementally with the information available. This has the effect on a slow Internet link or with a large image of making the image appear piecemeal, as and when the data becomes available. This usually gives the user a visual cue that, even though the entire image has not been retrieved yet, something is still happening.

Implementing imageUpdate()

Most of the time, the default implementation of `imageUpdate()` is sufficient – it just repaints the component. If we use the `getImage()` method in the previous example to load the image, we have two objects that are interested in the image data, the applet object and the `ImagePanel` object. For the `ImagePanel` object, the default `imageUpdate()` method is fine. The applet object is different. It wants to know the height and the width of the image so it can resize itself to fit the image. Calling `getWidth()` and `getHeight()` for the `Image` object provide these values if they are available, but these may not be available during the execution of the `init()` method even when the image file is local, in which case these methods will return `-1`. In this case therefore, we can usefully implement our own version of the `imageUpdate()` method to do what we want.

There are six parameters to the `imageUpdate()` method and it returns a `boolean` value so its outline implementation is like this:

```
public boolean imageUpdate(Image img,      // Reference to the image
                           int flags,     // Flags identifying what is available
                           int x,        // x coordinate
                           int y,        // y coordinate
                           int width,     // Image width
                           int height)    // Image height
{
    // Code to respond to data that is available...
}
```

The `imageUpdate()` method will be called when any new item of image data becomes available, so the data will typically be incomplete. You might have the width of the image available for instance but not the height. A return value of `false` indicates that you have received all the image data that you need, otherwise the method should return `true` so it will be called again later when more information is available.

The reference passed as the first parameter identifies the image that the call relates to, so if your class object is observing more than one image, you can use this to tell to which `Image` object the call of the method relates. The second parameter provides the key to determining what image data is available. The `flags` value is a combination of one or more of the following single bit flags:

Parameter	Description
<code>WIDTH</code>	The width of the image is available.
<code>HEIGHT</code>	The height of the image is available.
<code>SOMEBITS</code>	Some more pixels required for a scaled version of the image are available, and the bounding rectangle for these is given by the <code>x</code> , <code>y</code> , <code>width</code> and <code>height</code> values.
<code>ALLBITS</code>	This flag indicates that all the bits of an image that has been drawn previously are now available. This enables the <code>updateImage()</code> method to decide when to return <code>false</code> when an image is being drawn incrementally.
<code>FRAMEBITS</code>	This indicates that another complete frame of a multi-frame image that has been previously drawn is now available to be drawn again. This can be used to determine when to call the <code>repaint()</code> method to display more of an image.
<code>PROPERTIES</code>	This flag indicates that the properties of the image are now available. These are obtained by calling the <code>getProperties()</code> method for the <code>Image</code> object.
<code>ABORT</code>	This flag indicates that the process of retrieving the image was aborted and no further image data will be available.
<code>ERROR</code>	This flag indicates that an error has occurred retrieving the image and no further image data will be available.

We aren't going to go into the detail of how all these flags are used, but the complete set is here for reference. We are only interested in the height and the width of the image in our applet, so we can implement the `imageUpdate()` method as:

```
public boolean imageUpdate(Image img,          // Reference to the image
                           int flags,          // Flags identifying what is available
                           int x,              // x coordinate
                           int y,              // y coordinate
                           int width,          // Image width
                           int height)         // Image height
{
    if((flags & WIDTH) > 0 && (flags & HEIGHT) > 0)
    {
        resize(width,height);                  // Set applet size to fit the image
        repaint();                             // Repaint the applet in its new size
        return false;
    }
    else
        return true;                          // More info required
}
```

We only want to resize the applet when we have both the width and height of the image, so we test for each of these flags by bitwise ANDing the `flags` value with the `WIDTH` and `HEIGHT` values that are defined in the `ImageObserver` interface. If the results of both operations are positive, both flags were set so we can resize the applet and repaint it. While either or both of the `WIDTH` and `HEIGHT` flags are not set, we return `true` so the method will be called again. Of course, the HTML rendering in a browser will work much better if the correct size for the applet is specified at the outset.

Try It Out – Implementing the `imageUpdate()` Method

Let's try out the whole applet in its revised guise:

```
import java.awt.*;
import javax.swing.*;
import java.net.*;

public class DisplayImage extends JApplet
{
    public void init()
    {
        Image image = null;
        try
        {
            // Image from a file specified by a URL
            image = getImage(new URL(getCodeBase(),"Images/wrox_logo.gif"));
        }
        catch (MalformedURLException e)
        {
            System.out.println("Failed to create URL:\n" + e);
            return;
        }
    }
}
```

```

        int imageWidth = image.getWidth(this);           // Get its width
        int imageHeight = image.getHeight(this);         // and its height
        if(imageWidth != -1 && imageHeight != -1)         // If they are available
            resize(imageWidth,imageHeight);              // set applet size to fit

        ImagePanel imagePanel = new ImagePanel(image);   // Create panel showing the image
        getContentPane().add(imagePanel);               // Add the panel to the
        content pane
    }

    public boolean imageUpdate(Image img,               // Reference to the image
                              int flags,               // Flags identifying what is available
                              int x,                   // x coordinate
                              int y,                   // y coordinate
                              int width,               // Image width
                              int height)              // Image height
    {
        if((flags & WIDTH) > 0 && (flags & HEIGHT) > 0)
        {
            resize(width,height);                       // Set applet size to fit the image
            repaint();                                   // Repaint the applet in its new size
            return false;
        }
        else
            return true;                                // More info required
    }

    // Class representing a panel displaying an image
    class ImagePanel extends JPanel
    {
        public ImagePanel(Image image)
        {
            this.image = image;
        }

        public void paint(Graphics g)
        {
            g.drawImage(image, 0, 0, this);              // Display the image
        }

        Image image;                                    // The image
    }
}

```

How It Works

The effect of the applet will appear much the same as before because the image file is local so everything will happen quite quickly. You can verify that the image data is not available in the `init()` method for the applet by commenting out the `if` that checks for a value of `-1` for the `width` or the `height` before calling `resize()`. Without this check, the `resize()` method will be called with either or both arguments as `-1` so the size of the applet will briefly become miniscule and the applet viewer window along with it.

A further experiment will show that our `imageUpdate()` method in the `DisplayImage` class is only called if we have previously requested image data that is not available. Modify the code in the `init()` method that calls the methods to obtain the image size like this:

```
int imageWidth = 30, imageHeight = 30;           // Added...

// imageWidth = image.getWidth(this);           // Get its width
// imageHeight = image.getHeight(this);         // and its height
// if(imageWidth != -1 && imageHeight != -1)
//     resize(imageWidth,imageHeight);           // Set applet size to fit the image
```

Now the applet will not be resized at all and the image will not be displayed. If you uncomment either or both of the statements calling `imageWidth` and `imageHeight` and recompile, everything will work as it should. To see when `imageUpdate()` is called, you could add a call to `System.out.println()` at the beginning of the method.

The default `imageUpdate()` method is called automatically for the `ImagePanel` object because the `paint()` method calls `drawImage()` and identifies the object as the image observer. The default version calls `repaint()` as new data becomes available, so over a slow link our image will be drawn incrementally.

Animation

You can create animated effects in Java on any component. You can create animation in a window or a panel, your buttons can have animated labels, and you can even animate your menu items if you wish. The general principles for producing animated images on your screen are the same as for a film. You display, or draw, a series of static images on the screen with a fixed interval of time between one image and the next. Each image differs slightly from its predecessor so that an object that is in a different position on successive images will appear to move. Since you know how to display one image you are part way there, and since you also know how to create a loop it is clearly not going to be too difficult to implement animated images.

There are two basic ways in which animation can be generated. You can create or obtain a set of images that are snapshots of the position of everything at fixed intervals, and then display them in sequence. Alternatively you can create or obtain an image of whatever you want to have moving, and display it at different positions at fixed intervals of time. Of course, before you display the moving entity at any given position, you must erase it at whatever position it was previously. Come to think of it, you already know one way to do this. Drawing a line or a circle in *Sketcher* produces an animated effect while you drag the mouse cursor.

Animation is often used in applets to make web pages more interesting and eye catching and there can be multiple, independent animated images in a page. The code producing an animated effect generally runs continuously, so you usually have to make this independent of any other code that may be running to allow the apparent concurrent operations. For this reason, you always implement code that generates an animated effect as a separate thread. If you don't implement your animation in a separate thread, it is unlikely to work properly, and other code that you expect to be executable while the animation is running will not work either. This goes for animations in applications as well as applets. We have already discussed threads in some detail so we just need to dredge the stuff up again and apply it for drawing images.

An Animated Applet

Just so that you know where we are heading, our first program illustrating animation will be an applet that drops the Wrox logo from a great height to see what happens. Since Wrox Press is an immensely resilient company, the logo will bounce.

To implement this we just need to draw the logo at its new position at fixed intervals of time – the new position being determined by how far the logo has fallen during the time interval. We can store the coordinates of the current location of the image in data members of the applet class, `imageX` and `imageY` and the `paint()` method of the inner class `ImagePanel` will draw the image at that position. The animation code in the `run()` method will need access to the height of the image so it can work out when the image hits the ground, so we should store the image dimensions, `imageWidth` and `imageHeight`, as data members too.

We will call the applet class `LogoBounce`, so the outline contents of the source file will be:

```
import java.awt.*;
import java.awt.image.*;
import javax.swing.*;
import java.net.*;

public class LogoBounce extends JApplet
    implements Runnable
{
    // This method is called when the applet is loaded
    public void init()
    {
        // Code to initialize the applet...
    }

    // This method is called when the browser starts the applet
    public void start()
    {
        bouncer = new Thread(this);           // Create animation thread
        bouncing = true;
        bouncer.start();                       // and start it
    }

    // This method is called when the browser wants to stop the applet
    // - when is it not visible for example
    public void stop()
    {
        bouncing = false;                     // Stop the animation loop
        bouncer = null;                       // Discard the thread
    }

    // This method is called when the animation thread is started
    public void run()
    {
        // Code for the animation thread...
    }
}
```

```

class ImagePanel extends JPanel
{
    public ImagePanel(Image image)
    {
        this.image = image;
    }

    public void paint(Graphics g)
    {
        // Initialize the animation...

        while(bouncing)
        {
            // Code for the animation loop
        }
    }

    Image image;                                // The image
}

Thread bouncer;                                // The animation thread
boolean bouncing = false;                      // Controls animation thread
ImagePanel imagePanel;                        // Panel for the image
int imageWidth, imageHeight;                  // Image dimensions
int imageX, imageY;                           // Current image position
}

```

All the basic things we need are here. In the `init()` method we will set up a component on which we can draw an image and add this to the content pane for the applet, just as we did in the previous example. The `start()` method creates the animation thread, `bouncer`; sets the variable, `bouncing`, that will control the animation loop to `true`; and starts the thread. The `run()` method will contain the animation loop that will continue to run as long as `bouncing` is `true`. In the `stop()` method we just set `bouncing` to `false` to stop the animation loop in the `run()` method, then discard the animation thread object by setting `bouncer` to `null`. The `start()` method will create a new thread if the animation needs to be restarted.

We will fetch the image from the file in the `init()` method using the `getImage()` method as we did in the previous example. There's a complication here though. The browser will call `start()` as soon as the `init()` method returns to start the applet, and this will start the thread to do the animation. Since this will involve calculations involving the height of the image, we don't want this to go ahead until we are sure the height is available. One way of determining when an image has been loaded is to make use of something called a **media tracker**.

Using a Media Tracker

A media tracker is an object of type `MediaTracker` that is used specifically for tracking the loading of images. This class is defined in the `java.awt` package, and while it only manages the loading of images at present, it may be extended in the future to track the loading of other kinds of media. There is just one `MediaTracker` constructor that expects to be passed a reference to a component as the argument – the component object being the one that is loading the images.

Using a media tracker is quite simple. After calling the `getImage()` method to obtain a reference to the `Image` object, you then pass the `Image` reference to the tracker by calling its `addImage()` method. This makes the `MediaTracker` object responsible for loading the image. There are two arguments to the `addImage()` method: the reference to the image to be tracked and an identifier of type `int` that is used to track the image. We can create and track our image in the `init()` method with the code:

```
public void init()
{
    tracker = new MediaTracker(this);
    Image image = null;
    try
    {
        // Image from a file specified by a URL
        image = getImage(new URL(getCodeBase(), "Images/wrox_logo.gif"));
    }
    catch(MalformedURLException e)
    {
        System.out.println("Failed to create URL:\n" + e);
    }
    tracker.addImage(image, 1);

    // Plus the rest of the code for the method...
}
```

We have specified the identifier for our image as 1. You can use a single `MediaTracker` object to track multiple images and several images can have the same identifier. The value of the identifier determines the priority for loading images; images associated with an identifier of a lower value will be loaded first. You start loading all the images associated with a particular identifier and wait for them to be loaded by calling the `waitForID()` method for the tracker object, and passing the identifier to it. This method will only return when all the images associated with the identifier have been loaded. This method can throw an exception of type `InterruptedException` if this thread gets interrupted by another thread, so you must put calls in a `try` block and catch the exception. In our implementation of `init()` we could write:

```
try
{
    tracker.waitForID(1);           // Load and wait for images with ID of 1
}
catch(InterruptedException e)
{
    System.out.println(e);         // Thread was interrupted
}
```

Clearly, if you want to wait for individual images to be loaded, you should give each of them a unique identifier. A common technique when loading multiple images is to store the references in an array, and use the array index for each image as its identifier.

You can also initiate loading and wait for all the images being tracked to be loaded by calling the `waitForAll()` method for the tracker. We could equally well write in our `init()` method:

```

try
{
    tracker.waitForAll();           // Load and wait for all images
}
catch(InterruptedException e)
{
    System.out.println(e);         // Thread was interrupted
}

```

This will initiate loading of our single image and return when it has been loaded.

While the two wait methods we have discussed will wait indefinitely, there are overloaded versions of both methods that accept an extra argument specifying the maximum number of milliseconds that the method should wait for the image to be loaded. In this case you won't know when the method returns whether the image or images have actually been loaded, or whether the time ran out. You can call the `checkAll()` method for the `MediaTracker` object with an ID as the argument to test this. A `true` return value indicates that the images for the ID have been loaded. You can also use a version of `checkAll()` with no argument to check whether all the images associated with the tracker have been loaded.

Even though `checkAll()` may indicate that all images have been loaded, you can't assume that no errors occurred while the images were loading. You must test for errors explicitly. You can call the `isErrorAny()` method to determine if any errors occurred with any of the images – a `false` return value indicating that there were no errors. If you want to test for errors more specifically, you can pass an ID to the `isErrorID()` method. A return value of `true` indicates that an error occurred in loading at least one of the images associated with the ID. If you want to know which image or images caused an error, you can call the `getErrorsID()` method with the ID as the argument. This method will return an array of references (as type `Object[]`) to the images for which a loading error occurs. Of course, if you use a unique ID for each image, you won't need to do this.

