
12

In this chapter:

- *ImageObserver*
- *ColorModel*
- *ImageProducer*
- *ImageConsumer*
- *ImageFilter*

Image Processing

The image processing parts of Java are buried within the `java.awt.image` package. The package consists of three interfaces and eleven classes, two of which are abstract. They are as follows:

- The `ImageObserver` interface provides the single method necessary to support the asynchronous loading of images. The interface implementers watch the production of an image and can react when certain conditions arise. We briefly touched on `ImageObserver` when we discussed the `Component` class (in Chapter 5, *Components*), because `Component` implements the interface.
- The `ImageConsumer` and `ImageProducer` interfaces provide the means for low level image creation. The `ImageProducer` provides the source of the pixel data that is used by the `ImageConsumer` to create an `Image`.
- The `PixelGrabber` and `ImageFilter` classes, along with the `AreaAveragingScaleFilter`, `CropImageFilter`, `RGBImageFilter`, and `ReplicateScaleFilter` subclasses, provide the tools for working with images. `PixelGrabber` consumes pixels from an `Image` into an array. The `ImageFilter` classes modify an existing image to produce another `Image` instance. `CropImageFilter` makes smaller images; `RGBImageFilter` alters pixel colors, while `AreaAveragingScaleFilter` and `ReplicateScaleFilter` scale images up and down using different algorithms. All of these classes implement `ImageConsumer` because they take pixel data as input.
- `MemoryImageSource` and `FilteredImageSource` produce new images. `MemoryImageSource` takes an array and creates an image from it. `FilteredImageSource` uses an `ImageFilter` to read and modify data from another image and produces the new image based on the original. Both `MemoryImageSource` and `FilteredImageSource` implement `ImageProducer` because they produce new pixel data.

- `ColorModel` and its subclasses, `DirectColorModel` and `IndexColorModel`, provide the palette of colors available when creating an image or tell you the palette used when using `PixelGrabber`.

The classes in the `java.awt.image` package let you create `Image` objects at runtime. These classes can be used to rotate images, make images transparent, create image viewers for unsupported graphics formats, and more.

12.1 *ImageObserver*

As you may recall from Chapter 2, *Simple Graphics*, the last parameter to the `drawImage()` method is the image's `ImageObserver`. However, in Chapter 2 I also said that you can use `this` as the image observer and forget about it. Now it's time to ask the obvious questions: what is an image observer, and what is it for?

Because `getImage()` acquires an image asynchronously, the entire `Image` object might not be fully loaded when `drawImage()` is called. The `ImageObserver` interface provides the means for a component to be told asynchronously when additional information about the image is available. The `Component` class implements the `imageUpdate()` method (the sole method of the `ImageObserver` interface), so that method is inherited by any component that renders an image. Therefore, when you call `drawImage()`, you can pass `this` as the final argument; the component on which you are drawing serves as the `ImageObserver` for the drawing process. The communication between the image observer and the image consumer happens behind the scenes; you never have to worry about it, unless you want to write your own `imageUpdate()` method that does something special as the image is being loaded.

If you call `drawImage()` to display an image created in local memory (either for double buffering or from a `MemoryImageSource`), you can set the `ImageObserver` parameter of `drawImage()` to `null` because no asynchrony is involved; the entire image is available immediately, so an `ImageObserver` isn't needed.

12.1.1 *ImageObserver Interface*

Constants

The various flags associated with the `ImageObserver` are used for the `infoFlags` argument to `imageUpdate()`. The flags indicate what kind of information is available and how to interpret the other arguments to `imageUpdate()`. Two or more flags are often combined (by an OR operation) to show that several kinds of information are available.

public static final int WIDTH

When the `WIDTH` flag is set, the `width` argument to `imageUpdate()` correctly indicates the image's width. Subsequent calls to `getWidth()` for the `Image` return the valid image width. If you call `getWidth()` before this flag is set, expect it to return `-1`.

public static final int HEIGHT

When the `HEIGHT` flag is set, the `height` argument to `imageUpdate()` correctly indicates the image's height. Subsequent calls to `getHeight()` for the `Image` return the valid image height. If you call `getHeight()` before this flag is set, expect it to return `-1`.

public static final int PROPERTIES

When the `PROPERTIES` flag is set, the image's properties are available. Subsequent calls to `getProperty()` return valid image properties.

public static final int SOMEBITS

When the `SOMEBITS` flag of `infoflags` (from `imageUpdate()`) is set, the image has started loading and at least some of its content are available for display. When this flag is set, the `x`, `y`, `width`, and `height` arguments to `imageUpdate()` indicate the bounding rectangle for the portion of the image that has been delivered so far.

public static final int FRAMEBITS

When the `FRAMEBITS` flag of `infoflags` is set, a complete frame of a multi-frame image has been loaded and can be drawn. The remaining parameters to `imageUpdate()` should be ignored (`x`, `y`, `width`, `height`).

public static final int ALLBITS

When the `ALLBITS` flag of `infoflags` is set, the image has been completely loaded and can be drawn. The remaining parameters to `imageUpdate()` should be ignored (`x`, `y`, `width`, `height`).

public static final int ERROR

When the `ERROR` flag is set, the production of the image has stopped prior to completion because of a severe problem. `ABORT` may or may not be set when `ERROR` is set. Attempts to reload the image will fail. You might get an `ERROR` because the `URL` of the `Image` is invalid (file not found) or the image file itself is invalid (invalid size/content).

public static final int ABORT

When the `ABORT` flag is set, the production of the image has aborted prior to completion. If `ERROR` is not set, a subsequent attempt to draw the image may succeed. For example, an image would abort without an error if a network error occurred (e.g., a timeout on the `HTTP` connection).

Method

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

The `imageUpdate()` method is the sole method in the `ImageObserver` interface. It is called whenever information about an image becomes available. To register an image observer for an image, pass an object that implements the `ImageObserver` interface to `getWidth()`, `getHeight()`, `getProperty()`, `prepareImage()`, or `drawImage()`.

The `image` parameter to `imageUpdate()` is the image being rendered on the observer. The `infoflags` parameter is a set of `ImageObserver` flags ORed together to signify the current information available about `image`. The meaning of the `x`, `y`, `width`, and `height` parameters depends on the current `infoflags` settings.

Implementations of `imageUpdate()` should return `true` if additional information about the image is desired; returning `false` means that you don't want any additional information, and consequently, `imageUpdate()` should not be called in the future for this image. The default `imageUpdate()` method returns `true` if neither `ABORT` nor `ALLBITS` are set in the `infoflags`—that is, the method `imageUpdate()` is interested in further information if no errors have occurred and the image is not complete. If either flag is set, `imageUpdate()` returns `false`.

You should not call `imageUpdate()` directly—unless you are developing an `ImageConsumer`, in which case you may find it worthwhile to override the default `imageUpdate()` method, which all components inherit from the `Component` class.

12.1.2 Overriding *imageUpdate*

Instead of bothering with the `MediaTracker` class, you can override the `imageUpdate()` method and use it to notify you when an image is completely loaded. Example 12-1 demonstrates the use of `imageUpdate()`, along with a way to force your images to load immediately. Here's how it works: the `init()` method calls `getImage()` to request image loading at some time in the future. Instead of waiting for `drawImage()` to trigger the loading process, `init()` forces loading to start by calling `prepareImage()`, which also registers an image observer. `prepareImage()` is a method of the `Component` class discussed in Chapter 5.

The `paint()` method doesn't attempt to draw the image until the variable `loaded` is set to `true`. The `imageUpdate()` method checks the `infoflags` argument to see whether `ALLBITS` is set; when it is set, `imageUpdate()` sets `loaded` to `true`, and schedules a call to `paint()`. Thus, `paint()` doesn't call `drawImage()` until the method `imageUpdate()` has discovered that the image is fully loaded.

