In this chapter:
- *Image Processing*
- *Implementing an ImageObserver*
- *Producing Image Data*
- *Image Producers and Consumers*
- *Filtering Image Data*

1

Working with Images

This section contains excerpts from chapter 17 of the 2nd edition of the book Exploring Java, O'Reilly & Associates. It describes the original Java ImageProducer and ImageConsumer APIs for generating image data and video. Although these APIs have been largely obsoleted by the Java2D and BufferedImage APIs, we provide this information for reference in understanding older applications using those techniques.

Image Processing

Up to this point, we've confined ourselves to working with the high-level drawing commands of the `Graphics` class and using images in a hands-off mode. In this section, we'll clear up some of the mystery surrounding images and see how they are produced and used. The classes in the `java.awt.image` package handle image processing; Figure 1-1 shows the classes in this package.

First, we'll return to our discussion about image observers and see how we can get more control over image data as it's processed asynchronously by AWT components. Then we'll open the hood and have a look at image production. If you're interested in creating sophisticated graphics, such as rendered images or video streams, this will teach you about the foundations of image construction in Java.*

Objects that work with image data fall into one of three categories: image-data producers, image-data consumers, and image-status observers. Image producers implement the `ImageProducer` interface. They create pixel data and distribute it to one or more consumers. Image consumers implement a corresponding

* You will also want to pay attention to the forthcoming Java Media API. Java Media will support plug-and-play streaming media.

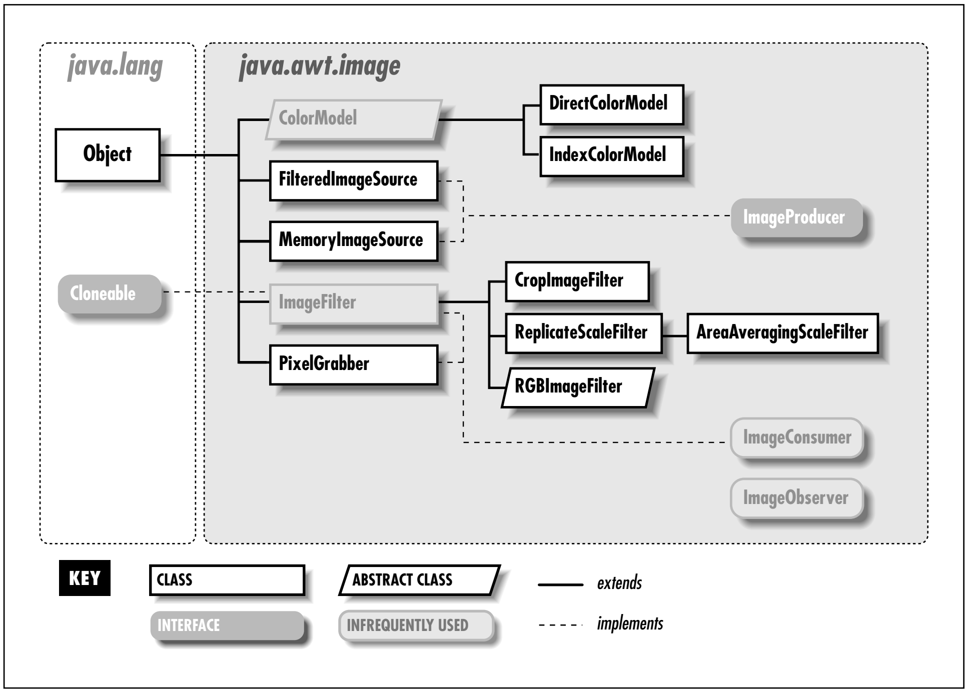

Figure 1–1: The java.awt.image package

`ImageConsumer` interface. They eat the pixel data and do something useful with it, such as display it on screen or analyze its contents. Image observers, as I mentioned earlier, implement the `ImageObserver` interface. They are effectively nosy neighbors of image consumers that watch as the image data arrives.

Image producers generate the information that defines each pixel of an image. A pixel has both a color and a transparency; the transparency specifies how pixels underneath the image show through. Image producers maintain a list of registered consumers for the image and send them this pixel data in one or more passes, as the pixels are generated. Image producers give the consumers other kinds of information as well, such as the image's dimensions. The producer also notifies the consumer when it has reached a boundary of the image. For a static image, such as GIF or JPEG data, the producer signals when the entire image is complete, and production is finished. For a video source or animation, the image producer could generate a continuous stream of pixel data and mark the end of each frame.

An image producer delivers pixel data and other image-attribute information by invoking methods in its consumers, as shown in Figure 1-2. This diagram illustrates an image producer sending pixel data to three consumers by invoking their `setPixels()` methods.