We can implement the `init()` for our applet as:

```

public void init()
{
    tracker = new MediaTracker(this);
    Image image = null;
    try
    {
        // Image from a file specified by a URL
        image = getImage(new URL(getCodeBase(),"Images/wrox_logo.gif"));
    }
    catch(MalformedURLException e)
    {
        System.out.println("Failed to create URL:\n" + e);
    }

    tracker.addImage(image,0);

    try
    {
        tracker.waitForAll();           // wait for image to load
        if(tracker.isErrorAny())        // If there is an error
            return;                    // give up
    }
}

```

```

        imageWidth = image.getWidth(this);    // Get image width
        imageHeight = image.getHeight(this);  // and its height
        resize(imageWidth,imageHeight);       // set applet size to fit
        imageY = -imageHeight;                // for image
        imagePanel = new ImagePanel(image);    // Create panel showing the image
        getContentPane().add(imagePanel);     // Add the panel to the content pane
    }
    catch(InterruptedException e)
    {
        System.out.println(e);
    }
}

```

We use a media tracker to manage loading of the image, and we only get the height and width of the image once we are sure that the image has been loaded with no errors. We set the applet size to accommodate the image and set the y coordinate of the starting position so that the image will be just out of sight above the x axis – which is the top of the applet. We create an `ImagePanel` object to display the image and add it to the content pane of the applet object.

The `init()` code assumes that `tracker` is a data member of our applet class, so add the following line to your class:

```
MediaTracker tracker;           // Tracks image loading
```

Of course, if there was an error loading the image, we don't want to start the animation thread, so we modify the `start()` method:

```

public void start()
{
    if(tracker.isErrorAny())           // If any image errors
        return;                       // don't create the thread
    bouncer = new Thread(this);
    bouncing = true;
    bouncer.start();
}

```

That's sufficient information about media trackers for our purposes, so let's get back to our animation. The main purpose of the code in the animation thread, the `run()` method in our applet, will be to determine when the next image has to be drawn. It will therefore boil down to calling a method that draws an image at fixed intervals. If you want your animation to display 10 frames per second, then every 100 milliseconds you want it to display another frame. Let's look at how we can implement the `run()` method to determine when the required time interval is up, to cue the drawing of another image.

Measuring Time Intervals

We will want to display an image at fixed intervals, of `interval` milliseconds say, from some arbitrary start time. We can record a starting time by calling the static `currentTimeMillis()` method in the `System` class. This returns the current time from the system clock in milliseconds as a value of type `long`. We could save this as our starting time with the statement:

```
long time = System.currentTimeMillis();
```

At this point we will immediately draw the image, so the next instance of drawing the image should be at the time `time+interval`. Of course, we have to take account of the time spent processing subsequent to the instance when the image was last displayed, so the time we need to wait before displaying the next image is going to be less than `interval`. However, this is not too difficult to organize. All we need to do is wait until the value returned by `currentTimeMillis()` is equal to or greater than this value, and the `sleep()` method for a thread will do this very nicely. After each interval, we just add the `interval` value to `time` to get the time when the next repaint of the image should occur. We also need to figure out where the image is at the end of each interval.

The basic code for the `run()` method is going to be:

```
public void run()
{
    long time = System.currentTimeMillis(); // Starting time
    long interval = 20;                    // Time interval msec

    // Move image while bouncing is true
    while(bouncing)
    {
        imagePanel.repaint();              // Repaint the image

        // Wait until the end of the next interval
        try
        {
            time += interval;
            Thread.sleep(Math.max(0, time - System.currentTimeMillis()));
        }
        catch (InterruptedException e)
        {
            break;
        }

        // Calculate distance moved in interval msec and update position of
        // image...
    }
}
```

After initializing `time` with the current time, and setting up the value of `interval`, we have a loop that runs as long as the `bouncer` thread is running. Within the loop we call `repaint()` for the `imagePanel` object to draw the image, and then update the value of `time` to the end of the next interval. The variable `time` will therefore define points in time that are exactly `interval` milliseconds apart. We then call `sleep()` for the animation thread and pass the number of milliseconds remaining between now – determined by calling `currentTimeMillis()` – and the end of the interval, which is the instant specified by the value of `time`. This automatically takes account of the time we spend executing code in the loop to repaint the image and doing other housekeeping. Using the static `max()` method from the `Math` class just ensures that if the repaint and housekeeping took longer than the interval, we pass a zero value to the `sleep()` method so the thread won't sleep at all.

All that's left is for us to work out how far the image moves in `interval` milliseconds.

Dropping the Logo

We will need to know the velocity of the image as it drops and at the end of each time interval. The velocity will increase gradually as it drops due to the force of gravity. We won't worry about the precise physics of this – we are interested in the image handling aspects so we just want to get a nice easy motion for the logo. We will calculate the change in velocity of the image at the end of each time interval as the acceleration multiplied by the time interval in seconds, and the distance traveled as the average velocity times the time interval in seconds. If we assume our units of distance are feet, then the acceleration due to gravity while the logo is dropping is 32 feet per second per second – that is, the velocity increases by 32 feet per second for every second the image drops. We'll also assume the image is stationary to start with so the initial velocity is zero.

The only other thing we need to consider is what happens when the logo hits the ground. The logo image will deform as it is very flexible, and as a consequence will experience a force upwards that increases as it is compressed. We will arbitrarily make this proportional to the degree by which the logo is compressed.

Eventually the downward velocity will be zero and the upward acceleration will then accelerate the image back up again – in other words it will bounce. We can implement this with the following code in the `run()` method:

```
public void run()
{
    long time = System.currentTimeMillis(); // Starting time
    long interval = 20;                    // Time interval msec
    float t = interval/1000.0f;            // and in seconds
    final float g = 32;                    // Acceleration due to gravity
    float a = g;                           // Initial acceleration
    float v = 0.0f;                        // Initial velocity

    // Move image while the bouncer thread is running
    while(Thread.currentThread() == bouncer)
    {
        imagePanel.repaint();              // Repaint the image

        // Wait until the end of the next interval
        try
        {
            time += interval;
            Thread.sleep(Math.max(0, time - System.currentTimeMillis()));
        }
        catch (InterruptedException e)
        {
            break;
        }

        imageY += (long)(t*(v+a*t/2));      // New image position
        v += a*t;                          // New velocity

        // Calculate distance moved in interval
        if(imageY>0)                        // Image compressed?
            a = g - 10.000f*g*imageY/imageHeight; // acceleration in opposite direction

        if(imageY<=0)                      // Image not compressed?
            a = g;                          // -then falling under gravity
    }
}
```


Outside the loop we initialize a variable `t` that stores the time interval in seconds as a value of type `float`. We will use this in our calculations for the image position and velocity. We also define a constant `g`, and initialize the acceleration, `a`, and the velocity `v`, of the image. Within the loop we just apply the calculations that we discussed earlier.

The image is compressed when the `y` coordinate, `imageY`, is positive. This is because the top of the applet is where `y` is 0, and the bottom of the applet is where `y` is `imageHeight`, so when the top-left corner of the image is below the top of the applet, it is being squashed. In this case, if the image is still heading downwards, we apply an acceleration in the opposite direction that is the value of the expression `-10.000f*g*imageY/imageHeight`. This expression is arbitrary, but it has the effect of slowing the image down more and more as it is compressed. When `imageY` is negative, the image is off the ground so we set the acceleration back to `g`.

Displaying the Image

The image is displayed by the `ImagePanel` object, so we need to implement the `paint()` method for this class to display the image normally when `imageY` is zero or negative, and deal with squashing the image when `imageY` is positive. This is going to be easy since we can use the `drawImage()` method we used in the previous example to draw the image normally, and use an overloaded version that is specifically intended for scaling an image on the fly to fit a particular area. The overloaded method has the following arguments:

```
drawImage(Image image,           // The image
          int destinationX,      // x coordinate of the display area
          int destinationY,      // y coordinate of the display area
          int destinationWidth,  // Width of the display area
          int destinationHeight, // Height of the display area
          int imageX,           // x coordinate of the image top left
          int imageY,           // y coordinate of the image top left
          int imageWidth,       // Width of the image
          int imageHeight,      // Height of the image
          ImageObserver observer // The image observer
)
```

When you call this method, the image is scaled on the fly to fit the destination area specified by the arguments. Since you specify the coordinates of the top-left of the image, as well as its width and height, it is possible to use this method to display part of an image, and fit it to the destination space that you specify. Like all the `drawImage()` methods, this method can draw part of an image when loading of the image is not complete. In this case the method returns `false`. The `ImageObserver` that is passed as the last argument – usually the applet itself – is notified when more of the image becomes available, with the result that the image is repainted. When the entire image has been drawn, the `drawImage()` method returns `true`.

It would also be useful to color the background with a color that contrasts with the image. With this in mind we can implement the `paint()` method for the `ImagePanel` class as:

```
public void paint(Graphics g)
{
    Graphics2D g2D = (Graphics2D)g;
```

```

        g2D.setPaint(Color.lightGray);           // Set a background color
        g2D.fillRect(0, 0, imageWidth, imageHeight); // paint background
        if(imageY<=0)
            g2D.drawImage(image, imageX, imageY, this); // Draw normally
        else // or scaled...
            g2D.drawImage(image, // The image
                           imageX, imageY, imageWidth, imageHeight, // Destination
                           0, 0, imageWidth, imageHeight, // Image area
                           this); // Image observer
    }

```

The `fillRect()` method fills the entire area of the applet with the color that we set in the `setPaint()` call, `Color.lightGray`. When `imageY` is greater than 0, we use the version of `drawImage()` that scales the image on the fly to fit the area available, from the `imageY` position to the bottom of the applet at the `y` coordinate, `imageHeight`.

It is worth noting that a `JPanel` object is double-buffered by default. This means that all rendering for a new image that is to be displayed is done in a buffer in memory, and only when image is complete are the pixels for the entire picture written to the screen. Since the existing image that is displayed is not altered while the new image is being created, this eliminates the flicker and flashing that can occur if your display buffer is updated incrementally while it is displayed.

If you have put together all the bits of code we have discussed, you should have a working applet.

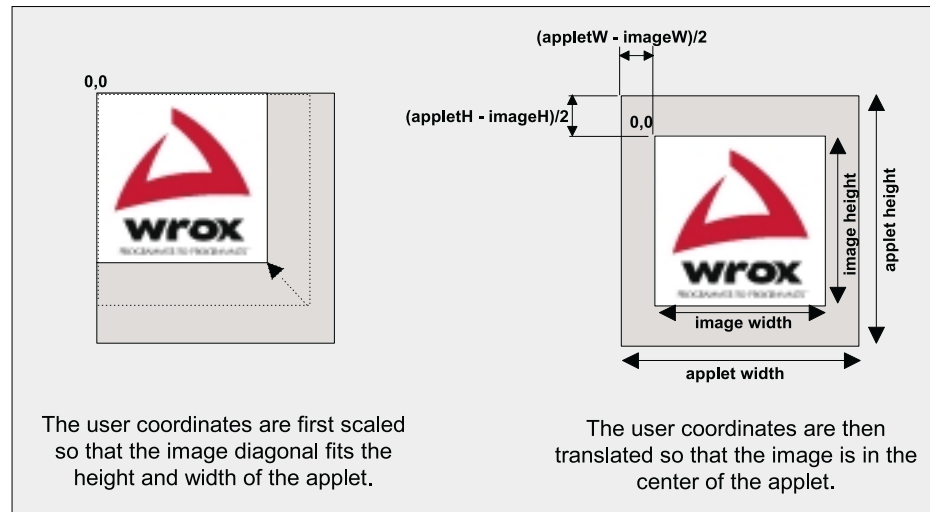
Transforming Images

We have already seen back in Chapter 15 that we can apply a transformation to a graphics context to modify the user coordinate system relative to the device coordinate system. The transformation can be a translation, a rotation, a scaling operation, a shearing operation or a combination of all four. Of course, such a transformation applies to images that you draw as well as anything else.

In our previous example we adjusted the size of the applet to accommodate the image. In many situations you would not want to do this. Typically, there are likely to be all kinds of things on the web page so you would want your applet to keep within the space allotted to it. We could have done this by scaling the image to fit the space available. Rather than go over the old ground let's create a new applet to try this out, and to add a bit of spice this time we will spin the image about its center, rather than dropping it. That way we will get to use a more complicated transform.

Try It Out – Spinning an Image

This applet will create an animation that spins the image about its center point. We will therefore need to make the diagonal of the image fit within both the height and width of the applet if we want to see all of it as it rotates. We also want the scaled image to fit in the center of the applet, so we will translate the user space after we have applied the scale transform.



Once the image is loaded, we can calculate the length of the diagonal of the image as the square root of the sum of the squares of the width and height. We can then calculate the scale factor that we require by dividing the width and height of the applet by the diagonal of the image, and taking the minimum of these two values. We can create an `AffineTransform` object in the `init()` method that combines both the scaling and the translation, and then just apply it in the `paint()` method for the `ImagePanel` class.

This applet class will contain the same methods as the previous example but with different implementations. The rotation will be accomplished by an additional transformation. We will also have an inner `ImagePanel` class to define the panel that will display the image. This will only differ in the implementation of the `paint()` method.

Let's start with the `init()` method and the data members in the `Applet` class. The loading of the image will be identical to the previous example, so we need the `image` and `tracker` members too. After the image has been loaded, we will calculate the scaling and translation that is necessary.

Here's the applet class with its data members and the `init()` method:

```
import java.awt.*;
import java.awt.image.*;
import javax.swing.*;
import java.net.*;
import java.awt.geom.*;           // For AffineTransform

public class WhirlingLogo extends JApplet
    implements Runnable
{
    // This method is called when the applet is loaded
    public void init()
    {
        tracker = new MediaTracker(this);
        Image image = null;
    }
}
```

```
try
{
    // Image from a file specified by a URL
    image = getImage(new URL(getCodeBase(),"Images/wrox_logo.gif"));
}
catch(MalformedURLException e)
{
    System.out.println("Failed to create URL:\n" + e);
}
tracker.addImage(image,0);                // Load image
try
{
    tracker.waitForAll();                  // Wait for image to load
    if(tracker.isErrorAny())               // If there is an error
        return;                           // give up

    Dimension size = getSize();            // Get applet size
    imageWidth = image.getWidth(this);     // Get image width
    imageHeight = image.getHeight(this);   // and its height

    // Calculate scale factor so diagonal of image fits width and height
    double diagonal = Math.sqrt(
        imageWidth*imageWidth + imageHeight*imageHeight);
    double scaleFactor = Math.min(size.width/diagonal, size.height/diagonal);

    // Create a transform to translate and scale the image
    at.setToTranslation((size.width-imageWidth*scaleFactor)/2,
        (size.height-imageHeight*scaleFactor)/2);
    at.scale(scaleFactor,scaleFactor);

    imagePanel = new ImagePanel(image);    // Create panel showing the image
    getContentPane().add(imagePanel);     // Add the panel to the content pane
}
catch(InterruptedException e)
{
    System.out.println(e);
}
}

// Plus the rest of the applet

Thread whirler;                          // Animation thread
boolean whirling = false;                 // Animation control
MediaTracker tracker;                     // Tracks image loading
ImagePanel imagePanel;

AffineTransform at = new AffineTransform();
int imageWidth, imageHeight;              // Image dimensions
double angle;                             // Rotation angle
final int INTERVAL = 50;                  // Time interval msec
final int ROTATION_TIME = 2000;           // Complete rotation time msec
final int STEPS_PER_ROTATION = ROTATION_TIME/INTERVAL;
int stepCount;                            // Total number of steps
}
```

The unshaded code is exactly the same as in the previous method. The `AffineTransform` member, `at`, stores a transform that scales and translates the image. The `angle` member will store the rotation angle in radians that will be calculated in the `run()` method for the `whirler` thread, and applied in the `paint()` method for the `imagePanel` object. We have made this a member of the class rather than declare it as a local variable in the `run()` method so we can pick up the value when the applet is stopped and restarted. The other fields that follow `angle` are all concerned with orienting and drawing the image.