Example 12–1: imageUpdate Override.

```
import java.applet.*;
import java.awt.*;
import java.awt.image.ImageObserver;
public class imageUpdateOver extends Applet {
    Image image;
    boolean loaded = false;
    public void init () {
        image = getImage (getDocumentBase(), "rosej.jpg");
        prepareImage (image, -1, -1, this);
    }
    public void paint (Graphics g) {
        if (loaded)
            g.drawImage (image, 0, 0, this);
    }
    public void update (Graphics g) {
        paint (g);
    }
    public synchronized boolean imageUpdate (Image image, int infoFlags,
        int x, int y, int width, int height) {
        if ((infoFlags & ImageObserver.ALLBITS) != 0) {
            loaded = true;
            repaint();
            return false;
        } else {
            return true;
        }
    }
}
```

Note that the call to `prepareImage()` is absolutely crucial. It is needed both to start image loading and to register the image observer. If `prepareImage()` were omitted, `imageUpdate()` would never be called, `loaded` would not be set, and `paint()` would never attempt to draw the image. As an alternative, you could use the `MediaTracker` class to force loading to start and monitor the loading process; that approach might give you some additional flexibility.

12.2 ColorModel

A color model determines how colors are represented within AWT. `ColorModel` is an abstract class that you can subclass to specify your own representation for colors. AWT provides two concrete subclasses of `ColorModel` that you can use to build your own color model; they are `DirectColorModel` and `IndexColorModel`. These two correspond to the two ways computers represent colors internally.

Most modern computer systems use 24 bits to represent each pixel. These 24 bits contain 8 bits for each primary color (red, green, blue); each set of 8 bits

represents the intensity of that color for the particular pixel. This arrangement yields the familiar “16 million colors” that you see in advertisements. It corresponds closely to Java’s direct color model.

However, 24 bits per pixel, with something like a million pixels on the screen, adds up to a lot of memory. In the dark ages, memory was expensive, and devoting this much memory to a screen buffer cost too much. Therefore, designers used fewer bits—possibly as few as three, but more often eight—for each pixel. Instead of representing the colors directly in these bits, the bits were an index into a color map. Graphics programs would load the color map with the colors they were interested in and then represent each pixel by using the index of the appropriate color in the map. For example, the value 1 might represent fuschia; the value 2 might represent puce. Full information about how to display each color (the red, green, and blue components that make up fuschia or puce) is contained only in the color map. This arrangement corresponds closely to Java’s indexed color model.

Because Java is platform-independent, you don’t need to worry about how your computer or the user’s computer represents colors. Your programs can use an indexed or direct color map as appropriate. Java will do the best it can to render the colors you request. Of course, if you use 5,000 colors on a computer that can only display 256, Java is going to have to make compromises. It will decide which colors to put in the color map and which colors are close enough to the colors in the color map, but that’s done behind your back.

Java’s default color model uses 8 bits per pixel for red, green, and blue, along with another 8 bits for alpha (transparency) level. However, as I said earlier, you can create your own `ColorModel` if you want to work in some other scheme. For example, you could create a grayscale color model for black and white pictures, or an HSB (hue, saturation, brightness) color model if you are more comfortable working with this system. Your color model’s job will be to take a pixel value in your representation and translate that value into the corresponding alpha, red, green, and blue values. If you are working with a grayscale image, your image producer could deliver grayscale values to the image consumer, plus a `ColorModel` that tells the consumer how to render these gray values in terms of ARGB components.

12.2.1 *ColorModel Methods*

Constructors

public ColorModel (int bits)

There is a single constructor for `ColorModel`. It has one parameter, `bits`, which describes the number of bits required per pixel of an image. Since this is an abstract class, you cannot call this constructor directly. Since each pixel value must be stored within an integer, the maximum value for `bits` is 32. If you request more, you get 32.

Pseudo-constructors

public static ColorModel getRGBdefault()

The `getRGBdefault()` method returns the default `ColorModel`, which has 8 bits for each of the components alpha, red, green, and blue. The order the pixels are stored in an integer is 0xAARRGGBB, or alpha in highest order byte, down to blue in the lowest.

Other methods

public int getPixelSize ()

The `getPixelSize()` method returns the number of bits required for each pixel as described by this color model. That is, it returns the number of bits passed to the constructor.

public abstract int getAlpha (int pixel)

The `getAlpha()` method returns the alpha component of `pixel` for a color model. Its range must be between 0 and 255, inclusive. A value of 0 means the pixel is completely transparent and the background will appear through the pixel. A value of 255 means the pixel is opaque and you cannot see the background behind it.

public abstract int getRed (int pixel)

The `getRed()` method returns the red component of `pixel` for a color model. Its range must be between 0 and 255, inclusive. A value of 0 means the pixel has no red in it. A value of 255 means red is at maximum intensity.

public abstract int getGreen (int pixel)

The `getGreen()` method returns the green component of `pixel` for a color model. Its range must be between 0 and 255, inclusive. A value of 0 means the pixel has no green in it. A value of 255 means green is at maximum intensity.

public abstract int getBlue (int pixel)

The `getBlue()` method returns the blue component of `pixel` for a color model. Its range must be between 0 and 255, inclusive. A value of 0 means the pixel has no blue in it. A value of 255 means blue is at maximum intensity.

public int getRGB(int pixel)

The `getRGB()` method returns the color of `pixel` in the default RGB color model. If a subclass has changed the ordering or size of the different color components, `getRGB()` will return the pixel in the RGB color model (0xAARRGGBB order). In theory, the subclass does not need to override this method, unless it wants to make it final. Making this method final may yield a significant performance improvement.

```
public void finalize ()
```

The garbage collector calls `finalize()` when it determines that the `ColorModel` object is no longer needed. `finalize()` frees any internal resources that the `ColorModel` object has used.

12.2.2 *DirectColorModel*

The `DirectColorModel` class is a concrete subclass of `ColorModel`. It specifies a color model in which each pixel contains all the color information (alpha, red, green, and blue values) explicitly. Pixels are represented by 32-bit (`int`) quantities; the constructor lets you change which bits are allotted to each component.

All of the methods in this class, except constructors, are `final`, because of assumptions made by the implementation. You can create subclasses of `DirectColorModel`, but you can't override any of its methods. However, you should not need to develop your own subclass. Just create an instance of `DirectColorModel` with the appropriate constructor. Any subclassing results in serious performance degradation, because you are going from fast, static `final` method calls to dynamic method lookups.

Constructors

```
public DirectColorModel (int bits, int redMask, int greenMask, int blueMask,  
int alphaMask)
```

This constructor creates a `DirectColorModel` in which `bits` represents the total number of bits used to represent a pixel; it must be less than or equal to 32. The `redMask`, `greenMask`, `blueMask`, and `alphaMask` specify where in a pixel each color component exists. Each of the bit masks must be contiguous (e.g., red cannot be the first, fourth, and seventh bits of the pixel), must be smaller than 2^{bits} , and should not exceed 8 bits. (You cannot display more than 8 bits of data for any color component, but the mask can be larger.) Combined, the masks together should be `bits` in length. The default RGB color model is:

```
new DirectColorModel (32, 0x00ff0000, 0x0000ff00, 0x000000ff, 0xff000000)
```

The run-time exception `IllegalArgumentException` is thrown if any of the following occur:

- The bits that are set in a mask are not contiguous.
- Mask bits overlap (i.e., the same bit is set in two or more masks).
- The number of mask bits exceeds `bits`.

public DirectColorModel (int bits, int redMask, int greenMask, int blueMask)

This constructor for `DirectColorModel` calls the first with an alpha mask of 0, which means that colors in this color model have no transparency component. All colors will be fully opaque with an alpha value of 255. The same restrictions for the red, green, and blue masks apply.