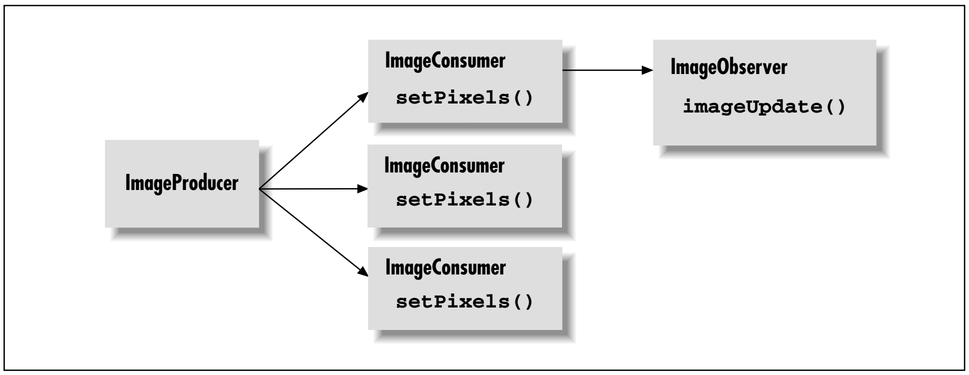

Figure 1–2: Image observers, producers, and consumers

Each consumer represents a view of the image. A given consumer might prepare the image for display on a particular medium, or it might simply serve as a filter and pass the image data to another consumer down the line.

Figure 1-2 also shows an image observer, watching the status of one of the consumers. The observer is notified as new portions of the image and new attributes are ready. Its job is to track this information and let another part of the application know its status. As I discussed earlier, the image observer is essentially a callback that is notified asynchronously as the image is built. The default `Component` class image observer that we used in our previous examples called `repaint()` for us each time a new section of the image was available, so that the screen was updated more or less continuously as the data arrived. A different kind of image observer might wait for the entire image before telling the application to display it; yet another observer might update a loading meter showing how far the image loading had progressed.

Implementing an ImageObserver

To be an image observer, you have to implement the single method, `imageUpdate()`, defined by the `java.awt.image.ImageObserver` interface:

```
public boolean imageUpdate(Image image, int flags, int x, int y,
                           int width, int height)
```

`imageUpdate()` is called by the consumer, as needed, to pass the observer information about the construction of its view of the image. Essentially, any time the image changes, the consumer tells the observer so that the observer can perform any necessary actions, like repainting. `image` holds a reference to the `Image` object the consumer is processing. `flags` is an integer whose bits specify what information about the image is now available. The values of the flags are defined as `static` identifiers in the `ImageObserver` interface, as shown in Table 1-1.

Table 1–1: ImageObserver Information Flags

Flag	Description
HEIGHT	The height of the image is ready.
WIDTH	The width of the image is ready.
FRAMEBITS	A frame is complete.
SOMEBITS	An arbitrary number of pixels have arrived.
ALLBITS	The image is complete.
ABORT	The image loading has been aborted.
ERROR	An error occurred during image processing; attempts to display the image will fail.

The flags determine which of the other parameters, `x`, `y`, `width`, and `height`, hold valid data and what that data means. To test whether a particular flag in the `flags` integer is set, we have to resort to some binary shenanigans. The following class, `MyObserver`, implements the `ImageObserver` interface and prints its information as it's called:

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

class MyObserver implements ImageObserver {

    public boolean imageUpdate( Image image, int flags, int x, int y,
                                int width, int height) {

        if ( (flags & HEIGHT) !=0 )
            System.out.println("Image height = " + height );

        if ( (flags & WIDTH ) !=0 )
            System.out.println("Image width = " + width );

        if ( (flags & FRAMEBITS) != 0 )
            System.out.println("Another frame finished.");

        if ( (flags & SOMEBITS) != 0 )
            System.out.println("Image section :"
                        + new Rectangle( x, y, width, height ) );

        if ( (flags & ALLBITS) != 0 ) {
            System.out.println("Image finished!");
            return false;
        }
```

```
        if ( (flags & ABORT) != 0 )  {
            System.out.println("Image load aborted...");
            return false;
        }


return true;
    }
}
```

The `imageUpdate()` method of `MyObserver` is called by the consumer periodically, and prints simple status messages about the construction of the image. Notice that `width` and `height` play a dual role. If `SOMEBITS` is set, they represent the size of the chunk of the image that has just been delivered. If `HEIGHT` or `WIDTH` is set, however, they represent the overall image dimensions. Just for amusement, we have used the `java.awt.Rectangle` class to help us print the bounds of a rectangular region.

`imageUpdate()` returns a `boolean` value indicating whether or not it's interested in future updates. If the image is finished or aborted, `imageUpdate()` returns `false` to indicate it isn't interested in further updates. Otherwise, it returns `true`.

The following example uses `MyObserver` to print information about an image as AWT loads it:

```
import java.awt.*;

public class ObserveImage extends java.applet.Applet {
    Image img;
    public void init() {
        img = getImage( getClass().getResource(getParameter("img")) );
        MyObserver mo = new MyObserver();
        img.getWidth( mo );
        img.getHeight( mo );
        prepareImage( img, mo );
    }
}
```

After requesting the `Image` object with `getImage()`, we perform three operations on it to kick-start the loading process. `getWidth()` and `getHeight()` ask for the image's width and height. If the image hasn't been loaded yet, or its size can't be determined until loading is finished, our observer will be called when the data is ready. `prepareImage()` asks that the image be readied for display on the component. It's a general mechanism for getting AWT started loading, converting, and possibly scaling the image. If the image hasn't been otherwise prepared or displayed, this happens asynchronously, and our image observer will be notified as the data is constructed.

You may be wondering where the image consumer is, since we never see a call to `imageUpdate()`. That's a good question, but for now I'd like you to take it on faith that the consumer exists. As you'll see later, image consumers are rather mysterious objects that tend to hide beneath the surface of image-processing applications. In this case, the consumer is hiding deep inside the implementation of `Applet`.

You should be able to see how we could implement all sorts of sophisticated image loading and tracking schemes. The two most obvious strategies, however, are to draw an image progressively, as it's constructed, or to wait until it's complete and draw it in its entirety. We have already seen that the `Component` class implements the first scheme. Another class, `java.awt.MediaTracker`, is a general utility that tracks the loading of a number of images or other media types for us. We'll look at it next.

Producing Image Data

What if we want to make our own image data? To be an image producer, we have to implement the five methods defined in the `ImageProducer` interface:

- `addConsumer()`
- `startProduction()`
- `isConsumer()`
- `removeConsumer()`
- `requestTopDownLeftRightResend()`

Four methods of `ImageProducer` simply deal with the process of registering consumers. `addConsumer()` takes an `ImageConsumer` as an argument and adds it to the list of consumers. Our producer can then start sending image data to the consumer whenever it's ready. `startProduction()` is identical to `addConsumer()`, except that it asks the producer to start sending data as soon as possible. The difference might be that a given producer would send the current frame of data or initiate construction of a frame immediately, rather than waiting until its next cycle. `isConsumer()` tests whether a particular consumer is already registered, and `removeConsumer()` removes a consumer from the list. We'll see shortly that we can perform these kinds of operations easily with a `Vector`.

An `ImageProducer` also needs to know how to use the `ImageConsumer` interface of its clients. The final method of the `ImageProducer` interface, `requestTopDownLeftRightResend()`, asks that the image data be resent to the consumer, in order, from beginning to end. In general, a producer can generate pixel data and send it to the consumer in any order that it likes. The `setPixels()` method of the

ImageConsumer interface takes parameters telling the consumer what part of the image is being delivered on each call. A call to requestTopDownLeftRightResend() asks the producer to send the pixel data again, in order. A consumer might do this so that it can use a higher quality conversion algorithm that relies on receiving the pixel data in sequence. It's important to note that the producer is allowed to ignore this request; it doesn't have to be able to send the data in sequence.

Color Models

Everybody wants to work with color in their application, but using color raises problems. The most important problem is simply how to represent a color. There are many different ways to encode color information: red, green, blue (RGB) values; hue, saturation, value (HSV); hue, lightness, saturation (HLS); and more. In addition, you can provide full color information for each pixel, or you can just specify an index into a color table (palette) for each pixel. The way you represent a color is called a *color model*. AWT provides tools for two broad groups of color models: *direct* and *indexed*.

As you might expect, you need to specify a color model in order to generate pixel data; the abstract class java.awt.image.ColorModel represents a color model. A ColorModel is one of the arguments to the setPixels() method an image producer calls to deliver pixels to a consumer. What you probably wouldn't expect is that you can use a different color model every time you call setPixels(). Exactly why you'd do this is another matter. Most of the time, you'll want to work with a single color model; that model will probably be the default direct color model. But the additional flexibility is there if you need it.

By default, the core AWT components use a direct color model called ARGB. The A stands for "alpha," which is the historical name for transparency. RGB refers to the red, green, and blue color components that are combined to produce a single, composite color. In the default ARGB model, each pixel is represented by a 32-bit integer that is interpreted as four 8-bit fields; in order, the fields represent the transparency (A), red, green, and blue components of the color, as shown in Figure 1-3.

To create an instance of the default ARGB model, call the static getRGBdefault() method in ColorModel. This method returns a DirectColorModel object; DirectColorModel is a subclass of ColorModel. You can also create other direct color models by calling a DirectColorModel constructor, but you shouldn't need to unless you have a fairly exotic application.

In an indexed color model, each pixel is represented by a smaller amount of information: an index into a table of real color values. For some applications, generating data with an indexed model may be more convenient. If you have an 8-bit display or smaller, using an indexed model may be more efficient, since your

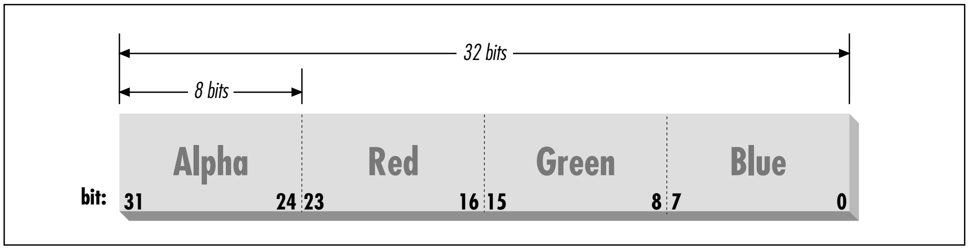

Figure 1–3: ARGB color encoding

hardware is internally using an indexed color model of some form.

While AWT provides `IndexedColorModel` objects, we won't cover them in this book. It's sufficient to work with the `DirectColorModel`. Even if you have an 8-bit display, the Java implementation on your platform should accommodate the hardware you have and, if necessary, dither colors to fit your display. Java also produces transparency on systems that don't natively support it by dithering colors.

Creating an Image

Let's take a look at producing some image data. A picture may be worth a thousand words, but fortunately, we can generate a picture in significantly fewer than a thousand words of Java. If we just want to render image frames byte by byte, we can use a utility class that acts as an `ImageProducer` for us.

`java.awt.image.MemoryImageSource` is a simple utility class that implements the `ImageProducer` interface; we give it pixel data in an array and it sends that data to an image consumer. A `MemoryImageSource` can be constructed for a given color model, with various options to specify the type and positioning of its data. We'll use the simplest form, which assumes an ARGB color model.

The following applet, `ColorPan`, creates an image from an array of integers holding ARGB pixel values:

```
import java.awt.*;
import java.awt.image.*;
public class ColorPan extends java.applet.Applet {
    Image img;
    int width, height;
    int [] pixData;
    public void init() {
        width = getSize().width;
        height = getSize().height;
        pixData = new int [width * height];
        int i=0;
        for (int y = 0; y < height; y++) {
            int red = (y * 255) / (height - 1);
```

```
            for (int x = 0; x < width; x++) {
                int green = (x * 255) / (width - 1);
                int blue = 128;
                int alpha = 255;
                pixData[i++] = (alpha << 24) | (red << 16) |
                               (green << 8 ) | blue;
            }
        }
    }
    public void paint( Graphics g ) {
        if ( img == null )
            img = createImage( new MemoryImageSource(width, height,
                                         pixData, 0, width));
        g.drawImage( img, 0, 0, this );
    }
}
```

Give it a try. The size of the image is determined by the size of the applet when it starts up. You should get a very colorful box that pans from deep blue at the upper left corner to bright yellow at the bottom right, with green and red at the other extremes.

We create the pixel data for our image in the `init()` method and then use `MemoryImageSource` to create and display the image in `paint()`. The variable `pixData` is a one-dimensional array of integers that holds 32-bit ARGB pixel values. In `init()` we loop over every pixel in the image and assign it an ARGB value. The alpha (transparency) component is always 255, which means the image is opaque. The blue component is always 128, half its maximum intensity. The red component varies from 0 to 255 along the y axis; likewise, the green component varies from 0 to 255 along the x axis. The line below combines these components into an ARGB value:

```
pixData[i++] = (alpha << 24) | (red << 16) | (green << 8 ) | blue;
```

The bitwise left-shift operator (<<) should be familiar to C programmers. It simply shoves the bits over by the specified number of positions. The alpha value takes the top byte of the integer, followed by the red, green, and blue values.

When we construct the `MemoryImageSource` as a producer for this data, we give it five parameters: the width and height of the image to construct (in pixels), the `pixData` array, an offset into that array, and the width of each scan line (in pixels). We'll start with the first element (offset 0) of `pixData`; the width of each scan line and the width of the image are the same. The array `pixData` has `width * height` elements, which means it has one element for each pixel.

We create the actual image once, in `paint()`, using the `createImage()` method that our applet inherits from `Component`. In the double-buffering and off-screen drawing examples, we used `createImage()` to give us an empty off-screen image buffer. Here we use `createImage()` to generate an image from a specified `ImageProducer`. `createImage()` creates the `Image` object and receives pixel data from the producer to construct the image. Note that there's nothing particularly special about `MemoryImageSource`; we could use any object that implements the image-producer interface inside of `createImage()`, including one we wrote ourselves. Once we have the image, we can draw it on the display with the familiar `drawImage()` method.

Updating a MemoryImageSource

`MemoryImageSource` can also be used to generate a sequence of images or to update an image dynamically. In Java 1.1, this is probably the easiest way to build your own low-level animation software. This example simulates the static on a television screen. It generates successive frames of random black and white pixels and displays each frame when it is complete. Figure 1-4 shows one frame of random static, followed by the code:

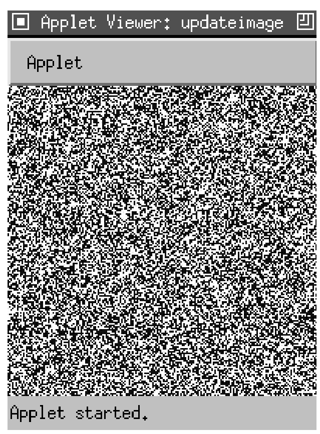

Figure 1–4: A frame of random static

```
import java.awt.*;
import java.awt.image.*;
public class StaticGenerator
        extends java.applet.Applet
        implements Runnable {
    int arrayLength, pixels[];
    MemoryImageSource source;
    Image image;
    int width, height;
    public void init() {
```