The constant, `INTERVAL`, stores the time interval between one instance of drawing the image and the next. We store the time for a complete rotation of the image through 360 degrees, which is 2π radians, in `ROTATIONTIME`. The variable `STEPS_PER_ROTATION` holds the number of steps for a complete rotation, so with the values we have set for the previous two variables, this will be 40. Finally, the variable, `stepCount`, will accumulate the total number of steps modulo `STEPS_PER_ROTATION`. The methods to start and stop the animation thread are the same as in the previous example, apart from the new names for the thread and the control variable:

```
// This method is called when the browser starts the applet
public void start()
{
    if(tracker.isErrorAny())                // If any image errors
        return;                            // don't create the thread
    whirler = new Thread(this);             // Create the animation thread
    whirling = true;
    whirler.start();                        // and start it
}

// This method is called when the browser want to stop the applet
// when is it not visible for example
public void stop()
{
    whirling = false;                      // Stop the animation loop
    whirler = null;                        // Discard the thread
}
```

The thread code itself will be very similar to the previous example – the timing mechanism is exactly the same. We now need to increment the rotation angle in each time interval so it will be much simpler:

```
// This method is called when the animation thread is started
public void run()
{
    long time = System.currentTimeMillis(); // Starting time

    // Move image while whirling is true
    while(whirling)
    {
        imagePanel.repaint();              // Repaint the image

        // Wait until the end of the interval
        try
        {
            time += INTERVAL;               // Increment the time
            angle = 2.0*Math.PI*stepCount++/ STEPS_PER_ROTATION;
            stepCount %= STEPS_PER_ROTATION;
            Thread.sleep(Math.max(0, time - System.currentTimeMillis()));
        }
    }
}
```

```
        catch (InterruptedException e)
        {
            break;
        }
    }
}
```

The `stepCount` variable starts at 0 and is incremented by 1 on each iteration of the loop. Since a complete rotation through 2 radians should occur after `STEPS_PER_ROTATION` steps, after `stepCount` steps the rotation angle is the result of the expression `2.0*Math.PI*stepCount/STEPS_PER_ROTATION`. In the statement calculating the rotation angle we also increment `stepCount` using the postfix increment operator. The image returns to its original position after `STEPS_PER_ROTATION` steps, so we maintain the value of `stepCount` modulo `STEPS_PER_ROTATION`.

The last bit we need to complete the applet is the `ImagePanel` class, and this only differs from the previous example in the implementation of the `paint()` method:

```
class ImagePanel extends JPanel
{
    public ImagePanel(Image image)
    {
        this.image = image;
    }

    public void paint(Graphics g)
    {
        Graphics2D g2D = (Graphics2D)g;
        g2D.transform(at); // Apply scale & translate

        g2D.setPaint(Color.lightGray);
        g2D.fillRect(0, 0, imageWidth, imageHeight);

        g2D.rotate(angle, imageWidth/2.0, imageHeight/2.0); // Rotate about center

        // draw scaled imaged with background
        g2D.drawImage(image, 0, 0, this);
    }

    Image image; // The image
}
```

That's the complete applet so give it a whirl. It would be a good idea to make the applet dimension larger in the html file – 200x200 say – then you can see the image more clearly. The downside to a larger applet is that it will take more processor time since there are more pixels to process.

How It Works

The basic principles are the same as in the previous example. The thread code in the `run()` method repaints the `imagePanel` object every `interval` milliseconds. The `while` loop that expedites this increments `angle` each time, and `angle` defines the rotation transformation that is applied in the `paint()` method for the `imagePanel` object.

We define the rotation with the statement:

```
g2D.rotate(angle, imageWidth/2.0, imageHeight/2.0);    // Rotate about center
```

The transformation specified by this version of the `rotate()` method is concatenated with the existing transform for the graphics context, which is the `AffineTransform` object that we create in the `init()` method for the applet. The transform created by this `rotate()` call is a composite of a translation to the center of the image, the coordinates of which are defined by the last two arguments, a rotation through `angle` radians – supplied as the first argument, then a translation back to the original origin point. Thus the rotation is about the center of the image.

Using Timers

We have adopted a do-it-yourself approach to timing when we need to redraw in animation. This provides a good insight into how animations operate but we can accomplish the same result rather more simply by making use of an object of the `Timer` class and objects of its associated `TimerTask` class. Both classes are defined in the `java.util` package. The `Timer` class we are discussing here schedules an operation that you define with a `TimerTask` object, either once after a given delay, or repeatedly with a given time interval between successive executions of the task. The task that is executed by the `TimerTask` object runs in a separate thread.

Be aware that there is another `Timer` class defined in `javax.swing` that provides a capability that appears somewhat similar at first sight but that has significant differences in the way that it works. The `Timer` class defined in the `javax.swing` package notifies its listeners (of type `ActionListener`) when a given time interval has passed, and it is up to the listener objects to carry out or initiate the task to be executed. As with the other `Timer` class, you can use a `Timer` object to do something just once after a given interval, or repeatedly after successive intervals of time. A major difference is that the listener object methods will execute on the same thread unless you provide code to ensure that is not the case.

The `Timer` and `TimerTask` combination of classes provide a way of executing repeated tasks that is easy to apply to animations so we will concentrate on those. While we will be applying them to animations here, keep in mind that you can use these methods for executing any kind of task repeatedly or after a fixed delay. We will start by looking at how you use a `Timer` object to schedule a task.

Timer Objects

You can use a single `Timer` object to schedule several different tasks, where each task will be defined by its own `TimerTask` object. Each `TimerTask` object defines a separate thread, so when you schedule multiple tasks they will each execute in a separate thread. The `Timer` class has been designed to allow large numbers of tasks to be executed concurrently – thousands, according to the documentation – without creating undue task scheduling overhead.

There are two constructors for `Timer` objects. The default constructor simply defines a `Timer` object that has an associated thread that is not run as a daemon thread. You will recall that a daemon thread is subordinate to the thread that created it and dies when its creator dies, whereas a non-daemon thread runs completely independently. You can make the `Timer` thread daemon by creating the `Timer` object using the constructor that accepts an argument of type `boolean`, and specifying the argument as `true`. For instance, the following statement creates a `Timer` object with a daemon thread:

```
Timer clock = new Timer(true); // Create a daemon Timer
```

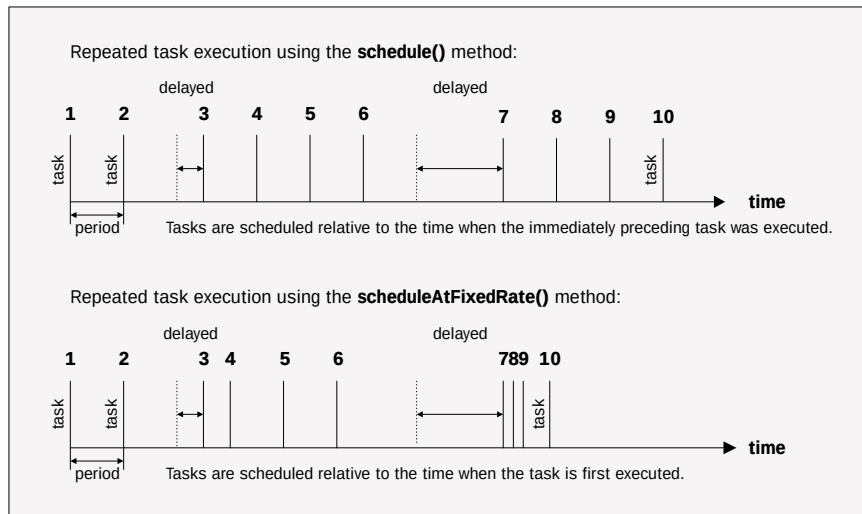
When you have no further need of a `Timer` object, you can terminate it by calling its `cancel()` method. Calling the `cancel()` method for a `Timer` object terminates all tasks scheduled by the timer, and terminates the timer's thread, so you cannot use the object again for scheduling tasks.

A `Timer` object provides you with two methods for scheduling tasks, the `schedule()` method and the `scheduleAtFixedRate()` method. Both of these methods come in overloaded flavors, so let's look at the `schedule()` method first.

The `schedule()` method is for executing a task either once at a given instant in time, or repeatedly with each subsequent execution starting after a fixed delay relative to the previous task. If any particular execution is delayed, subsequent executions will be delayed. This mode of repeated task execution is referred to as **fixed-delay execution** because priority is given to maintaining the time interval between task executions, rather than scheduling each execution at a precise time. This is suitable for applications where the priority is for repeated executions of a task to be evenly distributed rather than being at fixed points in time. Animations fall into this category since you will generally want to have the animation as smooth as possible. You have four version of the `schedule()` method available:

Method	Description
<code>schedule(TimerTask task, Date time)</code>	This schedules the task determined by the first argument, <code>task</code> , to be executed once at the time instant specified by the second argument, <code>time</code> . If the current time is later than the time specified by <code>time</code> , then the task executes immediately.
<code>schedule(TimerTask task, Date firstTime, long period)</code>	This schedules the task determined by the first argument, <code>task</code> , to be executed repeatedly starting at the time specified by the second argument, <code>firstTime</code> . The third argument, <code>period</code> , specifies the period in milliseconds between the start time of one execution of the task and the start time of the next. If the current time is later than the time specified for the first execution of the task, then the task executes immediately.
<code>schedule(TimerTask task, long delay)</code>	This schedules the task determined by the first argument, <code>task</code> , to be executed once after a delay relative to the current time of <code>delay</code> milliseconds.
<code>schedule(TimerTask task, long delay, long period)</code>	This schedules the task determined by the first argument, <code>task</code> , to be executed repeatedly with the first execution starting after a delay relative to the current time of <code>delay</code> milliseconds. The third argument, <code>period</code> , specified the period in milliseconds between the start time of one execution of the task and the start time of the next.

You use the `scheduleAtFixedRate()` method for repeated executions of a task where the precise timing is more important than maintaining the interval between successive executions. This is referred to as **fixed-rate execution**. Each execution is scheduled relative to the first execution of the task, not the preceding one. If you wanted to simulate a clock for instance using `Timer` and `TimerTask` objects you would use the `scheduleAtFixedRate()` method to schedule updating the position of the hands on the clock rather than the `schedule()` method because you want the hand positions to be set as close as possible to absolute time. If any execution of the update to the hand position is delayed for any reason, succeeding executions will 'bunch-up' in time in order to try to maintain their schedule in real time.



This is shown graphically in the diagram, which contrasts the two types of scheduling operations you can use.

You have two versions of the `scheduleAtFixedRate()` method available to you, both of which are for scheduling a task repeatedly:

Method	Description
<code>scheduleAtFixedRate</code> <code>(TimerTask task,</code> <code> Date firstTime,</code> <code> long period)</code>	This schedules the task determined by the first argument, <code>task</code> , to be executed repeatedly starting at the time specified by the second argument, <code>firstTime</code> . The third argument, <code>period</code> , specifies the period in milliseconds between the start time of one execution of the task and the start time of the next. If the current time is later than the time specified for the first execution of the task, then the task executes immediately.
<code>scheduleAtFixedRate</code> <code>(TimerTask task,</code> <code> long delay,</code> <code> long period)</code>	This schedules the task determined by the first argument, <code>task</code> , to be executed repeatedly with the first execution starting after a delay relative to the current time of <code>delay</code> milliseconds. The third argument, <code>period</code> , specifies the period in milliseconds between the start time of one execution of the task and the start time of the next.

Both the `schedule()` and `scheduleAtFixedRate()` methods can throw exceptions. An exception of type `IllegalArgumentException` will be thrown if a delay argument is negative or if an argument of type `Date` represents a negative time value (as returned by its `getTime()` method). An exception of type `IllegalStateException` will be thrown if the task was already scheduled or if the task or the timer was cancelled.

Let's turn to how we use the `TimerTask` class to define a task to be scheduled by a `Timer` object.

TimerTask Objects

`TimerTask` is an abstract class, so you will need to derive your own class from it. The class implements the `Runnable` interface so a `TimerTask` object defines a thread. The `run()` method in the `TimerTask` class is abstract because it is this method that specifies the task to be executed and it's your job to decide this. Of course, you can define your class with `TimerTask` as a base in its own source file, but more often than not you will want to define it using an anonymous class. The form of the code for scheduling such a task will be something like this:

```
TimerTask task = new TimerTask()
{
    public void run()
    {
        // Code defining the task to be executed.
    }
}
```

To schedule the task that this creates for repeated execution at one second intervals starting five seconds from now we could write:

```
Timer timer = new Timer(true);           // Create a timer with a daemon thread
timer.scheduleAtFixedRate(task, new Date(System.currentTimeMillis()+5000), 1000);
```

The `currentTimeMillis()` method returns the current time from the system clock in milliseconds. We add 5000 milliseconds to this to specify the instant five seconds from now. Since we call the `scheduleAtFixedRate()` method here to schedule the task, the fixed-delay execution method will apply.

The `TimerTask` class contains just one other method besides the `run()` method – the `cancel()` method, which you call to cancel the execution of a task. Calling the `cancel()` method for a given `TimerTask` object permanently stops execution of that task, whether it has been scheduled for one-time execution or repeated execution. For example:

```
task.cancel();           // Terminate the task
```

The task will be terminated and cannot be run again. If you want to run the task again you need to create a new `TimerTask` object and schedule that for execution. If a task has been canceled previously, calling `cancel()` again will have no effect. The `cancel()` method for a `TimerTask` object provides you with control at a task level. As we said earlier, you can use a `Timer` object to schedule several different tasks, each of which will be defined by an object that has `TimerTask` as a superclass. Calling the `cancel()` method for an individual task will terminate the execution of that task without affecting any others.

When you want to terminate all the tasks currently managed by a `Timer` object you just call the `cancel()` method for the `Timer` object. For instance:

```
timer.cancel();           // Terminate the timer
```

This terminates all tasks scheduled by `timer`, and also terminates the `Timer` object's thread so it cannot be used again.

Let's rewrite the previous `WhirlingLogo` example to use a `Timer` object.