Methods

final public int getAlpha (int pixel)

The `getAlpha()` method returns the alpha component of `pixel` for the color model as a number from 0 to 255, inclusive. A value of 0 means the pixel is completely transparent, and the background will appear through the pixel. A value of 255 means the pixel is opaque, and you cannot see the background behind it.

final public int getRed (int pixel)

The `getRed()` method returns the red component of `pixel` for the color model. Its range is from 0 to 255. A value of 0 means the pixel has no red in it. A value of 255 means red is at maximum intensity.

final public int getGreen (int pixel)

The `getGreen()` method returns the green component of `pixel` for the color model. Its range is from 0 to 255. A value of 0 means the pixel has no green in it. A value of 255 means green is at maximum intensity.

final public int getBlue (int pixel)

The `getBlue()` method returns the blue component of `pixel` for the color model. Its range is from 0 to 255. A value of 0 means the pixel has no blue in it. A value of 255 means blue is at maximum intensity.

final public int getRGB (int pixel)

The `getRGB()` method returns the color of `pixel` in the default RGB color model. If a subclass has changed the ordering or size of the different color components, `getRGB()` will return the pixel in the RGB color model (0xAAR-RGGBB order). The `getRGB()` method in this subclass is identical to the method in `ColorModel` but overrides it to make it final.

Other methods

final public int getAlphaMask ()

The `getAlphaMask()` method returns the `alphaMask` from the `DirectColorModel` constructor (or 0 if constructor did not have `alphaMask`). The `alphaMask` specifies which bits in the pixel represent the alpha transparency component of the color model.

final public int getRedMask ()

The `getRedMask()` method returns the `redMask` from the `DirectColorModel` constructor. The `redMask` specifies which bits in the pixel represent the red component of the color model.

final public int getGreenMask ()

The `getGreenMask()` method returns the `greenMask` from the `DirectColorModel` constructor. The `greenMask` specifies which bits in the pixel represent the green component of the color model.

final public int getBlueMask ()

The `getBlueMask()` method returns the `blueMask` from the `DirectColorModel` constructor. The `blueMask` specifies which bits in the pixel represent the blue component of the color model.

12.2.3 IndexColorModel

The `IndexColorModel` is another concrete subclass of `ColorModel`. It specifies a `ColorModel` that uses a color map lookup table (with a maximum size of 256), rather than storing color information in the pixels themselves. Pixels are represented by an index into the color map, which is at most an 8-bit quantity. Each entry in the color map gives the alpha, red, green, and blue components of some color. One entry in the map can be designated “transparent.” This is called the “transparent pixel”; the alpha component of this map entry is ignored.

All of the methods in this class, except constructors, are final because of assumptions made by the implementation. You shouldn’t need to create subclasses; you can if necessary, but you can’t override any of the `IndexColorModel` methods. Example 12-2 (later in this chapter) uses an `IndexColorModel`.

Constructors

There are two sets of constructors for `IndexColorModel`. The first two constructors use a single-byte array for the color map. The second group implements the color map with separate byte arrays for each color component.

public IndexColorModel (int bits, int size, byte colorMap[], int start, boolean hasAlpha, int transparent)

This constructor creates an `IndexColorModel`. `bits` is the number of bits used to represent each pixel and must not exceed 8. `size` is the number of elements in the map; it must be less than 2^{bits} . `hasAlpha` should be `true` if the color map includes alpha (transparency) components and `false` if it doesn’t. `transparent` is the location of the transparent pixel in the map (i.e., the pixel value that is considered transparent). If there is no transparent pixel, set `transparent` to -1.

The `colorMap` describes the colors used to paint pixels. `start` is the index within the `colorMap` array at which the map begins; prior elements of the array are ignored. An entry in the map consists of three or four consecutive bytes, representing the red, green, blue, and (optionally) alpha components. If `hasalpha` is `false`, a map entry consists of three bytes, and no alpha components are present; if `hasalpha` is `true`, map entries consist of four bytes, and all four components must be present.

For example, consider a pixel whose value is `p`, and a color map with a `hasalpha` set to `false`. Therefore, each element in the color map occupies three consecutive array elements. The red component of that pixel will be located at `colorMap[start + 3*p]`; the green component will be at `colorMap[start + 3*p + 1]`; and so on. The value of `size` may be smaller than 2^{bits} , meaning that there may be pixel values with no corresponding entry in the color map. These pixel values (i.e., $\text{size} \leq p < 2^{\text{bits}}$) are painted with the color components set to 0; they are transparent if `hasalpha` is `true`, opaque otherwise.

If `bits` is too large (greater than 8), `size` is too large (greater than 2^{bits}), or the `colorMap` array is too small to hold the map, the run-time exception `ArrayIndexOutOfBoundsException` will be thrown.

```
public IndexColorModel (int bits, int size, byte colorMap[], int start, boolean hasalpha)
```

This version of the `IndexColorModel` constructor calls the previous constructor with a `transparent` index of -1; that is, there is no transparent pixel. If `bits` is too large (greater than 8), or `size` is too large (greater than 2^{bits}), or the `colorMap` array is too small to hold the map, the run-time exception, `ArrayIndexOutOfBoundsException` will be thrown.

```
public IndexColorModel (int bits, int size, byte red[], byte green[], byte blue[],  
int transparent)
```

The second set of constructors for `IndexColorModel` is similar to the first group, with the exception that these constructors use three or four separate arrays (one per color component) to represent the color map, instead of a single array.

The `bits` parameter still represents the number of bits in a pixel. `size` represents the number of elements in the color map. `transparent` is the location of the transparent pixel in the map (i.e., the pixel value that is considered transparent). If there is no transparent pixel, set `transparent` to -1.

The `red`, `green`, and `blue` arrays contain the color map itself. These arrays must have at least `size` elements. They contain the red, green, and blue components of the colors in the map. For example, if a pixel is at position `p`, `red[p]` contains the pixel's red component; `green[p]` contains the green

component; and `blue[p]` contains the blue component. The value of `size` may be smaller than 2^{bits} , meaning that there may be pixel values with no corresponding entry in the color map. These pixel values (i.e., $\text{size} \leq p < 2^{\text{bits}}$) are painted with the color components set to 0.

If `bits` is too large (greater than 8), `size` is too large (greater than 2^{bits}), or the red, green, and blue arrays are too small to hold the map, the run-time exception `ArrayIndexOutOfBoundsException` will be thrown.

public IndexColorModel (int bits, int size, byte red[], byte green[], byte blue[])

This version of the `IndexColorModel` constructor calls the previous one with a transparent index of -1; that is, there is no transparent pixel. If `bits` is too large (greater than 8), `size` is too large (greater than 2^{bits}), or the red, green, and blue arrays are too small to hold the map, the run-time exception `ArrayIndexOutOfBoundsException` will be thrown.

public IndexColorModel (int bits, int size, byte red[], byte green[], byte blue[], byte alpha[])

Like the previous constructor, this version creates an `IndexColorModel` with no transparent pixel. It differs from the previous constructor in that it supports transparency; the array `alpha` contains the map's transparency values. If `bits` is too large (greater than 8), `size` is too large (greater than 2^{bits}), or the red, green, blue, and alpha arrays are too small to hold the map, the run-time exception `ArrayIndexOutOfBoundsException` will be thrown.

Methods

final public int getAlpha (int pixel)

The `getAlpha()` method returns the alpha component of `pixel` for a color model, which is a number between 0 and 255, inclusive. A value of 0 means the pixel is completely transparent and the background will appear through the pixel. A value of 255 means the pixel is opaque and you cannot see the background behind it.

final public int getRed (int pixel)

The `getRed()` method returns the red component of `pixel` for a color model, which is a number between 0 and 255, inclusive. A value of 0 means the pixel has no red in it. A value of 255 means red is at maximum intensity.

final public int getGreen (int pixel)

The `getGreen()` method returns the green component of `pixel` for a color model, which is a number between 0 and 255, inclusive. A value of 0 means the pixel has no green in it. A value of 255 means green is at maximum intensity.

final public int getBlue (int pixel)

The `getBlue()` method returns the blue component of `pixel` for a color model, which is a number between 0 and 255, inclusive. A value of 0 means the pixel has no blue in it. A value of 255 means blue is at maximum intensity.

final public int getRGB (int pixel)

The `getRGB()` method returns the color of `pixel` in the default RGB color model. If a subclass has changed the ordering or size of the different color components, `getRGB()` will return the pixel in the RGB color model (0xAAR-RGGBB order). This version of `getRGB` is identical to the version in the `ColorModel` class but overrides it to make it final.