```
        width = getSize().width; height = getSize().height;
        arrayLength = width * height;
        pixels = new int [arrayLength];
        source = new MemoryImageSource(width, height, pixels, 0, width);
        source.setAnimated(true);
        image = createImage(source);
        new Thread(this).start();
    }
    public void run() {
        while (true) {
            try {
                Thread.sleep(1000/24);
            } catch( InterruptedException e ) { /* die */ }
            for (int x = 0; x < width; x++)
                for (int y = 0; y < height; y++)  {
                    boolean rand = Math.random() > 0.5;
                    pixels[y*width+x] =
                        rand ? Color.black.getRGB() : Color.white.getRGB();
                }
            // Push out the new data
            source.newPixels(0, 0, width, height);
        }
    }
    public void paint( Graphics g ) {
        g.drawImage(image, 0, 0, this);
    }
}
```

The `init()` method sets up the `MemoryImageSource` that produces the sequence of images. It first computes the size of the array needed to hold the image data. It then creates a `MemoryImageSource` object that produces images the width and height of the display, using the default color model (the constructor we use assumes that we want the default). We start taking pixels from the beginning of the pixel array, and scan lines in the array have the same width as the image. Once we have created the `MemoryImageSource`, we call its `setAnimated()` method to tell it that we will be generating an image sequence. Then we use the source to create an `Image` that will display our sequence.

We next start a thread that generates the pixel data. For every element in the array, we get a random number and set the pixel to black if the random number is greater than 0.5. Because `pixels` is an `int` array, we can't assign `Color` objects to it directly; we use `getRGB()` to extract the color components from the black and white `Color` constants. When we have filled the entire array with data, we call the `newPixels()` method, which delivers the new data to the image.

That's about all there is. We provide a very uninteresting `paint()` method that just calls `drawImage()` to put the current state of the image on the screen. Whenever `paint()` is called, we see the latest collection of static. The image observer, which is the `Applet` itself, schedules a call to `paint()` whenever anything interesting has happened to the image. It's worth noting how simple it is to create this animation. Once we have the `MemoryImageSource`, we use it to create an image that we treat like any other image. The code that generates the image sequence can be arbitrarily complex—certainly in any reasonable example, it would be more complex than our (admittedly cheesy) static. But that complexity never infects the simple task of getting the image on the screen and updating it.

Image Producers and Consumers

In this section we'll create an image producer that generates a stream of image frames rather than just a static image. Unfortunately, it would take too many lines of code to generate anything really interesting, so we'll stick with a simple modification of our `ColorPan` example. After all, figuring out what to display is your job; I'm primarily concerned with giving you the necessary tools. After this, you should have the needed tools to implement more interesting applications.

A word of advice: if you find yourself writing image producers, you're probably making your life excessively difficult. Most situations can be handled by the dynamic `MemoryImageSource` technique that we just demonstrated. Before going to the trouble of writing an image producer, convince yourself that there isn't a simpler solution. Even if you never write an image producer yourself, it's good (like Motherhood and Apple Pie) to understand how Java's image-rendering tools work.

Image Consumers

First, we have to know a little more about the image consumers we'll be feeding. An image consumer implements the seven methods that are defined in the `ImageConsumer` interface. Two of these methods are overloaded versions of the `setPixels()` method that accept the actual pixel data for a region of the image. They are identical except that one takes the pixel data as an array of integers, and the other uses an array of bytes. (An array of bytes is natural when you're using an indexed color model because each pixel is specified by an index into a color array.) A call to `setPixels()` looks something like:

```
setPixels(x, y, width, height, colorModel, pixels, offset, scanLength);
```

`pixels` is the one-dimensional array of bytes or integers that holds the pixel data. Often, you deliver only part of the image with each call to `setPixels()`. The `x`, `y`, `width`, and `height` values define the rectangle of the image for which pixels are being delivered. `x` and `y` specify the upper left-hand corner of the chunk you're delivering, relative to the upper left-hand corner of the image as a whole. `width` specifies the width in pixels of the chunk; `height` specifies the number of scan lines in the chunk. `offset` specifies the point in `pixels` at which the data being delivered in this call to `setPixels()` starts. Finally, `scanLength` indicates the width of the entire image, which is not necessarily the same as `width`. The `pixels` array must be large enough to accommodate `width*length+offset` elements; if it's larger, any leftover data is ignored.

We haven't said anything yet about the `colorModel` argument to `setPixels()`. In our previous example, we drew our image using the default ARGB color model for pixel values; the version of the `MemoryImageSource` constructor that we used supplied the default color model for us. In this example, we also stick with the default model, but this time we have to specify it explicitly. The remaining five methods of the `ImageConsumer` interface accept general attributes and framing information about the image:

- `setHints()`
- `setDimensions()`
- `setProperties()`
- `setColorModel()`
- `imageComplete()`

Before delivering any data to a consumer, the producer should call the consumer's `setHints()` method to pass it information about how pixels will be delivered. Hints are specified in the form of flags defined in the `ImageConsumer` interface. The flags are described in Table 1-2. The consumer uses these hints to optimize the way it builds the image; it's also free to ignore them.

Table 1–2: ImageConsumer setHints() Flags

Flag	Description
`RANDOMPIXELORDER`	The pixels are delivered in random order
`TOPDOWNLEFTRIGHT`	The pixels are delivered from top to bottom, left to right
`COMPLETESCANLINES`	Each call to `setPixels()` delivers one or more complete scan lines
`SINGLEPASS`	Each pixel is delivered only once
`SINGLEFRAME`	The pixels define a single, static image

`setDimensions()` is called to pass the width and height of the image when they are known.

`setProperties()` is used to pass a hashtable of image properties, stored by name. This method isn't particularly useful without some prior agreement between the producer and consumer about what properties are meaningful. For example, image formats such as GIF and TIFF can include additional information about the image. These image attributes could be delivered to the consumer in the hashtable.

`setColorModel()` is called to tell the consumer which color model will be used to process most of the pixel data. However, remember that each call to `setPixels()` also specifies a `ColorModel` for its group of pixels. The color model specified in `setColorModel()` is really only a hint that the consumer can use for optimization. You're not required to use this color model to deliver all (or for that matter, any) of the pixels in the image.

The producer calls the consumer's `imageComplete()` method when it has completely delivered the image or a frame of an image sequence. If the consumer doesn't wish to receive further frames of the image, it should unregister itself from the producer at this point. The producer passes a status flag formed from the flags shown in Table 1-3.

Table 1–3: ImageConsumer imageComplete() Flags

Flag	Description
`STATICIMAGEDONE`	A single static image is complete
`SINGLEFRAMEDONE`	One frame of an image sequence is complete
`IMAGEERROR`	An error occurred while generating the image

As you can see, the `ImageProducer` and `ImageConsumer` interfaces provide a very flexible mechanism for distributing image data. Now let's look at a simple producer.

A Sequence of Images

The following class, `ImageSequence`, shows how to implement an `ImageProducer` that generates a sequence of images. The images are a lot like the `ColorPan` image we generated a few pages back, except that the blue component of each pixel changes with every frame. This image producer doesn't do anything you couldn't do with a `MemoryImageSource`. It reads ARGB data from an array and consults the object that creates the array to give it an opportunity to update the data between each frame.

This is a complex example, so before diving into the code, let's take a broad look at the pieces. The `ImageSequence` class is an image producer; it generates data and sends it to image consumers to be displayed. To make our design more modular, we define an interface called `FrameARGBData` that describes how our rendering code provides each frame of ARGB pixel data to our producer. To do the computation and provide the raw bits, we create a class called `ColorPanCycle` that implements `FrameARGBData`. This means that `ImageSequence` doesn't care specifically where the data comes from; if we wanted to draw different images, we could just drop in another class, provided that the new class implements `FrameARGBData`. Finally, we create an applet called `UpdatingImage` that includes two image consumers to display the data.

Here's the `ImageSequence` class:

```
import java.awt.image.*;
import java.util.*;
public class ImageSequence extends Thread implements ImageProducer {
    int width, height, delay;
    ColorModel model = ColorModel.getRGBdefault();
    FrameARGBData frameData;
    private Vector consumers = new Vector();
    public void run() {
        while ( frameData != null ) {
            frameData.nextFrame();
            sendFrame();
            try {
                sleep( delay );
            } catch ( InterruptedException e ) {}
        }
    }
    public ImageSequence(FrameARGBData src, int maxFPS ) {
        frameData = src;
        width = frameData.size().width;
        height = frameData.size().height;
        delay = 1000/maxFPS;
        setPriority( MIN_PRIORITY + 1 );
    }
    public synchronized void addConsumer(ImageConsumer c) {
        if ( isConsumer( c ) )
            return;
        consumers.addElement( c );
        c.setHints(ImageConsumer.TOPDOWNLEFTRIGHT |
                 ImageConsumer.SINGLEPASS );
        c.setDimensions( width, height );
        c.setProperties( new Hashtable() );
        c.setColorModel( model );
    }
    public synchronized boolean isConsumer(ImageConsumer c) {
        return ( consumers.contains( c ) );
    }
    public synchronized void removeConsumer(ImageConsumer c) {
        consumers.removeElement( c );
    }
    public void startProduction(ImageConsumer ic) {
        addConsumer(ic);
    }
```

```
        public void requestTopDownLeftRightResend(ImageConsumer ic) { }
        private void sendFrame() {
            for ( Enumeration e = consumers.elements(); e.hasMoreElements();  ) {
                ImageConsumer c = (ImageConsumer)e.nextElement();
                c.setPixels(0, 0, width, height, model, frameData.getPixels(),
                            0, width);
                c.imageComplete(ImageConsumer.SINGLEFRAMEDONE);
            }
        }
    }
```

The bulk of the code in `ImageSequence` creates the skeleton we need for implementing the `ImageProducer` interface. `ImageSequence` is actually a simple subclass of `Thread` whose `run()` method loops, generating and sending a frame of data on each iteration. The `ImageSequence` constructor takes two items: a `FrameARGBData` object that updates the array of pixel data for each frame, and an integer that specifies the maximum number of frames per second to generate. We give the thread a low priority (`MIN_PRIORITY+1`) so that it can't run away with all of our CPU time.

Our `FrameARGBData` object implements the following interface:

```
    interface FrameARGBData {
        java.awt.Dimension size();
        int [] getPixels();
        void nextFrame();
    }
```

In `ImageSequence`'s `run()` method, we call `nextFrame()` to compute the array of pixels for each frame. After computing the pixels, we call our own `sendFrame()` method to deliver the data to the consumers. `sendFrame()` calls `getPixels()` to retrieve the updated array of pixel data from the `FrameARGBData` object. `sendFrame()` then sends the new data to all of the consumers by invoking each of their `setPixels()` methods and signaling the end of the frame with `imageComplete()`. Note that `sendFrame()` can handle multiple consumers; it iterates through a `Vector` of image consumers. In a more realistic implementation, we would also check for errors and notify the consumers if any occurred.

The business of managing the `Vector` of consumers is handled by `addConsumer()` and the other methods in the `ImageProducer` interface. `addConsumer()` adds an item to `consumers`. A `Vector` is a perfect tool for this task, since it's an automatically extendable array, with methods for finding out how many elements it has, whether or not a given element is already a member, and so on.

`addConsumer()` also gives the consumer hints about how the data will be delivered by calling `setHints()`. This image provider always works from top to bottom and left to right, and makes only one pass through the data. `addConsumer()` next gives the consumer an empty hashtable of image properties. Finally, it reports that most of the pixels will use the default ARGB color model (we initialized the variable `model` to `ColorModel.getRGBDefault()`). In this example, we always start sending image data on the next frame, so `startProduction()` simply calls `addConsumer()`.

We've discussed the mechanism for communications between the consumer and producer, but I haven't yet told you where the data comes from. We have a `FrameARGBData` interface that defines how to retrieve the data, but we don't yet have an object that implements the interface. The following class, `ColorPanCycle`, implements `FrameARGBData`; we'll use it to generate our pixels:

```
import java.awt.*;
class ColorPanCycle implements FrameARGBData {
    int frame = 0, width, height;
    private int [] pixels;
    ColorPanCycle ( int w, int h ) {
        width = w;
        height = h;
        pixels = new int [ width * height ];
        nextFrame();
    }
    public synchronized int [] getPixels() {
        return pixels;
    }
    public synchronized void nextFrame() {
        int index = 0;
        for (int y = 0; y < height; y++) {
            for (int x = 0; x < width; x++) {
                int red = (y * 255) / (height - 1);
                int green = (x * 255) / (width - 1);
                int blue = (frame * 10) & 0xff;
                pixels[index++] =
                    (255 << 24) | (red << 16) | (green << 8) | blue;
            }
        }
        frame++;
    }
    public Dimension size() {
        return new Dimension ( width, height );
    }
}
```

`ColorPanCycle` is like our previous `ColorPan` example, except that it adjusts each pixel's blue component each time `nextFrame()` is called. This should produce a color cycling effect; as time goes on, the image becomes more blue.

Now let's put the pieces together by writing an applet that displays a sequence of changing images: `UpdatingImage`. In fact, we'll do better than displaying one sequence. To prove that `ImageSequence` really can deal with multiple consumers, UpdatingImage creates two components that display different views of the image. Once the mechanism has been set up, it's surprising how little code you need to add additional displays.