Try It Out – Using a Timer

Much of the code will be the same so we will only repeat the essentials here. The class no longer needs to implement the `Runnable` interface so the `run()` method is no longer required in the applet class.

```
import java.awt.*;
import java.awt.image.*;
import javax.swing.*;
import java.net.*;
import java.awt.geom.*;          // For AffineTransform

public class TimedWhirlingLogo extends JApplet
{
    // This method is called when the applet is loaded
    public void init()
    {
        // Code exactly as before...
    }

    // This method is called when the browser starts the applet
    public void start()
    {
        if(tracker.isErrorAny())          // If any image errors
            return;                      // don't create the thread

        timer = new java.util.Timer(true);
        timer.schedule(new java.util.TimerTask()
        {
            public void run()
            {
                imagePanel.repaint();      // Repaint the image
                angle = 2.0*Math.PI*stepCount++/ STEPS_PER_ROTATION;
                stepCount = ++stepCount%STEPS_PER_ROTATION;
            }
        },
        0, INTERVAL);
    }

    // This method is called when the browser wants to stop the applet
    // - when is it not visible for example
    public void stop()
    {
        timer.cancel();
    }
}

// Class representing a panel displaying an image
```

```
class ImagePanel extends JPanel
{
    // Code exactly as before...
}

java.util.Timer timer;           // Animation timer
MediaTracker tracker;           // Tracks image loading
ImagePanel imagePanel;
AffineTransform at = new AffineTransform();
int imageWidth, imageHeight;     // Image dimensions
double angle;                   // Rotation angle
final int INTERVAL = 50;        // Time interval msec
final int ROTATION_TIME = 2000; // Complete rotation time msec
final int STEPS_PER_ROTATION = ROTATION_TIME/ INTERVAL;
int stepCount;                  // Total number of steps
}
```

If you compile and run this applet, it should run just as well as the previous version.

How It Works

The code is a lot shorter because the `Timer` object does all the scheduling work. The `start()` method in our applet class creates the `Timer` object we will use to schedule the animation with the statement:

```
timer = new java.util.Timer(true);
```

The variable, `timer`, is a member of the `TimedWhirlingLogo` class rather than a variable local to the `start()` method because we also need to reference it in the `stop()` method. Note how we have used the fully qualified name for the `Timer` class here, and in the declaration of `timer` as a member of the applet class. This is essential in this case. Importing the package, `java.util`, containing the `Timer` class would not be sufficient. As we said earlier, the `javax.swing` package also defines a class with the name, `Timer`, so without qualification of the name, the compiler would be unable to decide which class we wanted to use.

We then use the `Timer` object to schedule the animation with the statement:

```
timer.schedule(new java.util.TimerTask()
{
    public void run()
    {
        imagePanel.repaint();           // Repaint the image
        angle = 2.0*Math.PI*stepCount++/ STEPS_PER_ROTATION;
        stepCount = ++stepCount%STEPS_PER_ROTATION;
    }
},
0, INTERVAL);
```

We use the `schedule()` method here because we need the task to be executed at evenly distributed intervals to get a smooth animation. The first argument to the `schedule()` method is defined by an anonymous class derived from `TimerTask`. We again use a fully qualified class name here – not because there is a duplicate use of the name, but because we have not imported the `java.util` package into our source file. The `run()` method in our anonymous class specifies the task to be executed. This consists of two steps: redrawing `imagePanel` containing the logo, and updating `angle` that determines the orientation of the logo next time around. The second argument to `schedule()` specifies a delay of zero milliseconds before the first execution of the task, so the animation will begin immediately. The third argument specifies the interval between successive frames of the animation. Defining the constant, `INTERVAL`, as a member of the applet class enables it to be referenced as an argument to the `schedule()` method and within the `run()` method of the anonymous class.

That's it. Using `Timer` objects makes scheduling animations a lot simpler. Let's try one more example to get a feel for using the `scheduleAtFixedRate()` method.

Try It Out – Fixed-Rate Task Execution

To make use of fixed-rate execution scheduling we'll create an applet that is a clock. The graphics will be much more complicated than the scheduling, but we will have an opportunity to explore yet another way of handling animation. It's quite a lot of code so we'll put it together piece by piece, starting with the applet class.

The basic code for the applet class will be like this:

```
import java.awt.*;
import javax.swing.*;
import java.awt.geom.*;
import java.util.*;

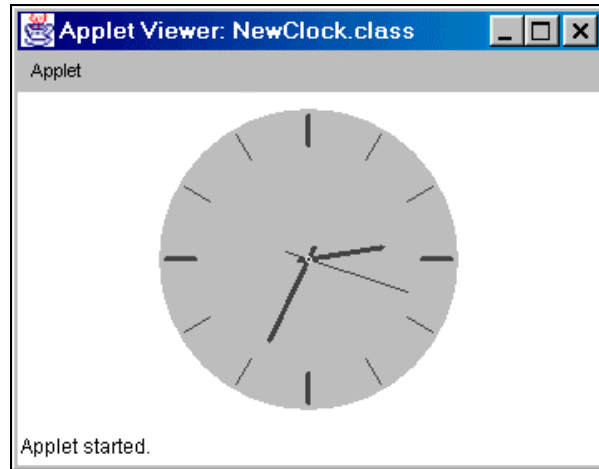
public class NewClock extends JApplet
{
    // This method is called when the applet is loaded
    public void init()
    {
        // Initialize the applet and set up the clock
    }

    // This method is called when the browser starts the applet
    public void start()
    {
        // Start the clock
    }

    // This method is called when the browser wants to stop the applet
    public void stop()
    {
        // Stop the clock
    }
}
```

The class has the name `NewClock` to differentiate from the myriad of other clock programs that are around. This class just contains the three basic methods that we will need to implement for our applet. We won't need to implement the `paint()` method as we will be adding the representation of the clock to the applet object and the clock will take care of drawing itself.

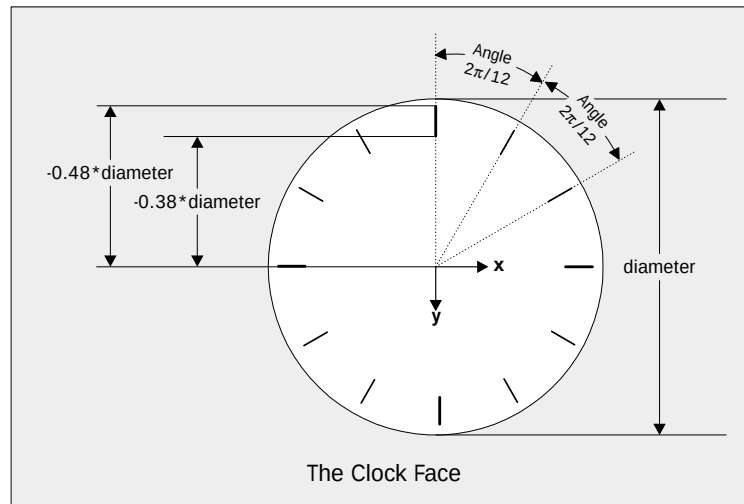
The clock when it is running will look as shown in the illustration.



We can consider the clock to be made up of two parts. One part is the face, consisting of the circular dial plus the hour marks, and which is static. The other part is the hands, consisting of the hour, minute, and second hands, plus the central boss holding them in place. This is the bit that is dynamic and has to be updated. By separating the hands from the face of the clock, we can write our applet so that we only need to update the hands as time passes, and avoid having to redraw the face each time we update the time shown on the clock. We can define the two parts of the clock by inner classes to the `NewClock` class. We shall bring all our creativity to bear and name these classes `ClockFace` and `Hands`. Let's define the first one first.

Defining the Clock Face

We can base our `ClockFace` class on `JPanel`. The clock face will need to be sized to fit within the applet so we will construct a `ClockFace` object with a given diameter. The basic geometry of the clock face is shown in the diagram.



We can create the circular face as a filled circle, which will be an ellipse object of type `Ellipse2D.Double` with the major and minor axes the same dimension, `diameter`. We want the hour marks to fit just inside the face and the dimensions shown as a proportion of the diameter will suit. We can create all the hour marks around the face very easily from a single vertical line of the appropriate length at the 12 o'clock position. We just need to repeatedly rotate the axes by one twelfth of 2π and redraw the line at a further eleven positions around the dial. We will define the line for the hour mark as an object of type `Line2D.Double`. Based on those ideas, here's the definition of the inner class:

```
// Class defining the static face of the clock
class ClockFace extends JPanel
{
    // Creates a clock face with the given diameter
    public ClockFace(int diameter)
    {
        this.diameter = diameter;
        face = new Ellipse2D.Double();
        hourMark = new Line2D.Double(0, -diameter*0.38,
                                     0, -diameter*0.48);
        setOpaque(false); // Set panel transparent
    }

    public void paint(Graphics g)
    {
        Dimension size = getSize();
        face setFrame((size.width-diameter)/2, // Set the size
                     (size.height-diameter)/2, // of the face centered
                     diameter, diameter);      // and of the given diameter

        Graphics2D g2D = (Graphics2D)g;

        // Clear the panel
        g2D.setPaint(CLEAR); // Transparent color
        g2D.fillRect(0,0,size.width,size.height); // Fill the background

        g2D.setPaint(Color.lightGray); // Face color
        g2D.fill(face); // Fill the clock face
        g2D.setPaint(Color.darkGray); // Face outline color
        g2D.setStroke(widePen); // Use wide pen
        g2D.draw(face); // Draw face outline

        // Move origin to center of face
        g2D.translate(size.width/2, size.height/2);

        // Paint hour marks
        for(int i = 0 ; i<12 ; i++)
        {
            if(i%3 == 0)
                g2D.setStroke(widePen); // Wide pen each quarter position
            else
                g2D.setStroke(narrowPen); // otherwise narrow pen

            g2D.draw(hourMark); // Draw the hour mark
            g2D.rotate(TWO_PI/12.0); // Rotate to next mark
        }
    }

    int diameter; // Face diameter
    Ellipse2D.Double face; // The face
    Line2D.Double hourMark; // Mark for hours
}
```

Note that the various shades of gray used here reflect the needs of printing in the book, rather than any somber character trait on my part. You may like to jazz the example up a bit with your own choice of colors.

The face of the clock will be a filled ellipse with major and minor axes equal. We create a default `Ellipse2D.Double` object for this in the constructor, because we will set up the dimensions of the ellipse based on the size of the panel in the `paint()` method. We do this by calling the `setFrame()` method for the ellipse where the first two arguments are the coordinates of the top-left of the enclosing rectangle, and the last two are the dimensions' major and minor axes.

In the constructor, we make the hour mark object a vertical line of type `Line2D.Double` with start and end points such that it fits just within the diameter of the face, and leaves enough room centrally for the hands. We draw the hour marks in the `paint()` method by applying a transform that rotates the axes by $2\pi/12$ between drawing one mark and the next. The marks on quarters need to be a different thickness to the others and for this we use a couple of `BasicStroke` objects that defines lines with specific characteristics. Whenever you draw a shape – a line, an ellipse or whatever, the characteristics of the line that is produced are determined by a `Stroke` object that is stored within the `Graphics2D` object. So far we have just been using the default setup, but you can define your own line types. We haven't met this before, so let's take a brief detour into how strokes work.

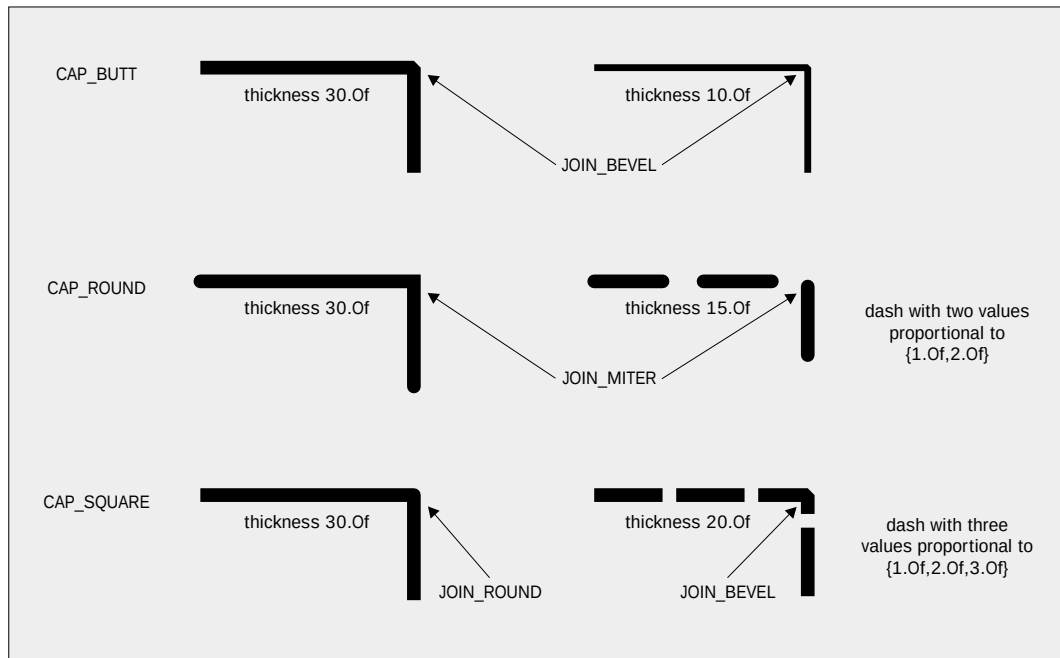
Different Strokes for Different Folks

The type of line that applies by default when you draw any shape in a `Graphics2D` context is a solid line with square ends and a line width of 1. By calling the `setStroke()` method for the `Graphics2D` object, you can change the type of line produced to whatever you want. The `setStroke()` method expects an argument of type `Stroke`, but `Stroke` is actually an interface and the class that defines line types and implements the `Stroke` interface is `BasicStroke`. A `BasicStroke` object defines a line in terms of four attributes:

Attribute	Description
Pen Width	This is the width of the line expressed as a value of type <code>float</code> .
End Caps	This determines what the end of a line looks like, and you specify it by one of the following three constants, which are of type <code>int</code> : <code>CAP_BUTT</code> – the end of the line is flat with nothing added. <code>CAP_ROUND</code> – a semi-circular end is added with a radius half the width of the line. <code>CAP_SQUARE</code> – a square end is added that extends the line by half the width. A default line has this end cap.
Joins	This specifies how connected line segments in a path join, using one of three <code>int</code> values: <code>JOIN_BEVEL</code> – The outer corners of the two line segments are connected by a straight edge. <code>JOIN_MITER</code> – The outer edges of both segments are extended at the join until they meet. There is a limit to the distance over which mitered joins will be made (the default is 10.0). <code>JOIN_ROUND</code> – The outer edges of the segments are connected by a circular arc with a radius of half the width of the line.

Attribute	Description
Dash pattern	You specify this by an array of values of type float that define a dashing pattern. The values alternate between the length of a dash in user coordinates and the length of space before the next dash. Clearly a minimum of two elements are necessary to define a typical dashed line, one for the line and one for the space, but there can be more. Another float value defines the distance from the start of a line where the dashing pattern starts. If this value is non-zero, then your line starts with a space rather than a dash. A default line is solid.