Other methods

final public int getMapSize()

The `getMapSize()` method returns the size of the color map (i.e., the number of distinct colors).

final public int getTransparentPixel ()

The `getTransparentPixel()` method returns the color map index for the transparent pixel in the color model. If no transparent pixel exists, it returns -1. It is not possible to change the transparent pixel after the color model has been created.

final public void getAlphas (byte alphas[])

The `getAlphas()` method copies the alpha components of the `ColorModel` into elements 0 through `getMapSize()-1` of the `alphas` array. Space must already be allocated in the `alphas` array.

final public void getReds (byte reds[])

The `getReds()` method copies the red components of the `ColorModel` into elements 0 through `getMapSize()-1` of the `reds` array. Space must already be allocated in the `reds` array.

final public void getGreens (byte greens[])

The `getGreens()` method copies the green components of the `ColorModel` into elements 0 through `getMapSize()-1` of the `greens` array. Space must already be allocated in the `greens` array.

final public void getBlues (byte blues[])

The `getBlues()` method copies the blue components of the `ColorModel` into elements 0 through `getMapSize()-1` of the `blues` array. Space must already be allocated in the `blues` array.

12.3 ImageProducer

The `ImageProducer` interface defines the methods that `ImageProducer` objects must implement. Image producers serve as sources for pixel data; they may compute the data themselves or interpret data from some external source, like a GIF file. No matter how it generates the data, an image producer's job is to hand that data to an image consumer, which usually renders the data on the screen. The methods in the `ImageProducer` interface let `ImageConsumer` objects register their interest in an image. The business end of an `ImageProducer`—that is, the methods it uses to deliver pixel data to an image consumer—are defined by the `ImageConsumer` interface. Therefore, we can summarize the way an image producer works as follows:

- It waits for image consumers to register their interest in an image.
- As image consumers register, it stores them in a `Hashtable`, `Vector`, or some other collection mechanism.
- As image data becomes available, it loops through all the registered consumers and calls their methods to transfer the data.

There's a sense in which you have to take this process on faith; image consumers are usually well hidden. If you call `createImage()`, an image consumer will eventually show up.

Every `Image` has an `ImageProducer` associated with it; to acquire a reference to the producer, use the `getSource()` method of `Image`.

Because an `ImageProducer` must call methods in the `ImageConsumer` interface, we won't show an example of a full-fledged producer until we have discussed `ImageConsumer`.

12.3.1 ImageProducer Interface

Methods

public void addConsumer (ImageConsumer ic)

The `addConsumer()` method registers `ic` as an `ImageConsumer` interested in the `Image` information. Once an `ImageConsumer` is registered, the `ImageProducer` can deliver `Image` pixels immediately or wait until `startProduction()` has been called.

Note that one image may have many consumers; therefore, `addConsumer()` usually stores image consumers in a collection like a `Vector` or `Hashtable`. There is one notable exception: if the producer has the image data in

memory, `addConsumer()` can deliver the image to the consumer immediately. When `addConsumer()` returns, it has finished with the consumer. In this case, you don't need to manage a list of consumers, because there is only one image consumer at a time. (In this case, `addConsumer()` should be implemented as a synchronized method.)

public boolean isConsumer (ImageConsumer ic)

The `isConsumer()` method checks to see if `ic` is a registered `ImageConsumer` for this `ImageProducer`. If `ic` is registered, `true` is returned. If `ic` is not registered, `false` is returned.

public void removeConsumer (ImageConsumer ic)

The `removeConsumer()` method removes `ic` as a registered `ImageConsumer` for this `ImageProducer`. If `ic` was not a registered `ImageConsumer`, nothing should happen. This is not an error that should throw an exception. Once `ic` has been removed from the registry, the `ImageProducer` should no longer send data to it.

public void startProduction (ImageConsumer ic)

The `startProduction()` method registers `ic` as an `ImageConsumer` interested in the `Image` information and tells the `ImageProducer` to start sending the `Image` data immediately. The `ImageProducer` sends the image data to `ic` and all other registered `ImageConsumer` objects, through `addConsumer()`.

public void requestTopDownLeftRightResend (ImageConsumer ic)

The `requestTopDownLeftRightResend()` method is called by the `ImageConsumer` `ic` requesting that the `ImageProducer` retransmit the `Image` data in top-down, left-to-right order. If the `ImageProducer` is unable to send the data in that order or always sends the data in that order (like with `MemoryImageSource`), it can ignore the call.

12.3.2 *FilteredImageSource*

The `FilteredImageSource` class combines an `ImageProducer` and an `ImageFilter` to create a new `Image`. The image producer generates pixel data for an original image. The `FilteredImageSource` takes this data and uses an `ImageFilter` to produce a modified version: the image may be scaled, clipped, or rotated, or the colors shifted, etc. The `FilteredImageSource` is the image producer for the new image. The `ImageFilter` object transforms the original image's data to yield the new image; it implements the `ImageConsumer` interface. We cover the `ImageConsumer` interface in Section 12.4 and the `ImageFilter` class in Section 12.5. Figure 12-1 shows the relationship between an `ImageProducer`, `FilteredImageSource`, `ImageFilter`, and the `ImageConsumer`.

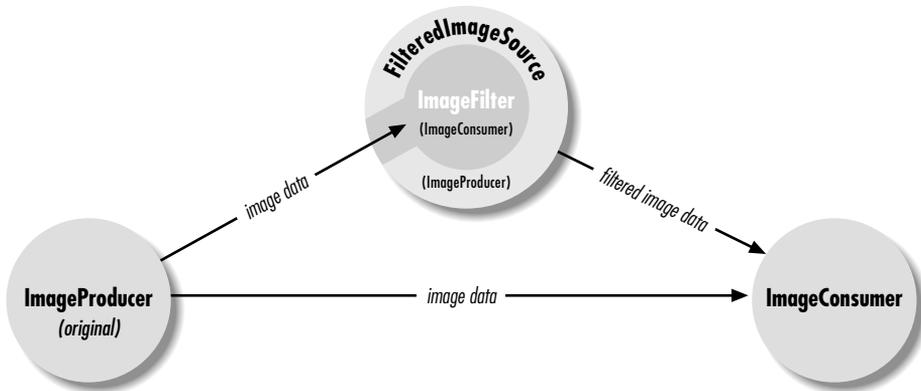


Figure 12–1: Image producers, filters, and consumers

Constructors

public FilteredImageSource (ImageProducer original, ImageFilter filter)

The `FilteredImageSource` constructor creates an image producer that combines an image, `original`, and a filter, `filter`, to create a new image. The `ImageProducer` of the original image is the constructor's first parameter; given an `Image`, you can acquire its `ImageProducer` by using the `getSource()` method. The following code shows how to create a new image from an original. Section 12.5 shows several extensive examples of image filters.

```
Image image = getImage (new URL
    ("http://www.ora.com/graphics/headers/homepage.gif"));
Image newOne = createImage (new FilteredImageSource
    (image.getSource(), new SomeImageFilter()));
```

ImageProducer interface methods

The `ImageProducer` interface methods maintain an internal table for the image consumers. Since this is private, you do not have direct access to it.

public synchronized void addConsumer (ImageConsumer ic)

The `addConsumer()` method adds `ic` as an `ImageConsumer` interested in the pixels for this image.

public synchronized boolean isConsumer (ImageConsumer ic)

The `isConsumer()` method checks to see if `ic` is a registered `ImageConsumer` for this `ImageProducer`. If `ic` is registered, `true` is returned. If not registered, `false` is returned.

```
public synchronized void removeConsumer (ImageConsumer ic)
```

The `removeConsumer()` method removes `ic` as a registered `ImageConsumer` for this `ImageProducer`.

```
public void startProduction (ImageConsumer ic)
```

The `startProduction()` method registers `ic` as an `ImageConsumer` interested in the `Image` information and tells the `ImageProducer` to start sending the `Image` data immediately.

```
public void requestTopDownLeftRightResend (ImageConsumer ic)
```

The `requestTopDownLeftRightResend()` method registers `ic` as an `ImageConsumer` interested in the `Image` information and requests the `ImageProducer` to retransmit the `Image` data in top-down, left-to-right order.