```
import java.awt.*;
import java.awt.image.*;
public class UpdatingImage extends java.applet.Applet {
    ImageSequence seq;
    public void init() {
        seq = new ImageSequence( new ColorPanCycle(100, 100), 10);
        setLayout( null );
        add( new ImageCanvas( seq, 50, 50 ) );
        add( new ImageCanvas( seq, 100, 100 ) );
        seq.start();
    }
    public void stop() {
        if ( seq != null ) {
            seq.stop();
            seq = null;
        }
    }
}
class ImageCanvas extends Canvas {
    Image img;
    ImageProducer source;
    ImageCanvas ( ImageProducer p, int w, int h ) {
        source = p;
        setSize( w, h );
    }
    public void update( Graphics g ) {
        paint(g);
    }
    public void paint( Graphics g ) {
        if ( img == null )
            img = createImage( source );
        g.drawImage( img, 0, 0, getSize().width, getSize().height, this );
    }
}
```

UpdatingImage constructs a new ImageSequence producer with an instance of our ColorPanCycle object as its frame source. It then creates two ImageCanvas components that create and display the two views of our animation. ImageCanvas is a subclass of Canvas; it takes an ImageProducer and a width and height in its constructor and creates and displays an appropriately scaled version of the image in its paint() method. UpdatingImage places the smaller view on top of the larger one for a sort of "picture in picture" effect.

If you've followed the example to this point, you're probably wondering where in the heck is the image consumer. After all, we spent a lot of time writing methods in

ImageSequence for the consumer to call. If you look back at the code, you'll see that an ImageSequence object gets passed to the ImageCanvas constructor, and that this object is used as an argument to createImage(). But nobody appears to call addConsumer(). And the image producer calls setPixels() and other consumer methods; but it always digs a consumer out of its Vector of registered consumers, so we never see where these consumers come from.

In UpdatingImage, the image consumer is behind the scenes, hidden deep inside the Canvas—in fact, inside the Canvas' peer. The call to createImage() tells its component (i.e., our canvas) to become an image consumer. Something deep inside the component is calling addConsumer() behind our backs and registering a mysterious consumer, and that consumer is the one the producer uses in calls to setPixels() and other methods. We haven't implemented any ImageConsumer objects in this book because, as you might imagine, most image consumers are implemented in native code, since they need to display things on the screen. There are others though; the java.awt.image.PixelGrabber class is a consumer that returns the pixel data as a byte array. You might use it to save an image. You can make your own consumer do anything you like with pixel data from a producer. But in reality, you rarely need to write an image consumer yourself. Let them stay hidden; take it on faith that they exist.

Now for the next question: How does the screen get updated? Even though we are updating the consumer with new data, the new image will not appear on the display unless the applet repaints it periodically. By now, this part of the machinery should be familiar: what we need is an image observer. Remember that all components are image observers (i.e., the class Component implements ImageObserver). The call to drawImage() specifies our ImageCanvas as its image observer. The default Component class-image-observer functionality then repaints our image whenever new pixel data arrives.

In this example, we haven't bothered to stop and start our applet properly; it continues running and wasting CPU time even when it's invisible. There are two strategies for stopping and restarting our thread. We can destroy the thread and create a new one, which would require recreating our ImageCanvas objects, or we could suspend and resume the active thread. Neither option is particularly difficult.

Filtering Image Data

As I said earlier, you rarely need to write an image consumer. However, there is one kind of image consumer that's worth knowing about. In this final section on images, we'll build a simple image filter. An image filter is simply a class that performs some work on image data before passing the data to another consumer.

The ColorSep applet acquires an image; uses an image filter to separate the image into red, green, and blue components; and displays the three resulting images. With this applet and a few million dollars, you could build your own color separation plant.

```
import java.awt.*;
import java.awt.image.*;
public class ColorSep extends java.applet.Applet {
    Image img, redImg, greenImg, blueImg;
    public void init() {
        img = getImage( getClass().getResource( getParameter("img")) );
        redImg = createImage(new FilteredImageSource(img.getSource(),
                                new ColorMaskFilter( Color.red )));
        greenImg = createImage(new FilteredImageSource(img.getSource(),
                                new ColorMaskFilter( Color.green )));
        blueImg = createImage(new FilteredImageSource(img.getSource(),
                                new ColorMaskFilter( Color.blue )));
    }
    public void paint( Graphics g ) {
        int width = getSize().width, height = getSize().height;
        g.drawImage( redImg, 0, 0, width/3, height, this );
        g.drawImage( greenImg, width/3, 0, width/3, height, this );
        g.drawImage( blueImg, 2*width/3, 0, width/3, height, this );
    }
}
class ColorMaskFilter extends RGBImageFilter {
    Color color;
    ColorMaskFilter( Color mask ) {
        color = mask;
        canFilterIndexColorModel = true;
    }
    public int filterRGB(int x, int y, int pixel ) {
        return
            255 << 24 |
            (((pixel & 0xff0000) >> 16) * color.getRed()/255) << 16 |
            (((pixel & 0xff00) >> 8) * color.getGreen()/255) << 8 |
            (pixel & 0xff) * color.getBlue()/255 ;
    }
}
```

The FilteredImageSource and RGBImageFilter classes form the basis for building and using image filters. A FilteredImageSource is an image producer (like MemoryImageSource) that is constructed from an image and an ImageFilter object. It fetches pixel data from the image and feeds it through the image filter before passing the data along. Because FilteredImageSource is an image producer, we can use it in our calls to createImage().

But where's the consumer? FilteredImageSource obviously consumes image data as well as producing it. The image consumer is still mostly hidden, but is peeking out from under its rock. Our class ColorMaskFilter extends RGBImageFilter, which in turn extends ImageFilter. And ImageFilter is (finally!) an image consumer. Of course, we still don't see the calls to addConsumer(), and we don't see an implementation of setPixels(); they're hidden in the ImageFilter sources and inherited by ColorMaskFilter.

So what does ColorMaskFilter actually do? Not much. ColorMaskFilter is a simple subclass of RGBImageFilter that implements one method, filterRGB(), through which all of the pixel data are fed. Its constructor saves a mask value we use for filtering. The filterRGB() method accepts a pixel value, along with its x and y coordinates, and returns the filtered version of the pixel. In ColorMaskFilter, we simply multiply the color components by the mask color to get the proper effect. A more complex filter, however, might use the coordinates to change its behavior based on the pixel's position.

One final detail: the constructor for ColorMaskFilter sets the flag canFilterIndexColorModel. This flag is inherited from RGBImageFilter. It means our filter doesn't depend on the pixel's position. In turn, this means it can filter the colors in a color table. If we were using an indexed color model, filtering the color table would be much faster than filtering the individual pixels.