The diagram shows the effects of some of these parameters for a **Stroke** object.



I created these by drawing a path six times in various positions. The lengths of the lines and the dash lengths were defined relative to the width of the area in which they were drawn, so the dash lengths are described in terms of relative proportions.

There are five constructors to define a **BasicStroke** object. The default constructor defines a default type of line. This corresponds to a line width of 1, **CAP_SQUARE** as the end and **JOIN_MITER** for joins with a miter distance limit of 10. The other four constructors have more arguments supplying values to replace the defaults, as follows:

```
BasicStroke(float width)
BasicStroke(float width, int cap, int join)
BasicStroke(float width, int cap, int join, float miterLimit)
BasicStroke(float width, int cap, int join, float miterLimit,
            float[] dashPattern, float offset)
```

Whatever `Stroke` you set in a graphics context applies to all subsequent shapes that you draw – until you set it to a different line type. Of course, there's a lot of potential for applying this in Sketcher. That said, we can return to drawing our clock.

The Stroke of Midnight

We want a `BasicStroke` object that we can use to draw a thicker line at the 12 o'clock position, as well as at 3, 6 and 9. We can define this with the following statement:

```
BasicStroke widePen = new BasicStroke(3.0f,           // Line width
                                     BasicStroke.CAP_ROUND, // End cap
                                     BasicStroke.JOIN_MITER); // Line join
```

Since we will be able to make use of the `widePen` object in the other inner class that will define the hands on our clock, we will include this object as a member of the applet class. Add this statement to the end of the `NewClock` class definition, following the code for the inner class, `ClockFace`. Note that we also draw the outline of the ellipse using this pen after we have created the filled ellipse.

For the intermediate hour marks we need a narrower line, which we can define with the statement:

```
BasicStroke narrowPen = new BasicStroke(1.0f,
                                     BasicStroke.CAP_ROUND,
                                     BasicStroke.JOIN_MITER);
```

This has a line width of `1.0f`, and the same end cap and join specification as `widePen`. We can add this as a member of the applet class, too.

Back in the `paint()` method for the `ClockFace` inner class, you can see that we set one or other of these `BasicStroke` objects depending on which hour mark we happen to be drawing in the loop. When the loop counter `i` is an exact multiple of 3, we are drawing one of the four heavy marks so we set `widePen` to be the `Stroke` object. The rest of the time we use `narrowPen`.

Note that in the `ClockFace` constructor, we call the `setOpaque()` method with the argument `false`. This method is inherited in our class from `JComponent` class. This call makes the clock face panel transparent, so only the face will obscure whatever is behind the clock.

Let's now turn to the inner class that will define the hands.

Defining the Hands on the Clock

Our clock will have three hands, a second hand, a minute hand, and an hour hand. To ensure the hands always fit within the face, the length of each hand will be defined in terms of the diameter of the clock face, and this dimension will be passed as an argument to the `Hands` constructor. To distinguish the hands, we will draw the second hand using `narrowPen` and the other two hands using `widePen`. To make the clock look more realistic, we will also add a central boss holding the hands on their spindle.

The class will need to know the angular position of each hand. We can store these three values in data members, `secondAngle`, `minuteAngle`, and `hourAngle`, which will be of type `double`, and we will assume the angles are measured in radians from the 12 o'clock position – clockwise of course. We could set these values by passing them to the constructor, but because we are likely to want to reset the positions of the hands whenever the `start()` method for the applet is called, it will be more convenient to add a method, `setPosition()` to the `Hands` class to provide for this.

There are various approaches that we could use to maintain the correct time and perhaps the best and most accurate would be to make the `Hands` object fetch the current time from the system clock each time the hands are redrawn. However, this would make the accuracy of the clock independent of our `Timer` object and we really want to try that out. Our clock will operate with our `Timer` object maintaining the time. In order to maintain the correct position of the hands over time, a `Hands` object will need to be able to increment the position of each hand when required. We could implement a method that provided a completely general increment capability, but for the sake of simplicity we will code the method to increment all three hands assuming one second has passed.

Here's the code for the `Hands` inner class to the `NewClock` class:

```
// Class defining the hands on the clock
class Hands extends JPanel
{
    public Hands(int clockDiameter)
    {
        center = new Ellipse2D.Double(-3,-3,6,6);           // Central boss
        hourHand = new Line2D.Double(0,6,0,-clockDiameter*0.25);
        minuteHand = new Line2D.Double(0,8,0,-clockDiameter*0.3);
        secondHand = new Line2D.Double(0,14,0,-clockDiameter*0.35);
        setOpaque(false);                                     // Set transparent
    }

    // Paint the hands
    public void paint(Graphics g)
    {
        // Get hand angles for the current time
        double secondAngle = seconds*TWO_PI/60;
        double minuteAngle = (secondAngle+minutes*TWO_PI)/60;
        double hourAngle = (minuteAngle+hours*TWO_PI)/12;

        Dimension size = getSize();
        Graphics2D g2D = (Graphics2D)g;
        g2D.setPaint(CLEAR);                                  // Transparent color
        g2D.fillRect(0,0,size.width,size.height);           // Fill to erase

        g2D.setPaint(Color.darkGray);                        // Hands color
        g2D.translate(size.width/2, size.height/2);          // Origin to center
        AffineTransform transform = g2D.getTransform();      // Save this xform

        // Draw hour hand
        g2D.setStroke(widePen);                               // Use wide pen
        g2D.rotate(hourAngle);                                // Rotate to hour position
        g2D.draw(hourHand);                                   // and draw hand

        // Draw minute hand
        g2D.setTransform(transform);                          // Reset transform
        g2D.rotate(minuteAngle);                              // Rotate to minute position
        g2D.draw(minuteHand);                                 // and draw hand

        // Draw second hand
        g2D.setStroke(narrowPen);                             // Use narrow pen
        g2D.setTransform(transform);                          // Reset transform
        g2D.rotate(secondAngle);                              // Rotate to second position
        g2D.draw(secondHand);                                // and draw hand
    }
}
```

```

        g2D.setPaint(Color.white);           // Center color
        g2D.draw(center);                   // Draw center
    }

    Line2D.Double hourHand;
    Line2D.Double minuteHand;
    Line2D.Double secondHand;
    Ellipse2D.Double center;
    final Color CLEAR = new Color(0,0,0,0);
}

```

The constructor is quite straightforward. The central boss holding the hands on is the `Ellipse2D.Double` object, `center`. This is a circle with a diameter of 6. After defining the lines representing the hands, we call `setOpaque()` for the panel to make the panel transparent.

The `paint()` method looks like a lot of code, but after calculating the angular position for each hand, it is just a series of separate drawing operations. The position of each hand is basically a proportion of 2π . Each second or minute contributes one sixtieth of 2π to the position of the corresponding hand and each hour is one twelfth. To the angular position of the minute hand we add the contribution that the angle of the second hand represents as a fraction of a minute, and for the hour hand we add the contribution of the minute hand. After getting the size of the panel, we fill the entire panel with the color `CLEAR`. This is to erase the previous instance of the hands drawn on the panel. This `Color` object is defined as a constant member of the class. The first three arguments to the `Color` constructor define the red, green and blue color components of the color to be zero. The fourth argument defines something called the **alpha component** for the color as zero, which defines this color as completely transparent. We will go into the significance of the alpha component in more detail later in this chapter. We'll just use it for now. We want the color to be transparent because the `HANDS` panel will be displayed on top of the `ClockFace` panel in the applet, and we want the face to be visible.

After moving the origin to the center of the panel we save the current transform. This will enable us to restore this position after rotating the axes to position each hand. Each hand is drawn in essentially the same way. The `Stroke` for the hand is set, then the axes are rotated to the required angle, and finally we draw the line representing the hand. The last step after drawing the three hands is to draw the central boss in white.

Now we have the inner classes for the face and hands defined, we are ready to complete the applet.

Initializing the Applet

In the `init()` method we will need to create the `ClockFace` object and the `Hands` object that will make up the clock. We just have to decide how the hands are going to overlay the clock face.

Back in Chapter 12 we saw how a `JFrame` object had several window panes, including a **content pane**, to which you typically add the components you want to display, plus a **glass pane** that overlays the content pane. An object of type `JApplet` has exactly the same pane structure. We can add an object of type `ClockFace` to the content pane for the applet, and make the `Hands` object the glass pane. As long as it's transparent, the content pane with its `ClockFace` component will be visible underneath the glass pane – which will be our `Hands` object. In fact, in general, a glass pane can be any object of a class that has `Component` as a superclass. To replace the default glass pane with your component, you just call the `setGlassPane()` method for the applet object with a reference to your component as the argument. Here's the code for the `init()` method that will set our clock up like that:

```

public void init()
{
    Dimension size = getSize();           // Get the applet size

    // Create the clockface to fit within the applet
    int clockDiameter = Math.min(size.width, size.height)*9/10;
    clock = new ClockFace(clockDiameter);
    getContentPane().add(clock);         // Add clockface to content pane

    hands = new Hands(clockDiameter);     // Create the hands panel
    setGlassPane(hands);                 // Make the hands the glass pane
    hands.setVisible(true);              // Set glass pane visible
}

```

We get the size of the applet by calling `getSize()`, and use that to decide the diameter of the clock. Ninety percent of the smaller of the width and height of the applet is a suitable choice to make it fit comfortably. Once we have created the `ClockFace` object, we just add it to the content pane for the applet. We then create the `Hands` object and pass it to `setGlassPane()` to make it the glass pane for the applet. Note that we must call `setVisible()` for the glass pane because it will be set as invisible by default.

That's all that's necessary to create the visual appearance of the clock. We just need to set it going somewhere, and that's the job of the `start()` method for the applet object.

Starting the Clock

The operation of the clock will be controlled by a `Timer` object, but before it starts the hands should be set to correspond to the current time. The static `getInstance()` method in the `Calendar` class that we discussed back in Chapter 10 returns a reference to a `Calendar` object that corresponds to the current time recorded in the system clock. We can then use the `get()` method to get the second, minute and hour values for the current time as integers. We can use these to figure out the angles for the hand positions and then call the `setPosition()` method for the `Hands` object. Here's how that can be coded:

```

public void start()
{
    Calendar now = Calendar.getInstance(); // Calendar for this instant

    // Get current seconds, minutes, and hours
    seconds = now.get(now.SECOND);
    minutes = now.get(now.MINUTE);
    hours = now.get(now.HOUR);

    timer = new java.util.Timer(true);    // Timer to run clock

    // Use fixed-rate execution to maintain time
    timer.scheduleAtFixedRate(new TimerTask()
    {
        public void run()
        {
            increment();                // Increment the time
            hands.repaint();            // and repaint the hands
        }
    },

```

```

        0,                                // ...starting now
        ONE_SECOND);                      // and at one second intervals
    }

```

The constants `TWO_PI` and `ONE_SECOND` need to be added as members of the applet class along with the `timer` field. We must also add the fields to store the time. The following statements will do that:

```

final double TWO_PI = 2.0*Math.PI;
final int ONE_SECOND = 1000;        // One second in milliseconds
java.util.Timer timer;              // Timer to control the clock
int seconds, minutes, hours;        // The current time

```

After saving the current time as seconds, minutes and hours, we create a `Timer` object to control the clock. We use the `scheduleAtFixedRate()` method to run the clock because we want the clock to be updated at intervals of one second, and if an update is delayed for any reason, we don't want subsequent updates to be delayed. The `TimerTask` object that is defined by an anonymous class calls the `repaint()` method for the `Hands` object, then the `increment()` method to update the time to the next second.

We can implement the `increment()` method in the `NewClock` class like this:

```

// Increment the time by one second
public void increment()
{
    if(++seconds>= 60)                // If seconds reach 60
        ++minutes;                  // ...increment minutes

    if(minutes>=60)                  // If minutes reach 60
        ++hours;                    // ...increment hours

    seconds %= 60;                    // Seconds 0 to 59
    minutes %= 60;                    // Minutes 0 to 59
    hours %= 12;                      // Hours 0 to 11
}

```

When we have accumulated 60 seconds after incrementing the time, we must increment the count of the number of minutes by one. Similarly, when minutes reaches 60, we must increment the hour count.

Stopping the Clock

Stopping the clock is just a question of canceling the timer:

```

public void stop()
{
    timer.cancel();                  // Cancel the clock timer
}

```

How It Works

While this example provides an illustration of using the fixed-rate execution mode with a timer, the more interesting aspect is the use of the glass pane to hold the animated portion of an image. You can apply this technique to any animation with any component that has a content pane and a glass pane. This includes components defined by the `JWindow`, `JDialog`, `JFrame` and `JInternalFrame` classes, as well as the `JApplet` class as we have just seen.

Alpha Compositing

Alpha compositing is all about the transparency of an image, and we are just going to look at the basic ideas here. The subject of color representation is itself a major topic with a range of different ways of representing color, but we will assume we are dealing with the RGB model throughout and ignore other color models. Note that Java does provide much more extensive support for color modeling than we have the space to discuss in this book, so if you want to know more, look in the classes of the `java.awt.color` package.

When you want to define an image to be transparent to some degree, so that when it is displayed on top of a background image the underlying image shows through, you can specify the degree of transparency by something called an **alpha component** for each pixel. An image format that has an alpha component for each pixel is said to have an **alpha channel**. Note that not all image formats support an alpha channel, but PNG files do, and GIF images can have a single transparent color. The alpha component value is used when an image is being overlaid by another image. The alpha value is multiplied by each of the color components to modify the contribution of each color component to the visual appearance of the pixel. The alpha value for a pixel in an image can vary from a minimum of 0.0, meaning completely transparent and therefore invisible since all the color components will be 0, to a maximum of 1.0, meaning completely opaque. In an RGB image with an alpha channel, each pixel is defined by four components, the three color components, red, green and blue, plus the alpha component. This allows the transparency to vary over the image, so some parts of the image could be opaque – with an alpha component of 1.0, and other parts may be more or less transparent, with alpha components for the pixels less than 1.0. It's worth noting that both the source image, the image that you are drawing, and the destination image, the background in other words, can have an alpha channel. If an image has no alpha channel, then the alpha component is assumed to be 1.0.

When you draw a source image over a destination image, there are basically two steps to the process. The color components for each pixel in the source image and the corresponding pixels in the destination image will be multiplied by their alpha component – often this will be done once and for all ahead of time for an image to avoid all those multiplications each time you draw an image. The source image is then rendered over the destination image according to the **alpha compositing rule** that is in effect. There are several alpha compositing rules as we shall see, but they each determine the fraction of the source image components and the fraction of the destination image components that contribute to the components of the result. In general, the components of the source and destination pixels are combined as follows:

$$\begin{aligned}\text{ColorR} &= \text{ColorS} * \text{AlphaS} * \text{FractionS} + \text{ColorD} * \text{AlphaD} * \text{FractionD} \\ \text{AlphaR} &= \text{AlphaS} * \text{FractionS} + \text{AlphaD} * \text{FractionD}\end{aligned}$$

The first equation applies to each of the three color components. The subscripts R, S and D refer to the resultant pixel, the source pixel and the destination pixel, respectively. Thus, `ColorR` refers to the color of the resultant pixel produced by combining the source and the destination, `AlphaS` refers to the alpha component for the source pixel and `FractionS` represents the fraction of the source pixel determined by the compositing rule in effect.