12.3.3 *MemoryImageSource*

The `MemoryImageSource` class allows you to create images completely in memory; you generate pixel data, place it in an array, and hand that array and a `ColorModel` to the `MemoryImageSource` constructor. The `MemoryImageSource` is an image producer that can be used with a consumer to display the image on the screen. For example, you might use a `MemoryImageSource` to display a Mandelbrot image or some other image generated by your program. You could also use a `MemoryImageSource` to modify a pre-existing image; use `PixelGrabber` to get the image's pixel data, modify that data, and then use a `MemoryImageSource` as the producer for the modified image. Finally, you can use `MemoryImageSource` to simplify implementation of a new image type; you can develop a class that reads an image in some unsupported format from a local file or the network; interprets the image file and puts pixel data into an array; and uses a `MemoryImageSource` to serve as an image producer. This is simpler than implementing an image producer yourself, but it isn't quite as flexible; you lose the ability to display partial images as the data becomes available.

In Java 1.1, `MemoryImageSource` supports multiframe images to animate a sequence. In earlier versions, it was necessary to create a dynamic `ImageFilter` to animate the image.

Constructors

There are six constructors for `MemoryImageSource`, each with slightly different parameters. They all create an image producer that delivers some array of data to an image consumer. The constructors are:

```
public MemoryImageSource (int w, int h, ColorModel cm, byte pix[], int off, int scan)  
public MemoryImageSource (int w, int h, ColorModel cm, byte pix[], int off, int scan,  
Hashtable props)
```

```
public MemoryImageSource (int w, int h, ColorModel cm, int pix[],  
int off, int scan)  
public MemoryImageSource (int w, int h, ColorModel cm, int pix[],  
int off, int scan, Hashtable props)  
public MemoryImageSource (int w, int h, int pix[], int off, int scan)  
public MemoryImageSource (int w, int h, int pix[], int off, int scan,  
Hashtable props)
```

The parameters that might be present are:

- w** Width of the image being created, in pixels.
- h** Height of the image being created, in pixels.
- cm** The `ColorModel` that describes the color representation used in the pixel data. If this parameter is not present, the `MemoryImageSource` uses the default RGB color model (`ColorModel.getRGBDefault()`).
- pix[]**
The array of pixel information to be converted into an image. This may be either a `byte` array or an `int` array, depending on the color model. If you're using a direct color model (including the default RGB color model), `pix` is usually an `int` array; if it isn't, it won't be able to represent all 16 million possible colors. If you're using an indexed color model, the array should be a `byte` array. However, if you use an `int` array with an indexed color model, the `MemoryImageSource` ignores the three high-order bytes because an indexed color model has at most 256 entries in the color map. In general: if your color model requires more than 8 bits of data per pixel, use an `int` array; if it requires 8 bits or less, use a `byte` array.
- off**
The first pixel used in the array (usually 0); prior pixels are ignored.
- scan**
The number of pixels per line in the array (usually equal to `w`). The number of pixels per scan line in the array may be larger than the number of pixels in the scan line. Extra pixels in the array are ignored.
- props**
A `Hashtable` of the properties associated with the image. If this argument isn't present, the constructor assumes there are no properties.

The pixel at location (x, y) in the image is located at `pix[y * scan + x + off]`.

ImageProducer interface methods

In Java 1.0, the `ImageProducer` interface methods maintain a single internal variable for the image consumer because the image is delivered immediately and synchronously. There is no need to worry about multiple consumers; as soon as one registers, you give it the image, and you're done. These methods keep track of this single `ImageConsumer`.

In Java 1.1, `MemoryImageSource` supports animation. One consequence of this new feature is that it isn't always possible to deliver all the image's data immediately. Therefore, the class maintains a list of image consumers that are notified when each frame is generated. Since this list is private, you do not have direct access to it.

public synchronized void addConsumer (ImageConsumer ic)

The `addConsumer()` method adds `ic` as an `ImageConsumer` interested in the pixels for this image.

public synchronized boolean isConsumer (ImageConsumer ic)

The `isConsumer()` method checks to see if `ic` is a registered `ImageConsumer` for this `ImageProducer`. If `ic` is registered, `true` is returned. If `ic` is not registered, `false` is returned.

public synchronized void removeConsumer (ImageConsumer ic)

The `removeConsumer()` method removes `ic` as a registered `ImageConsumer` for this `ImageProducer`.

public void startProduction (ImageConsumer ic)

The `startProduction()` method calls `addConsumer()`.

public void requestTopDownLeftRightResend (ImageConsumer ic)

The `requestTopDownLeftRightResend()` method does nothing since in-memory images are already in this format or are multiframed, with each frame in this format.

Animation methods

In Java 1.1, `MemoryImageSource` supports animation; it can now pass multiple frames to interested image consumers. This feature mimics GIF89a's multiframe functionality. (If you have GIF89a animations, you can display them using `getImage()` and `drawImage()`; you don't have to build a complicated creature using `MemoryImageSource`.) . An animation example follows in Example 12-3 (later in this chapter).

public synchronized void setAnimated(boolean animated) ★

The `setAnimated()` method notifies the `MemoryImageSource` if it will be in animation mode (`animated` is `true`) or not (`animated` is `false`). By default, animation is disabled; you must call this method to generate an image sequence.

To prevent losing data, call this method immediately after calling the `MemoryImageSource` constructor.

public synchronized void setFullBufferUpdates(boolean fullBuffers) ★

The `setFullBufferUpdates()` method controls how image updates are done during an animation. It is ignored if you are not creating an animation. If `fullBuffers` is `true`, this method tells the `MemoryImageSource` that it should always send all of an image's data to the consumers whenever it received new data (by a call to `newPixels()`). If `fullBuffers` is `false`, the `MemoryImageSource` sends only the changed portion of the image and notifies consumers (by a call to `ImageConsumer.setHints()`) that frames sent will be complete.

Like `setAnimated()`, `setFullBufferUpdates()` should be called immediately after calling the `MemoryImageSource` constructor, before the animation is started.

To do the actual animation, you update the image array `pix[]` that was specified in the constructor and call one of the overloaded `newPixels()` methods to tell the `MemoryImageSource` that you have changed the image data. The parameters to `newPixels()` determine whether you are animating the entire image or just a portion of the image. You can also supply a new array to take pixel data from, replacing `pix[]`. In any case, `pix[]` supplies the initial image data (i.e., the first frame of the animation).

If you have not called `setAnimated(true)`, calls to any version of `newPixels()` are ignored.

public void newPixels() ★

The version of `newPixels()` with no parameters tells the `MemoryImageSource` to send the entire pixel data (frame) to all the registered image consumers again. Data is taken from the original array `pix[]`. After the data is sent, the `MemoryImageSource` notifies consumers that a frame is complete by calling `imageComplete(ImageConsumer.SINGLEFRAMEDONE)`, thus updating the display when the image is redisplayed. Remember that in many cases, you don't need to update the entire image; updating part of the image saves CPU time, which may be crucial for your application. To update part of the image, call one of the other versions of `newPixels()`.

public synchronized void newPixels(int x, int y, int w, int h) ★

This `newPixels()` method sends part of the image in the array `pix[]` to the consumers. The portion of the image sent has its upper left corner at the point (x, y) , width `w` and height `h`, all in pixels. Changing part of the image rather than the whole thing saves considerably on system resources. Obviously, it is appropriate only if most of the image is still. For example, you could use

this method to animate the steam rising from a cup of hot coffee, while leaving the cup itself static (an image that should be familiar to anyone reading JavaSoft's Web site). After the data is sent, consumers are notified that a frame is complete by a call to `imageComplete(ImageConsumer.SINGLEFRAMEDONE)`, thus updating the display when the image is redisplayed.