This sounds a lot more complicated than it really is, so don't be put off by these equations. The alpha compositing rules that you can use in Java are implemented by the `AlphaComposite` class that is defined in the `java.awt` package. The `Graphics2D` class defines the method `setComposite()` that takes an `AlphaComposite` argument in order to set the alpha compositing rule to be used when you draw in the graphics context. Let's take a look at the `AlphaComposite` class in more detail.

The AlphaComposite Class

There is no constructor for the `AlphaComposite` class, so you cannot create objects directly. There is a static class member, `getInstance()`, that will return a reference to an `AlphaComposite` object with the compositing rule specified by the argument, which is a value of type `int`. There is also an overloaded version of this method where you can specify an alpha value as a second argument of type `float`, which is multiplied by the alpha for the source image. This is particularly useful when the source image has no alpha channel. Since an image with no alpha channel has an alpha component that is assumed to be 1.0, the alpha value that you specify in the call to `getInstance()` becomes the alpha value for all the pixels in the source. Most of the time, your images will not have an alpha channel so this is a way for you to specify the transparency of the source image directly in the graphics context.

There are eight possible alpha compositing rules, determined by constants of type `int` that are defined in the `AlphaComposite` class. In reviewing these, we will assume that, if the source or destination image has an alpha channel, the color components, `ColorS` and `ColorD`, for each pixel have already been pre-multiplied by the alpha component, `AlphaS` or `AlphaD` respectively. In the illustration of the effect of each rule described below, the source image is the light gray circle, and this is rendered over the darker gray rectangle (the destination image). The source image has its alpha component set to 0.5f.



SRC_OVER

This is the default rule that applies in a graphics context and is the rule that you are most likely to be using. The fraction of the source that contributes to the result is 1, and the fraction of the destination contributing to the result is $1 - \text{AlphaS}$. Therefore from our general equations, the source pixels are combined with the corresponding destination pixels using the following operations:

$$\begin{aligned}\text{ColorR} &= \text{ColorS} + (1 - \text{AlphaS}) * \text{ColorD} \\ \text{AlphaR} &= \text{AlphaS} + (1 - \text{AlphaS}) * \text{AlphaD}\end{aligned}$$

The calculation of the resultant color is applied to each of the red, green and blue components of each pixel. You can see from the equations above that if the alpha component for the source, `AlphaS`, is 1, then the fraction of the destination will be zero so the result is just the original source pixel – in other words the source is opaque. If the alpha component for the source is 0, then the result is just the destination pixel so the source is completely transparent and would be invisible. The illustration shows the source with an alpha of 0.5f so the destination shows through.



SRC

With this rule, the source pixels replace the destination pixels, so the operations determining the resultant color and alpha components for each pixel are:

$$\begin{aligned}\text{ColorR} &= \text{ColorS} \\ \text{AlphaR} &= \text{AlphaS}\end{aligned}$$



SRC_IN

With this rule, the fraction of the source in the result is the alpha for the destination, AlphaD , and the fraction of the destination in the result is zero. Thus only the source pixels that fall within the area destination image are rendered. All other pixels rendered from the source will have zero color components. As you can see, the outline of the destination image acts like a pastry cutter on the source image. The operations for the rule are:

$$\begin{aligned}\text{ColorR} &= \text{ColorS} * \text{AlphaD} \\ \text{AlphaR} &= \text{AlphaS} * \text{AlphaD}\end{aligned}$$



SRC_OUT

With this rule, only the source pixels outside the area of the destination will be rendered. The outline of the destination image also acts like a pastry cutter here, but the source image inside the outline is discarded, and only the source image lying outside of the destination image is kept. The area of the source that lies inside the boundary of the destination results in pixels with color components that are zero. The calculation of the components of the result is defined as:

$$\begin{aligned}\text{ColorR} &= \text{ColorS} * (1 - \text{AlphaD}) \\ \text{AlphaR} &= \text{AlphaS} * (1 - \text{AlphaD})\end{aligned}$$



DST_OVER

With this rule, the fraction of each source pixel in the result is $1 - \text{AlphaD}$, and the fraction of the corresponding destination pixel is 1. Where the source overlaps the destination, the destination is effectively rendered over the source. The operation of this rule is defined by:

$$\begin{aligned}\text{ColorR} &= \text{ColorS} * (1 - \text{AlphaD}) + \text{ColorD} \\ \text{AlphaR} &= \text{AlphaS} * (1 - \text{AlphaD}) + \text{AlphaD}\end{aligned}$$

Since in our illustration the destination has an alpha of 1.0f, the fraction of the source pixel is 0.0 so the destination completely hides the part of the source that is underneath.



DST_IN

With this rule, the fraction of the source in the result is 0 and the fraction of the destination is AlphaS. The operations are:

```
ColorR = ColorD*AlphaS
AlphaR = AlphaD*AlphaS
```

Thus the destination is rendered inside the boundary of the source, but with the alpha from the source, so it looks lighter than the rest of the destination.



DST_OUT

The source fraction in the result is zero, and the destination fraction is $1 - \text{AlphaS}$. Thus the rule operation is:

```
ColorR = ColorD*(1 - AlphaS)
AlphaR = AlphaD*(1 - AlphaS)
```



CLEAR

The fractions of the source and destination pixels involved in the operation of this rule are both zero. The effect is that pixels corresponding to the source pixels are cleared, so they will have all color components at zero.

The `AlphaComposite` class also defines static members that are `AlphaComposite` objects. Each object corresponds to one of the rules we have just discussed and they all have an alpha of 1.0f. These members are:

<code>SrcOver</code>	<code>SrcIn</code>	<code>SrcOut</code>	<code>Src</code>
<code>DstOver</code>	<code>DstIn</code>	<code>DstOut</code>	<code>Clear</code>

As the `Clear` object has an alpha of 1.0f, using this for alpha compositing will result in an opaque black result.

Since the alpha for an image determines the transparency, we can fade an image into the background by repainting the image with a shrinking value for the alpha. Let's see how that works with an example.

Try It Out – Fading an Image

We will fade the Wrox Press logo into the background until it disappears, and then fade it in again cyclically. We can write an applet to do this and while we are about it, we can try using parameters for the applet that we can set in the web page. Here's the code for the applet:

```
import java.awt.*;
import java.awt.image.*;
import javax.swing.*;

public class FaderApplet extends JApplet
{
    public void init()
    {
        // Get parameters - if any
        String fade = getParameter("fadeTime");
        if(fade != null)
            fadeTime = Integer.parseInt(fade);
        String fps = getParameter("frameRate");
        if(fps != null)
            frameRate = Integer.parseInt(fps);

        maxCount = frameRate*fadeTime;           // Count of steps to complete fade
        imagePanel = new ImagePanel(getSize());
        getContentPane().add(imagePanel);

        composite = AlphaComposite.SrcOver;
    }

    // Parameter information for anyone that needs it
    public String[][] getParameterInfo()
    {
        String[][] info = {
            {"fadeTime", "integer", "time to complete fade in seconds"},
            {"frameRate", "integer", "frames per second"}
        };
        return info;
    }

    public void start()
    {
        timer = new java.util.Timer(true);           // Timer to run clock
        count = maxCount;                           // Set repaint counter
        alphaStep = 1.0f/count;
        long frameInterval = ONE_SECOND/frameRate;
    }
}
```

```

// Use fixed-delay execution to get smooth fade
timer.schedule(new java.util.TimerTask()
{
    public void run()
    {
        imagePanel.repaint();           // Repaint the image

        // Update alpha composite for next frame
        if(count ==maxCount)
            countDelta = -1;
        else if(count == 0)
            countDelta = 1;
        count += countDelta;
        composite = AlphaComposite.getInstance(
            AlphaComposite.SRC_OVER,count*alphaStep);
    }

    int countDelta = -1; // Anonymous class member - count incr.
},
0,                               // ...starting now
frameInterval);                 // Repaint interval
}

```

```

public void stop()
{
    timer.cancel();
}

```

```

class ImagePanel extends JPanel
{
    // Panel creates its own image from an image icon
    public ImagePanel(Dimension size)
    {
        ImageIcon icon = new ImageIcon("Images/wrox_logo.gif");
        image = icon.getImage();

        // Create a scaled image to fit within the size
        image = image.getScaledInstance(4*size.width/5, 4*size.height/5,
                                           Image.SCALE_SMOOTH);

        // Wait for scaled image to load
        MediaTracker tracker = new MediaTracker(this);
        tracker.addImage(image,0);           // Image to track
        try
        {
            tracker.waitForID(0);
        }
        catch(InterruptedException e)
        {
            System.out.println(e);           // Exception...
        }
    }
}

```

```

        System.exit(1); // ...so abandon ship!
    }
}

public void paint(Graphics g)
{
    Graphics2D g2D = (Graphics2D)g;
    Dimension size = getSize(); // Get panel size
    g2D.setPaint(Color.lightGray); // Background color
    g2D.fillRect(0,0,size.width,size.height); // fill the panel
    g2D.setComposite(composite); // Set current alpha

    // Scale and draw image
    g2D.drawImage(image, // Image to be drawn
                  size.width/10,size.height/10, // Image position inset
                  null);

}

// ImagePanel data members
Image image; // The image
int imageWidth; // and its width
int imageHeight; // and height
}

// Applet data members
final int ONE_SECOND = 1000; // One second in milliseconds
int frameRate = 20; // Default fade change frequency frames per sec
int fadeTime = 3; // Default time to fade completely in seconds
int count; // Repaint cycle counter
int maxCount;

ImagePanel imagePanel; // Panel displaying the image
AlphaComposite composite; // Alpha value for the image
float alphaStep; // Alpha increment for fade step
java.util.Timer timer; // Timer to control fading
}

```

We can optionally supply parameters for the applet, so you can use the following html:

```

<applet code="FaderApplet.class" width="300" height="330">
  <param name="frameRate" value="20">
  <param name="fadeTime" value="3">
</applet>

```

You can use `appletviewer` to run the applet with an HTML file with the contents above. You should see the image fade out then back in again.

How It Works

In the `init()` method we try to read the two parameter values. If the `<PARAM>` tags are not specified, then the `String` object returned by the `getParameter()` calls will be `null`. In this case the data members `fadeTime` and `frameRate` will be left at their default values. We use the values in these members to calculate `maxCount`, the count of the number of steps needed to completely fade the image.

Since the `ImagePanel` class loads its own icon image, all we have to do in the `init()` method is create the `ImagePanel` object and add it to the content pane for the applet. We pass the size of the applet to the `ImagePanel` constructor, which will scale the image to fit. We also initialize the composite member that is the `AlphaComposite` object used in the `paint()` method for the `imagePanel` object to `SrcOver`, which implements the `SRC_OVER` rule with the alpha set to 1.

The `Timer` object that we create in the `start()` method manages the animation. After initializing count to the number of steps to a complete fade, we calculate the increment for the alpha value between steps, and the time interval in milliseconds between one step and the next. We use the timer's `schedule()` method to fade the image. In the `run()` method for the `TimerTask` object, the alpha is set to a new value – greater than the previous value if we are fading in, and less than the previous value if we are fading out – after each repaint of the panel. An `AlphaComposite` object with the rule `SRC_OVER` and the current alpha value is stored in the `composite` member of the applet class. In this way we cycle the alpha for the `AlphaComposite` object from 0.0f to 1.0f and back again.

The `ImagePanel` constructor loads the image as an `ImageIcon` object – you will recall that the constructor takes care of loading the image, and will not return until loading is complete. The constructor then extracts the `Image` object from the `ImageIcon` object. Since we want to be sure the image fits comfortably in the space available to the applet, we call the `getScaledInstance()` method for the `Image` object to create a new object that is scaled to the x and y dimensions supplied as the first two arguments. The third argument to the `getScaledInstance()` method is an integer that must be one of five constant values that are defined in the `Image` class, and that select a particular scaling algorithm:

Scale Algorithm	Description
<code>SCALE_DEFAULT</code>	The default scaling algorithm
<code>SCALE_FAST</code>	A fast scaling algorithm
<code>SCALE_SMOOTH</code>	An algorithm designed for a smooth resultant image rather than speed
<code>SCALE_REPLICATE</code>	Use the algorithm defined by the <code>ReplicateScaleFilter</code> class. This algorithm duplicates pixels to scale up, or deletes pixels to scale down.
<code>SCALE_AREA_AVERAGING</code>	Use the algorithm defined by the <code>AreaAveragingScaleFilter</code> class.

Scaling an image is achieved by applying an algorithm that calculates the pixel values for the new image from those of the old. Some algorithms such as that defined by the `ReplicateScaleFilter` are very simple, and hence very fast to execute. Others such as the `SCALE_SMOOTH` algorithm are more complex, and hence slower, but produce a much better result.

The `getScaledInstance()` method returns immediately, even if the scaled image has not yet been constructed, so we use a tracker to wait for the image to complete.

The image is drawn in the `paint()` method for the `imagePanel` object. The `paint()` method starts by filling the panel in light gray to provide the background to the image. The current `composite` object is then set in the graphics context and the alpha for this determines the transparency of the image when we draw it. The image is then drawn using the `drawImage()` method. The second and third arguments are the coordinates of the top-left corner of the image. The last argument can be a reference to an `ImageObserver`, but we just supply `null` as we have ensured that the image is completely loaded in the `ImagePanel` constructor.

Synthesizing Images

You don't have to read all your images in from files – you can create your own. The most flexible way of doing this involves using a `BufferedImage` object. This is a subclass of the `Image` class that has the image data stored in a buffer that you can access. It also supports a variety of ways of storing the pixel data: with or without an alpha channel, different types of color model and with various precisions for color components. The `ColorModel` class provides a flexible way to define various kinds of color models for use with a `BufferedImage` object but to understand the basics of how this works we will only use one color model, the default where color components are RGB values, and one buffer type – storing 8 bit RGB color values plus an alpha channel. This type is specified in the `BufferedImage` class by the constant `TYPE_INT_ARGB`, which implies an `int` value is used for each pixel. The value for each pixel stores an alpha component plus the RGB color components as 8 bit bytes. We can create a `BufferedImage` object of this type with a given width and height with statements such as:

```
int width = 200;
int height = 300;
BufferedImage image = new BufferedImage(width, height,
                                         BufferedImage.TYPE_INT_ARGB);
```

This creates a `BufferedImage` object that represents an image 200 pixels wide and 300 pixels high. To draw on the image, we need a graphics context, and the `createGraphics()` method for the `BufferedImage` object returns a `Graphics2D` object that relates to the image:

```
Graphics2D g2D = image.createGraphics();
```

Operations using the `g2D` object modify pixels in the `BufferedImage` object, `image`. With this object available you now have the full capability to draw on the `BufferedImage` object – you can draw shapes, images, `GeneralPath` objects or whatever, and you can set the alpha compositing object for the graphics context as we discussed in the previous section. And you also have all the affine transform capability that comes with a `Graphics2D` object.