If `setFullBufferUpdates()` was called, the entire image is sent, and the dimensions of the bounding box are ignored.

public synchronized void newPixels(int x, int y, int w, int h, boolean frameNotify) ★

This `newPixels()` method is identical to the last, with one exception: consumers are notified that new image data is available only when `frameNotify` is true. This method allows you to generate new image data in pieces, updating the consumers only once when you are finished.

If `setFullBufferUpdates()` was called, the entire image is sent, and the dimensions of the bounding box are ignored.

public synchronized void newPixels(byte[] newpix, ColorModel newmodel, int offset, int scansize) ★

public synchronized void newPixels(int[] newpix, ColorModel newmodel, int offset, int scansize) ★

These `newPixels()` methods change the source of the animation to the byte or int array `newpix[]`, with a `ColorModel` of `newmodel`. `offset` marks the beginning of the data in `newpix` to use, while `scansize` states the number of pixels in `newpix` per line of Image data. Future calls to other versions of `newPixels()` should modify `newpix[]` rather than `pix[]`.

Using MemoryImageSource to create a static image

You can create an image by generating an integer or byte array in memory and converting it to an image with `MemoryImageSource`. The following `MemoryImage` applet generates two identical images that display a series of color bars from left to right. Although the images look the same, they were generated differently: the image on the left uses the default `DirectColorModel`; the image on the right uses an `IndexColorModel`.

Because the image on the left uses a `DirectColorModel`, it stores the actual color value of each pixel in an array of integers (`rgbPixels[]`). The image on the right can use a byte array (`indPixels[]`) because the `IndexColorModel` puts the color information in its color map instead of the pixel array; elements of the pixel array need to be large enough only to address the entries in this map. Images that are based on `IndexColorModel` are generally more efficient in their use of space (integer vs. byte arrays, although `IndexColorModel` requires small support arrays) and in performance (if you filter the image).

The output from this example is shown in Figure 12-2. The source is shown in Example 12-2.

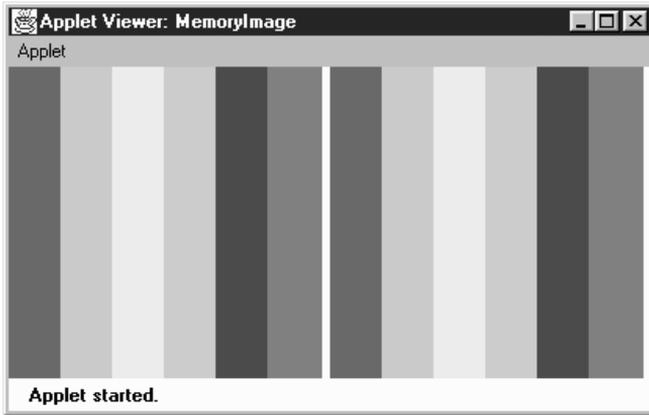


Figure 12-2: *MemoryImage* applet output

Example 12-2: *MemoryImage* Test Program

```
import java.applet.*;
import java.awt.*;
import java.awt.image.*;
public class MemoryImage extends Applet {
    Image i, j;
    int width = 200;
    int height = 200;
    public void init () {
        int rgbPixels[] = new int [width*height];
        byte indPixels[] = new byte [width*height];
        int index = 0;
        Color colorArray[] = {Color.red, Color.orange, Color.yellow,
            Color.green, Color.blue, Color.magenta};
        int rangeSize = width / colorArray.length;
        int colorRGB;
        byte colorIndex;
        byte reds[] = new byte[colorArray.length];
        byte greens[] = new byte[colorArray.length];
        byte blues[] = new byte[colorArray.length];
        for (int i=0;i<colorArray.length;i++) {
            reds[i] = (byte)colorArray[i].getRed();
            greens[i] = (byte)colorArray[i].getGreen();
            blues[i] = (byte)colorArray[i].getBlue();
        }
        for (int y=0;y<height;y++) {
            for (int x=0;x<width;x++) {
                if (x < rangeSize) {
                    colorRGB = Color.red.getRGB();
                    colorIndex = 0;
                } else if (x < (rangeSize*2)) {
```

Example 12–2: MemoryImage Test Program (continued)

```

        colorRGB = Color.orange.getRGB();
        colorIndex = 1;
    } else if (x < (rangeSize*3)) {
        colorRGB = Color.yellow.getRGB();
        colorIndex = 2;
    } else if (x < (rangeSize*4)) {
        colorRGB = Color.green.getRGB();
        colorIndex = 3;
    } else if (x < (rangeSize*5)) {
        colorRGB = Color.blue.getRGB();
        colorIndex = 4;
    } else {
        colorRGB = Color.magenta.getRGB();
        colorIndex = 5;
    }
    rgbPixels[index] = colorRGB;
    indPixels[index] = colorIndex;
    index++;
}
}
i = createImage (new MemoryImageSource (width, height, rgbPixels,
0, width));
j = createImage (new MemoryImageSource (width, height,
new IndexColorModel (8, colorArray.length, reds, greens, blues),
indPixels, 0, width));
}
public void paint (Graphics g) {
    g.drawImage (i, 0, 0, this);
    g.drawImage (j, width+5, 0, this);
}
}
}

```

Almost all of the work is done in `init()` (which, in a real applet, isn't a terribly good idea; ideally `init()` should be lightweight). Previously, we explained the color model's use for the images on the left and the right. Toward the end of `init()`, we create the images `i` and `j` by calling `createImage()` with a `MemoryImageSource` as the image producer. For image `i`, we used the simplest `MemoryImageSource` constructor, which uses the default RGB color model. For `j`, we called the `IndexColorModel` constructor within the `MemoryImageSource` constructor, to create a color map that has only six entries: one for each of the colors we use.

Using MemoryImageSource for animation

As we've seen, Java 1.1 gives you the ability to create an animation using a `MemoryImageSource` by updating the image data in memory; whenever you have finished an update, you can send the resulting frame to the consumers. This technique gives you a way to do animations that consume very little memory, since you keep

overwriting the original image. The applet in Example 12-3 demonstrates `MemoryImageSource`'s animation capability by creating a Mandelbrot image in memory, updating the image as new points are added. Figure 12-3 shows the results, using four consumers to display the image four times.