You can retrieve an individual pixel from a `BufferedImage` object by calling its `getRGB()` method and supplying the `x,y` coordinates of the pixel as arguments of type `int`. The pixel is returned as type `int` in the `TYPE_INT_ARGB` format, which consists of four 8-bit values for the alpha and RGB color components packed into the 32-bit word. There is also an overloaded version of `getRGB()` that returns an array of pixels from a portion of the image data. You can set an individual pixel value by calling the `setRGB()` method. The first two arguments are the coordinates of the pixel, and the third argument is the value that is to be set, of type `int`. There is also a version of this method that will set the values of an array of pixels. We won't be going into pixel operations further, but we will venture drawing on a `BufferedImage` object with a working example.

We will put together an applet that uses a `BufferedImage` object. The applet will create an animation of the buffered image object that is created against a background of the Wrox logo. This will also demonstrate how you can make part of an image transparent. The applet will be broadly similar to applets we have produced earlier in this chapter; the basic contents of the source file will look like this:

```

import java.awt.*;
import java.awt.image.*;
import java.awt.geom.*;
import javax.swing.*;

public class ImageDrawDemo extends JApplet
{
    // The init() method to initialize everything...

    // The start() method to start the animation...
    // The stop() method to stop the animation...
    // The ImagePanel class defining the panel displaying the animation...

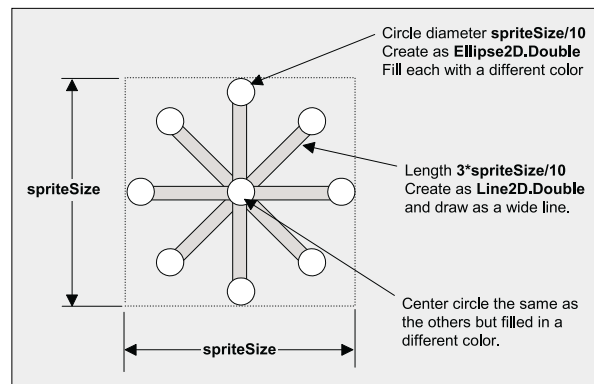
    // Data members for the applet...
}

```

Creating an Image

A **sprite** is a small graphical image that you can draw over a static image to create an animation. To create the animation effect, you just draw the sprite in different positions and orientations over time, and of course transformations of the coordinate system can be a great help in making this easier. Games often use sprites – they can make the animation take much less processor time because you only need to draw the sprite against a static background. Our interest in using **BufferedImage** objects means we won't get into the best techniques for minimizing processor time. We will instead concentrate on understanding how we can create and use images internally in a program.

Our **BufferedImage** object is going to look as shown below.



The image is a square with sides of length **spriteSize**. Dimensions of other parts of the image are relative to this. There are really only two geometric entities here, a line and a circle, each repeated in different positions and orientations, so if we create a **Line2D.Double** object for the line, and an **Ellipse2D.Double** object for the circle, we should be able to draw the whole thing by moving the user coordinate system around and drawing one or other of these two objects.

A true object-oriented approach would define a class representing a sprite, possibly as a subclass of **BufferedImage**, but since we are exploring the mechanics of using a **BufferedImage** object, it will suit our purpose better to develop a method, **createSprite()**, that draws the sprite on a **BufferedImage** object. The method will just be a member of our applet class so we will add data members to the applet to store any data required. You can plug the data members we will be using into the applet class outline now:


```

double totalAngle;           // Current angular position of sprite
double spriteAngle;         // Rotation angle of sprite about its center
ImagePanel imagePanel;      // Panel to display animation

BufferedImage sprite;       // Stores reference to the sprite
int spriteSize = 100;       // Diameter of the sprite
Ellipse2D.Double circle;    // A circle - part of the sprite
Line2D.Double line;         // A line - part of the sprite

// Colors used in sprite
Color[] colors = {Color.red , Color.yellow, Color.green , Color.blue,
Color.cyan, Color.pink , Color.magenta, Color.orange};

java.util.Timer timer;      // Timer for the animation
long interval = 50;         // Time interval msec between repaints

```

The general use of these members should be clear from the comments. We will see how they are used as we develop the code.

The first thing the `createSprite()` method needs to do is to create the `BufferedImage` object, `sprite`, and we will need a `Graphics2D` object to use to draw on the `sprite` image. The code to do this is as follows:

```

BufferedImage createSprite(int spriteSize)
{
    // Create image with RGB and alpha channel
    BufferedImage sprite = new BufferedImage(spriteSize, spriteSize,
                                           BufferedImage.TYPE_INT_ARGB);

    Graphics2D g2D = sprite.createGraphics();    // Context for buffered image
    // plus the rest of the method...
}

```

The `sprite` object has a width and height of `spriteSize`, and the image is of the type `TYPE_INT_ARGB`, so the alpha and color components for each pixel will be stored as a single `int` value, and the color will be stored as 8 bit red, green and blue components. This means that our `sprite` image will occupy 40,000 bytes, which is a small indication of how browsing a web page can gobble up memory. This doesn't affect the download time for the page – this memory is allocated in the local machine when the applet is executed. Apart from the contents of the HTML file that is the page, the download time is affected by the size of the `.class` file for the applet, plus any image or other files that it downloads when it executes.

Creating a Transparent Background

The alpha channel is important in our `sprite` image because we want the background to be completely transparent – only the `sprite` object itself should be seen when we draw it, not the whole 100x100 rectangle of the image. We can achieve this quite easily by making the whole `sprite` image area transparent to start with – that is with an alpha of 0.0f – and then drawing what we want on top of this as opaque with an alpha of 1.0f. Here's the code to make the entire image transparent:

```
// Clear image with transparent alpha by drawing a rectangle
g2D.setComposite(AlphaComposite.getInstance(AlphaComposite.CLEAR, 0.0f));
Rectangle2D.Double rect = new Rectangle2D.Double(0,0, spriteSize, spriteSize);
g2D.fill(rect);
```

We first set the alpha composite using an `AlphaComposite` object with the `CLEAR` rule – this will set the color components to zero, and with an alpha of `0.0f`, which will make it transparent. We then fill a rectangle that covers the whole area of the image. We don't need to set a color since with the `CLEAR` rule the fraction of the foreground and background for each pixel is zero, so neither participates in the resulting pixel. We still have to fill the rectangle though, since this determines the image pixels that are operated on.

At this point we can take a short detour into how you can affect the quality of your images.

Rendering Hints

With many aspects of rendering operations, there is a choice between quality and speed. Rendering operations are like most things – quality comes at a cost and the cost here is processing time. There are defaults set for all of the rendering operations where a choice exists and the default is platform specific, but you can make your own choices by calling the `setRenderingHint()` method for the `Graphics2D` object that is doing the rendering. These are only hints though, and if your computer does not support the option for a rendering operation corresponding to the hint that you specify, the hint will have no effect.

We can ensure that we get the best possible result from our alpha compositing operations by adding the following call to the `createSprite()` method:

```
BufferedImage createSprite(int spriteSize)
{
    // Create image with RGB and alpha channel
    BufferedImage sprite = new BufferedImage(spriteSize, spriteSize,
                                           BufferedImage.TYPE_INT_ARGB);

    Graphics2D g2D = sprite.createGraphics();           // Context for buffered image

    // Set best alpha interpolation quality
    g2D.setRenderingHint(RenderingHints.KEY_ALPHA_INTERPOLATION,
                        RenderingHints.VALUE_ALPHA_INTERPOLATION_QUALITY);

    // Clear image with transparent alpha by drawing a rectangle
    g2D.setComposite(AlphaComposite.getInstance(AlphaComposite.CLEAR, 0.0f));
    Rectangle2D.Double rect = new Rectangle2D.Double(0,0, spriteSize, spriteSize);
    g2D.fill(rect);

    // plus the rest of the method...
}
```

The `RenderingHints` class defines rendering hints of various kinds, and these are stored in a `Graphics2D` object in a map collection, so the arguments to the `setRenderingHint()` method are a key and a value corresponding to the key. The key for the alpha compositing hint is the first argument in our code, and the second argument is the value of the hint. Other possible values for this hint are `VALUE_ALPHA_INTERPOLATION_DEFAULT` for the platform default, and `VALUE_ALPHA_INTERPOLATION_SPEED` for speed rather than quality.

You can also supply a hint for the following keys:

Key	Description
KEY_ANTIALIASING	When you render angled lines, you will get a step-wise arrangement of pixels a lot of the time, making the line look less than smooth, often referred to as 'jaggies'. Antialiasing is a technique to set the brightness of pixels for an angled line to make the line appear smoother. Thus this hint determines whether time is spent reducing 'jaggies' when rendering angled lines. Possible values are VALUE_ANTIALIAS_ON, _OFF or _DEFAULT.
KEY_COLOR_RENDERING	Affects how color rendering is done. Possible values are VALUE_COLOR_RENDER_SPEED, _QUALITY or _DEFAULT.
KEY_DITHERING	Dithering is a process whereby a broader range of colors can be synthesized with a limited set of colors by coloring adjacent pixels to produce the illusion of a color that is not in the set. Possible values are VALUE_DITHER_ENABLE, _DISABLE or _DEFAULT.
KEY_FRACTIONALMETRICS	Affects the quality of displaying text. Possible values are VALUE_FRACTIONALMETRICS_ON, _OF or _DEFAULT.
KEY_INTERPOLATION	When you transform a source image, the transformed pixels will rarely correspond exactly with the pixel positions in the destination. In this case the color values for each transformed pixel have to be determined from the surrounding pixels. Interpolation is the process by which this is done and there are various techniques that can be used. Possible values from most to least expensive in time are VALUE_INTERPOLATION_BICUBIC, _BILINEAR and _NEAREST_NEIGHBOR.
KEY_RENDERING	This determines the rendering technique trading speed and quality. Possible values are VALUE_RENDERING_SPEED, _QUALITY and _DEFAULT.
KEY_TEXT_ANTIALIASING	This determines whether antialiasing is done when rendering text. Possible values are VALUE_TEXT_ANTIALIASING_ON, _OFF and _DEFAULT.

That's enough of a detour. Let's get back to drawing our sprite.

Drawing on an Image

We have the transparent background so we need to fill in the image. The first step is to change the alpha compositing operation to the `SRC_OVER` rule with the alpha set so the result is opaque. Adding the following statement at the end of those we have in the `createSprite()` method will do this:

```
g2D.setComposite(AlphaComposite.SrcOver);
```

This uses the `SrcOver` member of the `AlphaComposite` class, which does exactly what we want.

We can create the two geometric entities, the circle and the line with the statements:

```
line = new Line2D.Double(spriteSize/20.0,0,0.3*spriteSize,0);
circle = new Ellipse2D.Double(0,0, spriteSize/10.0, spriteSize/10.0);
```

The line is a horizontal line from an x position of `spriteSize/20` (half the diameter of the circle) so this is from the right edge of the center circle of the sprite, to `0.3*spriteSize`, which is the length we specified in the design. The circle has its top-left corner at the origin and a diameter of `spriteSize/10`, 10% of the width of the image.

If you take another look at the diagram, you will see that the sprite has eight arms, all identical apart from their orientation, with each arm rotated $\pi/4$ radians relative to its predecessor. If we had a method to draw one arm in a given orientation, horizontal say, we could draw all eight by rotating the coordinate system by $\pi/4$ radians and calling the method to draw the arm eight times. Good idea!

Drawing an Arm

The `drawArm()` method will need the `Graphics2D` object for the image as an argument, plus an index value to select the color of the circle at the end of the arm. In the diagram we noted that we would fill the circles with different colors and we will do this by selecting an element from the `colors` array. The process is very simple. We will draw the arm aligned horizontally along the x axis. We will then translate the coordinate system so the origin is where the top-left corner of the colored, outer circle should be.

Here's the code for the `drawArm()` method:

```
void drawArm(Graphics2D g2D, int i)
{
    AffineTransform at = g2D.getTransform();           // Save current transform
    g2D.setPaint(Color.darkGray);                       // Save current stroke
    Stroke stroke = g2D.getStroke();                     // Set stroke as wide line
    g2D.setStroke(new BasicStroke(3.0f));               // Draw the line
    g2D.draw(line);                                     // Restore old stroke
    g2D.setStroke(stroke);

    g2D.translate(line.getX2(), -spriteSize/20);        // Translate to circle position
    g2D.setPaint(colors[i%colors.length]);              // Set the fill color
    g2D.fill(circle);
    g2D.setTransform(at);                              // Restore original transform()
}
```

We first save the current transform so we can restore it later, after we have finished messing about with it in the graphics context object, `g2D`. We will draw all the arms in dark gray, so we set that color in the graphics context first. We then have a couple of statements that save the current line type by calling `getStroke()` for `g2D`, and set a new line type with a greater width by calling the `setStroke()` method with `stroke` as the argument, which as you see is a line of width 3. The next step is to draw the line by calling the `draw()` method with `line` as the `Shape` argument. This will be drawn in the current color, `Color.darkGray`.

Before drawing the circle at the end of the arm, we must move the origin to where the top-left corner point of the circle will be. This is at the x coordinate of the end point of the line that we obtain by calling the `getX2()` method for the line object, and a y coordinate that is up by the radius of the circle, `spriteSize/20`. With the origin in the right place, we then just need to select the color from the `colors` array, and fill the `circle` object.

Completing the Sprite Image

We can use the `drawArm()` method to draw most of the sprite. By default the method draws the line with the circle at the end horizontally – along the x axis from the origin. We want to apply this eight times at angles that are $\pi/4$ radians apart rotating about the center of the image. If we translate the user coordinate system so that the origin is at the center of the image, we can draw the arms in a loop like this:

```
BufferedImage createSprite(int spriteSize)
{
    // Create image with RGB and alpha channel
    BufferedImage sprite = new BufferedImage(spriteSize, spriteSize,
                                           BufferedImage.TYPE_INT_ARGB);

    Graphics2D g2D = sprite.createGraphics();          // Context for buffered image

    // Set best alpha interpolation quality
    g2D.setRenderingHint(RenderingHints.KEY_ALPHA_INTERPOLATION,
                        RenderingHints.VALUE_ALPHA_INTERPOLATION_QUALITY);

    // Clear image with transparent alpha by drawing a rectangle
    g2D.setComposite(AlphaComposite.getInstance(AlphaComposite.CLEAR, 0.0f));
    Rectangle2D.Double rect = new Rectangle2D.Double(0, 0, spriteSize,
                                                    spriteSize);
    g2D.fill(rect);

    g2D.setComposite(AlphaComposite.SrcOver);
    line = new Line2D.Double(spriteSize/20.0, 0, 0.3*spriteSize, 0);
    circle = new Ellipse2D.Double(0, 0, spriteSize/10.0, spriteSize/10.0);

    // Since sprite is symmetric, move origin to the center
    g2D.translate(spriteSize/2, spriteSize/2);

    int armCount = 8;                                // Number of arms in the sprite
    for(int i = 0; i < armCount; i++)
    {
        g2D.rotate(2*Math.PI/armCount);               // Rotate by pi/4
        drawArm(g2D, i);                               // Draw the arm
    }

    // plus the rest of the code...
}
```

With the origin at the center of the image, we rotate the coordinate system about the origin by $\pi/4$ radians and draw the arm on each iteration of the loop. The `drawArm()` method also transforms the coordinates, but it always resets the transform back to what it was when the method was called. The rotations in the loop are cumulative, so we draw the arm at successive orientations around the origin.

The last thing we need to do is to fill the circle at the center with the color dark gray. Adding the following statements to the `createSprite()` method will complete it:

```
g2D.setPaint(Color.lightGray);           // Set the fill color
g2D.translate(-spriteSize/20,-spriteSize/20); // Move origin to circle top left
g2D.fill(circle);                         // Fill the circle
g2D.dispose();                           // Dispose of the context
return sprite;                           // Return the finished image
```

Now we can create a `BufferedImage` object containing the sprite by calling `createSprite()` in the `init()` method for the applet:

```
public void init()
{
    sprite = createSprite(spriteSize);

    imagePanel = new ImagePanel();
    getContentPane().add(imagePanel);
}
```

The `imagePanel` object's `paint()` method will draw the sprite, but before we get to that, let's look at how the animation thread is going to work.

Animating the Sprite

The `start()` and `stop()` methods for the applet that control the animation will use a `Timer` object, exactly as you have seen before:

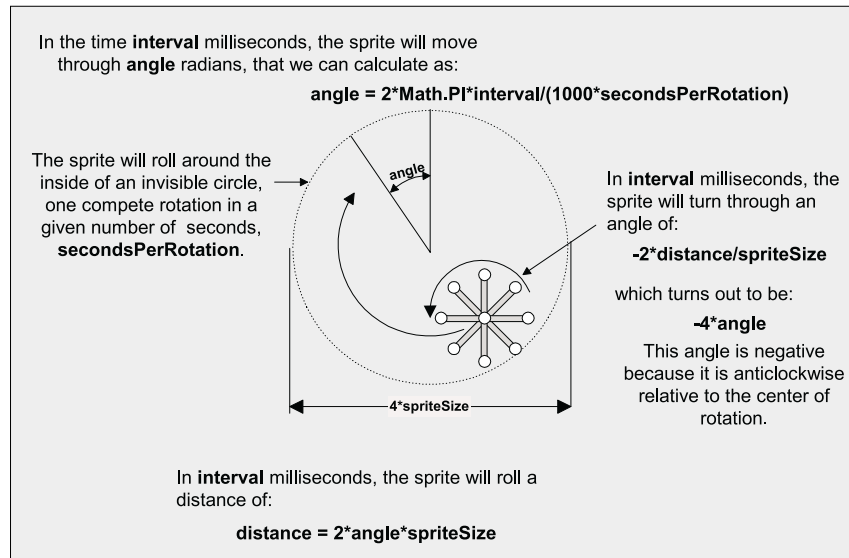
```
public void start()
{
    timer = new java.util.Timer(true);

    // Do necessary initialization...

    // Use fixed-delay execution to get smooth animation
    timer.schedule(new java.util.TimerTask()
    {
        public void run()
        {
            // Painting for animation + necessary updating
        }
    },
    0, // ...starting now
    interval); // Repaint interval
}

public void stop()
{
    timer.cancel(); // Stop the timer
}
```

All the hard work is still to do, in the initialization in the `start()` method and in the `run()` method for the `TimerTask` object, so let's consider first what our animation is going to be. We can draw our sprite anywhere at any orientation just by transforming the coordinate system, so let's choose an animation to show this off. A simple illustration would be to have the sprite roll round the inside of an invisible circle that is four times the diameter of the sprite, as illustrated below.



This will give us plenty of opportunity to use transforms. It may also give those who didn't pay attention to trigonometry in high school some cause for regret. We will set the speed of rotation around the circle by specifying the number of seconds, `secondsPerRotation`, it takes to complete a revolution. We don't want to have this happening too fast – 15 seconds is good. We can add a member to the applet class to define this:

```
double secondsPerRotation = 15; // Time for one complete revolution
```

We can position the sprite at the end of each time interval for updating the image if we know two angles – the rotation of the position of the sprite about the center of the circle in the given time interval, and the rotation of the sprite about its own center due to rolling. We can then position the sprite at any time by going through a series of transforms. First we will apply a rotation about the center of the circle to establish the angular position of the sprite about the circle. From that position we can apply a translation of the axes to position the center of the sprite. We will then apply a rotation about the new origin to get the sprite to its correct orientation. Finally we will apply another translation of the axes to the position where the top left of the sprite image should be.

Given that we update the image every `interval` milliseconds, we can work out the angle that the sprite moves through, relative to the center of the circle in this time period. We'll call this `angle`. The time, `interval`, divided by the time for a complete rotation, `secondsPerRotation`, will be the fraction of 2π that the sprite will rotate through, since 2π is a complete revolution. These times have to be in the same units of course, either both seconds or both milliseconds. This angle will be positive because the rotation is clockwise with respect to the center of rotation – the center of the circle in this case.

Since we know the circumference of the outer circle is 2π times the radius, which is $2 * \text{spriteSize}$, we can easily get the distance traveled around that circle in `interval` milliseconds with the expression shown in the diagram. As the sprite rolls this distance, it is also the distance moved around the circumference of the sprite – which in terms of the sprite's rotation about its center is the angle turned through times the radius, `spriteSize/2`. Thus the angle through which the sprite itself rotates, which we'll call `spriteAngleIncrement`, is given by the expression in the diagram. This angle has to be negative because it is counterclockwise. We can now add two further members to the applet class that we will need for positioning the sprite, `angle` and `spriteAngleIncrement`:

```
double angle = 2.*interval*Math.PI/(1000.0*secondsPerRotation);
double spriteAngleIncrement = -4*angle;
```

We can now use these to complete the `start()` method:

```
public void start()
{
    timer = new java.util.Timer(true);

    // Do necessary initialization...
    totalAngle = 0;                // Position around the circle
    spriteAngle = 0;              // Sprite orientation

    // Use fixed-delay execution to get smooth animation
    timer.schedule(new java.util.TimerTask()
    {
        public void run()
        {
            // Painting for animation + necessary updating
            imagePanel.repaint();    // Repaint the image
            totalAngle += angle;    // Increment the total angle
            spriteAngle += spriteAngleIncrement; // and the sprite angle
        }
    },
    0,                               // ...starting now
    interval);                      // Repaint interval
}
```

The calculation of the angles is precisely as we have discussed. The expressions from the diagram are plugged into the code here. You may notice that we continue to increment the values of `totalAngle` and `spriteAngle` indefinitely here. However, since these are both of type `double`, it will be a while before we exceed the number of digits available. All we have to do now is to implement how the sprite is painted.

Painting the Sprite

We do this in the `paint()` method for the `ImagePanel` class. To make it a bit more interesting we will use the Wrox Press logo as the background to our sprite animation. This will show up the fact that the sprite background is transparent, and our alpha compositing is really working. To shorten the code we will get the logo using an `ImageIcon` object, so the constructor for the inner class will be implemented as:

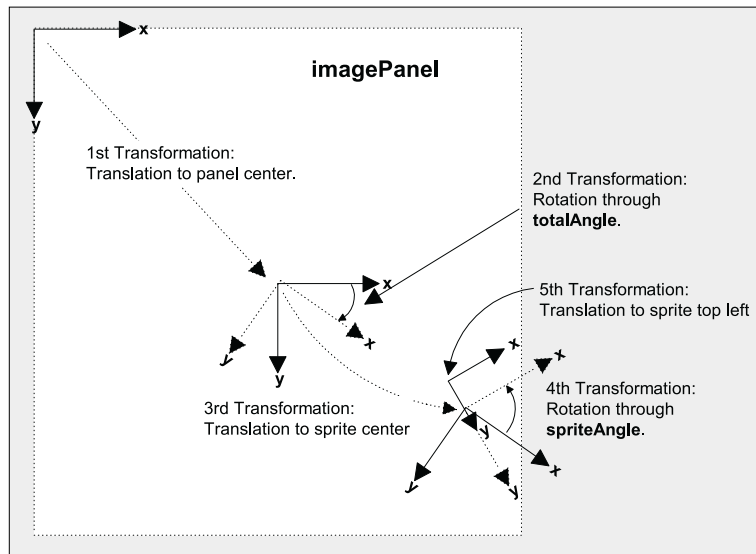

```

class ImagePanel extends JPanel
{
    public ImagePanel()
    {
        ImageIcon icon = new ImageIcon("Images\\wrox_logo.gif");
        image = icon.getImage();
    }

    Image image;                                // The logo image
}

```

The Wrox logo will be drawn in the center of the space available to the `ImagePanel` object, and the invisible circle will have its center here too, so the first transformation we can apply in the `paint()` method is to move the origin of the user coordinate system to the center of the panel.



With the coordinate system at the center of the panel, we can then draw the logo image by specifying its top left corner x, y coordinates as minus half its width and minus half its height.

Positioning the sprite using the `totalAngle` and `spriteAngle` angles will involve four further transformations as shown in the diagram, but we will make use of the version of `rotate()` that does a translation followed by a rotation, and then a translation back again, just to try it out. Here's how the `paint()` method looks:

```

public void paint(Graphics g)
{
    Dimension size = getSize();                                // Get panel dimensions
    Graphics2D g2D = (Graphics2D)g;

    // Now fill the panel background
    g2D.setPaint(Color.gray);
    Rectangle2D.Double rect = new Rectangle2D.Double(0, 0, size.width, size.height);
    g2D.fill(rect);
}

```

```

g2D.translate(size.width/2, size.height/2); // Move origin to panel center

g2D.drawImage(image,                                // Draw the logo
               -image.getWidth(ImageDrawDemo.this)/2, // Top left x
               -image.getWidth(ImageDrawDemo.this)/2, // Top left y
               ImageDrawDemo.this);

g2D.rotate(totalAngle); // Rotate to the angle for the sprite position

// Translate and rotate to sprite position - then translate back
g2D.rotate(spriteAngle, 3*spriteSize/2, 0);

g2D.translate(spriteSize, -spriteSize/2); // Translate to sprite top left

g2D.drawImage(sprite ,null, 0, 0);          // Draw the sprite
}

```

The sprite and the applet should be ready to roll, barring typos, so try it out.

Try It Out – Rolling the Sprite

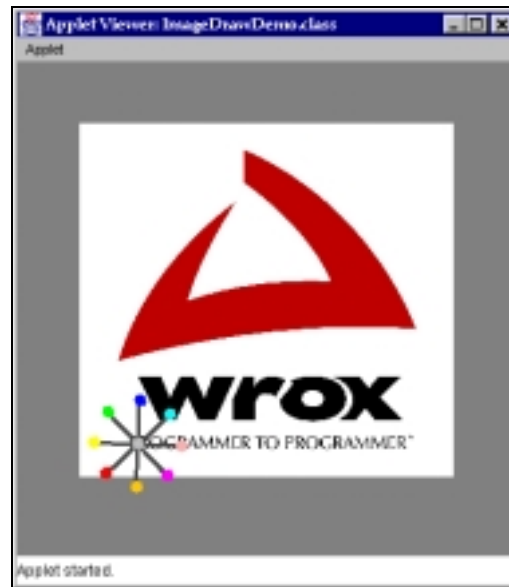
All the code should be there. You need to set the size of the applet so it is sufficient to accommodate everything, so it needs to be at least four times the sprite diameter. I used the following HTML for the applet:

```

<applet code="ImageDrawDemo.class" width=400 height=400>
</applet>

```

Of course if you want to do a bit more work, you could always either adapt the applet size to fit the size of the sprite – or vice versa. Here is a screenshot of the applet in operation.



You can see here that the background to the sprite is transparent – the text and the background to the logo under the sprite show quite clearly.

Summary

This chapter has been a basic introduction to handling images and producing animations. Java provides much greater capabilities in this area than we have looked at, but with what you now know, you should not find it too difficult to explore those additional facilities yourself.

The important points that we covered in this chapter were:

- ❑ Implementing an applet generally involves implementing four methods that are called by the browser or the context in which the applet is running – the `init()` method that initializes the applet, the `start()` method that is called to start the applet, the `stop()` method that is called to `stop()` whatever the applet is doing and the `destroy()` method that is called when the applet ends. You should also implement the `getAppletInfo()` and `getParameterInfo()` methods for your applet.
- ❑ Objects of the `URL` class represent uniform resource locators on the Internet. You can define a URL in an applet either as an absolute URL, or relative to the original URL.
- ❑ An `Image` object represents an image, and you can create images from files in GIF, PNG or JPEG format.
- ❑ The control of the timing interval for an animation generally runs in a separate thread, and in an applet the thread is usually started in the `start()` method and stopped in the `stop()` method.
- ❑ You can use an object of the `java.util.Timer` class to control an animation that you expedite through a `TimerTask` object. You can apply `Timer` and `TimerTask` object to scheduling any operation that recurs at a fixed time interval.
- ❑ The alpha component for a pixel specifies its transparency, where an alpha value of zero is transparent and a value of one is opaque.
- ❑ Alpha compositing determines how the components of the pixels for a source image are combined with the components of the destination image over which it is rendered.
- ❑ Rendering hints provide a way for you to trade between speed and quality in rendering operations.
- ❑ The type of line produced by drawing operations in a graphics context is determined by the current stroke set in the context. A line type is specified by a `BasicStroke` object.
- ❑ The glass pane for a `JApplet` or `JFrame` object overlays the content pane, so you can draw animated elements of an image in the glass pane and reserve the content pane for the fixed elements. You must ensure the background to the glass pane is transparent to enable the content pane to be seen.
- ❑ You can synthesize images by creating a `BufferedImage` object and drawing on it using the `Graphics2D` object that represents the image.

Exercises

- 1.** Create an applet that will create an image from a GIF file and display four copies of it in two rows of two.
- 2.** Modify the result of the previous exercise so that the applet obtains the URL for the image file from a `PARAM`.
- 3.** Modify the last example to use the glass pane to display the sprite, and arrange that the logo is only visible inside the circle that the sprite rolls around. (Hint: draw an image on top of the logo with a transparent circle.)
- 4.** Adapt the last example of the chapter to simultaneously display a fading logo and the rotating sprite in two threads.
- 5.** The animation in the last example in the chapter could be generated by a fixed sequence of images that repeats cyclically. Generate the animation by creating all the images you need beforehand and making the `paint()` method in the `ImagePanel` class display the appropriate image.
- 6.** Modify `Sketcher` to support lines of various types and widths – at least dashed and dotted lines in addition to solid lines.
- 7.** Create an applet to create a sequence of images from an arbitrary number of GIF files, and replay them as an animation at an interval specified by a `<PARAM>` tag. (Hint: you can give your GIF files names such as `image0.gif`, `image1.gif`, `image3.gif` and so on. The `getImage()` method that your applet class inherits returns `null` if the URL cannot be found.)