Example 12-3: Mandelbrot Program

```
// Java 1.1 only
import java.awt.*;
import java.awt.image.*;
import java.applet.*;

public class Mandelbrot extends Applet implements Runnable {
    Thread animator;
    Image im1, im2, im3, im4;
    public void start() {
        animator = new Thread(this);
        animator.start();
    }
    public synchronized void stop() {
        animator = null;
    }
    public void paint(Graphics g) {
        if (im1 != null)
            g.drawImage(im1, 0, 0, null);
        if (im2 != null)
            g.drawImage(im2, 0, getSize().height / 2, null);
        if (im3 != null)
            g.drawImage(im3, getSize().width / 2, 0, null);
        if (im4 != null)
            g.drawImage(im4, getSize().width / 2, getSize().height / 2, null);
    }
    public void update (Graphics g) {
        paint (g);
    }
    public synchronized void run() {
        Thread.currentThread().setPriority(Thread.MIN_PRIORITY);
        int width = getSize().width / 2;
        int height = getSize().height / 2;
        byte[] pixels = new byte[width * height];
        int index = 0;
        int iteration=0;
        double a, b, p, q, psq, qsq, pnew, qnew;
        byte[] colorMap = {(byte)255, (byte)255, (byte)255, // white
                           (byte)0, (byte)0, (byte)0}; // black
        MemoryImageSource mis = new MemoryImageSource(
            width, height,
            new IndexColorModel (8, 2, colorMap, 0, false, -1),
            pixels, 0, width);
        mis.setAnimated(true);
        im1 = createImage(mis);
        im2 = createImage(mis);
        im3 = createImage(mis);
    }
}
```

Example 12-3: Mandelbrot Program (continued)

```

im4 = createImage(mis);
// Generate Mandelbrot
final int ITERATIONS = 16;
for (int y=0; y<height; y++) {
    b = ((double)(y-64))/32;
    for (int x=0; x<width; x++) {
        a = ((double)(x-64))/32;
        p=q=0;
        iteration = 0;
        while (iteration < ITERATIONS) {
            psq = p*p;
            qsq = q*q;
            if ((psq + qsq) >= 4.0)
                break;
            pnew = psq - qsq + a;
            qnew = 2*p*q+b;
            p = pnew;
            q = qnew;
            iteration++;
        }
        if (iteration == ITERATIONS) {
            pixels[index] = 1;
            mis.newPixels(x, y, 1, 1);
            repaint();
        }
        index++;
    }
}
}

```

Most of the applet in Example 12-3 should be self-explanatory. The `init()` method starts the thread in which we do our computation. `paint()` just displays the four images we create. All the work, including the computation, is done in the thread's `run()` method. `run()` starts by setting up a color map, creating a `MemoryImageSource` with animation enabled and creating four images using that source as the producer. It then does the computation, which I won't explain; for our purposes, the interesting part is what happens when we've computed a pixel. We set the appropriate byte in our data array, `pixels[]`, and then call `newPixels()`, giving the location of the new pixel and its size (1 by 1) as arguments. Thus, we redraw the images for every new pixel. In a real application, you would probably compute a somewhat larger chunk of new data before updating the screen, but the same principles apply.

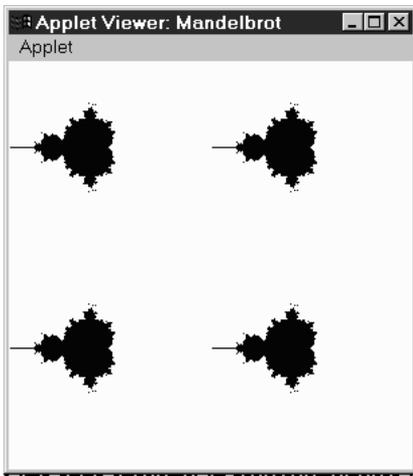


Figure 12-3: Mandelbrot output

12.4 ImageConsumer

The `ImageConsumer` interface specifies the methods that must be implemented to receive data from an `ImageProducer`. For the most part, that is the only context in which you need to know about the `ImageConsumer` interface. If you write an image producer, it will be handed a number of obscure objects, about which you know nothing except that they implement `ImageConsumer`, and that you can therefore call the methods discussed in this section to deliver your data. The chances that you will ever implement an image consumer are rather remote, unless you are porting Java to a new environment. It is more likely that you will want to subclass `ImageFilter`, in which case you may need to implement some of these methods. But most of the time, you will just need to know how to hand your data off to the next element in the chain.

The `java.awt.image` package includes two classes that implement `ImageConsumer`: `PixelGrabber` and `ImageFilter` (and its subclasses). These classes are unique in that they don't display anything on the screen. `PixelGrabber` takes the image data and stores it in a pixel array; you can use this array to save the image in a file, generate a new image, etc. `ImageFilter`, which is used in conjunction with `FilteredImageSource`, modifies the image data; the `FilteredImageSource` sends the modified image to another consumer, which can further modify or display the new image. When you draw an image on the screen, the JDK's `ImageRepresentation` class is probably doing the real work. This class is part of the `sun.awt.image` package. You really don't need to know anything about it, although you may see `ImageRepresentation` mentioned in a stack trace if you try to filter beyond the end of a pixel array.

12.4.1 ImageConsumer Interface

Constants

There are two sets of constants for `ImageConsumer`. One set represents those that can be used for the `imageComplete()` method. The other is used with the `setHints()` method. See the descriptions of those methods on how to use them.

The first set of flags is for the `imageComplete()` method:

public static final int IMAGEABORTED

The `IMAGEABORTED` flag signifies that the image creation process was aborted and the image is not complete. In the image production process, an abort could mean multiple things. It is possible that retrying the production would succeed.

public static final int IMAGEERROR

The `IMAGEERROR` flag signifies that an error was encountered during the image creation process and the image is not complete. In the image production process, an error could mean multiple things. More than likely, the image file or pixel data is invalid, and retrying won't succeed.

public static final int SINGLEFRAMEDONE

The `SINGLEFRAMEDONE` flag signifies that a frame other than the last has completed loading. There are additional frames to display, but a new frame is available and is complete. For an example of this flag in use, see the dynamic `ImageFilter` example in Example 12-8.

public static final int STATICIMAGEDONE

The `STATICIMAGEDONE` flag signifies that the image has completed loading. If this is a multiframe image, all frames have been generated. For an example of this flag in use, see the dynamic `ImageFilter` example in Example 12-8.

The following set of flags can be ORed together to form the single parameter to the `setHints()` method. Certain flags do not make sense set together, but it is the responsibility of the concrete `ImageConsumer` to enforce this.

public static final int COMPLETESCANLINES

The `COMPLETESCANLINES` flag signifies that each call to `setPixels()` will deliver at least one complete scan line of pixels to this consumer.

public static final int RANDOMPIXELORDER

The `RANDOMPIXELORDER` flag tells the consumer that pixels are not provided in any particular order. Therefore, the consumer cannot perform optimization that depends on pixel delivery order. In the absence of both `COMPLETESCANLINES` and `RANDOMPIXELORDER`, the `ImageConsumer` should assume pixels will arrive in `RANDOMPIXELORDER`.

public static final int SINGLEFRAME

The `SINGLEFRAME` flag tells the consumer that this image contains a single non-changing frame. This is the case with most image formats. An example of an image that does not contain a single frame is the multiframe GIF89a image.

public static final int SINGLEPASS

The `SINGLEPASS` flag tells the consumer to expect each pixel once and only once. Certain image formats, like progressive JPEG images, deliver a single image several times, with each pass yielding a sharper image.

public static final int TOPDOWNLEFTRIGHT

The final `setHints()` flag, `TOPDOWNLEFTRIGHT`, tells the consumer to expect the pixels in a top-down, left-right order. This flag will almost always be set.

Methods

The interface methods are presented in the order in which they are normally called by an `ImageProducer`.

void setDimensions (int width, int height)

The `setDimensions()` method should be called once the `ImageProducer` knows the width and height of the image. This is the actual width and height, not necessarily the scaled size. It is the consumer's responsibility to do the scaling and resizing.

void setProperties (Hashtable properties)

The `setProperties()` method should only be called by the `ImageProducer` if the image has any properties that should be stored for later retrieval with the `getProperty()` method of `Image`. Every image format has its own property set. One property that tends to be common is the "comment" property. `properties` represents the `Hashtable` of properties for the image; the name of each property is used as the `Hashtable` key.

void setColorModel (ColorModel model)

The `setColorModel()` method gives the `ImageProducer` the opportunity to tell the `ImageConsumer` that the `ColorModel` `model` will be used for the majority of pixels in the image. The `ImageConsumer` may use this information for optimization. However, each call to `setPixels()` contains its own `ColorModel`, which isn't necessarily the same as the color model given here. In other words, `setColorModel()` is only advisory; it does not guarantee that all (or any) of the pixels in the image will use this model. Using different color models for different parts of an image is possible, but not recommended.

void setHints (int hints)

An `ImageProducer` should call the `setHints()` method prior to any `setPixels()` calls. The hints are formed by ORing the constants `COMPLETESCANLINES`, `RANDOMPIXELORDER`, `SINGLEFRAME`, `SINGLEPASS`, and `TOPDOWNLEFTRIGHT`. These hints give the image consumer information about the order in which the producer will deliver pixels. When the `ImageConsumer` is receiving pixels, it can take advantage of these hints for optimization.

void setPixels (int x, int y, int width, int height, ColorModel model, byte pixels[], int offset, int scansize)

An `ImageProducer` calls the `setPixels()` method to deliver the image pixel data to the `ImageConsumer`. The bytes are delivered a rectangle at a time. (x, y) represents the top left corner of the rectangle; its dimensions are `width` `height`. `model` is the `ColorModel` used for this set of pixels; different calls to `setPixels()` may use different color models. The pixels themselves are taken from the byte array `pixels`. `offset` is the first element of the pixel array that will be used. `scansize` is the length of the scan lines in the array. In most cases, you want the consumer to render all the pixels on the scan line; in this case, `scansize` will equal `width`. However, there are cases in which you want the consumer to ignore part of the scan line; you may be clipping an image, and the ends of the scan line fall outside the clipping region. In this case, rather than copying the pixels you want into a new array, you can specify a `width` that is smaller than `scansize`.

That's a lot of information, but it's easy to summarize. A pixel located at point $(x1, y1)$ within the rectangle being delivered to the consumer is located at position $((y1 - y) * scansize + (x1 - x) + offset)$ within the array `pixels[]`. Figure 12-4 shows how the pixels delivered by `setPixels()` fit into the complete image; Figure 12-5 shows how pixels are stored within the array.

void setPixels (int x, int y, int width, int height, ColorModel model, int pixels[], int offset, int scansize)

The second `setPixels()` method is similar to the first. `pixels[]` is an array of `ints`; this is necessary when you have more than eight bits of data per pixel.

void imageComplete (int status)

The `ImageProducer` calls `imageComplete()` to tell an `ImageConsumer` that it has transferred a complete image. The status argument is a flag that describes exactly why the `ImageProducer` has finished. It may have one of the following values: `IMAGEABORTED` (if the image production was aborted); `IMAGEERROR` (if an error in producing the image occurred); `SINGLEFRAMEDONE` (if a single frame of a multiframe image has been completed); or `STATICIMAGEDONE` (if all pixels have been delivered). When `imageComplete()` gets called, the `ImageConsumer` should call the image producer's `removeConsumer()` method, unless it wants to receive additional frames (status of `SINGLEFRAMEDONE`).

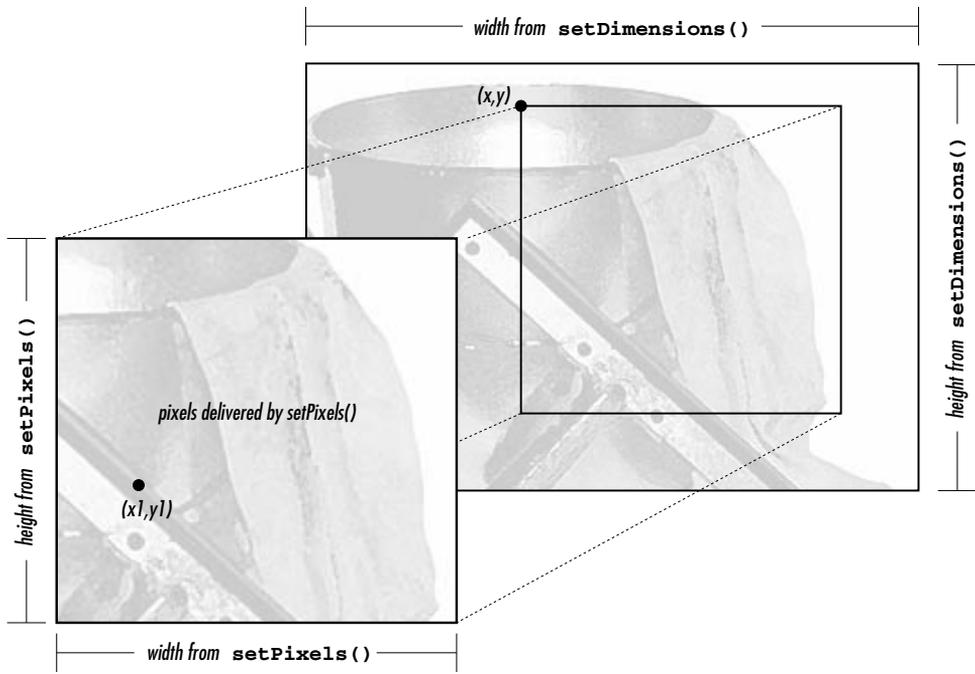


Figure 12-4: Delivering pixels for an image

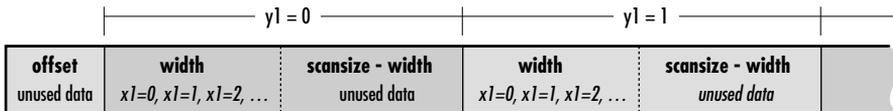


Figure 12-5: Storing pixels in an array

PPMImageDecoder

Now that we have discussed the `ImageConsumer` interface, we're finally ready to give an example of a full-fledged `ImageProducer`. This producer uses the methods of the `ImageConsumer` interface to communicate with image consumers; image consumers use the `ImageProducer` interface to register themselves with this producer.

Our image producer will interpret images in the PPM format.* PPM is a simple image format developed by Jef Poskanzer as part of the *pbmplus* image conversion package. A PPM file starts with a header consisting of the image type, the image's width and height in pixels, and the maximum value of any RGB component. The

* For more information about PPM and the *pbmplus* package, see *Encyclopedia of Graphics File Formats*, by James D. Murray and William VanRyper (from O'Reilly & Associates). See also <http://www.acme.com/>.

header is entirely in ASCII. The pixel data follows the header; it is either in binary (if the image type is P6) or ASCII (if the image type is P3). The pixel data is simply a series of bytes describing the color of each pixel, moving left to right and top to bottom. In binary format, each pixel is represented by three bytes: one for red, one for green, and one for blue. In ASCII format, each pixel is represented by three numeric values, separated by white space (space, tab, or newline). A comment may occur anywhere in the file, but it would be surprising to see one outside of the header. Comments start with # and continue to the end of the line. ASCII format files are obviously much larger than binary files. There is no compression on either file type.

The `PPMImageDecoder` source is listed in Example 12-4. The applet that uses this class is shown in Example 12-5. You can reuse a lot of the code in the `PPMImageDecoder` when you implement your own image producers.

Example 12-4: PPMImageDecoder Source

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

public class PPMImageDecoder implements ImageProducer {

    /* Since done in-memory, only one consumer */
    private ImageConsumer consumer;
    boolean loadError = false;
    int width;
    int height;
    int store[][];
    Hashtable props = new Hashtable();
    /* Format of Ppm file is single pass/frame, w/ complete scan lines in order */
    private static int PpmHints = (ImageConsumer.TOPDOWNLEFTTORIGHT |
                                   ImageConsumer.COMPLETESCANLINES |
                                   ImageConsumer.SINGLEPASS |
                                   ImageConsumer.SINGLEFRAME);
```

The class starts by declaring class variables and constants. We will use the variable `PpmHints` when we call `setHints()`. Here, we set this variable to a collection of “hint” constants that indicate we will produce pixel data in top-down, left-right order; we will always send complete scan lines; we will make only one pass over the pixel data (we will send each pixel once); and there is one frame per image (i.e., we aren’t producing a multiframe sequence).

The next chunk of code implements the `ImageProducer` interface; consumers use it to request image data:

```

/* There is only a single consumer. When it registers, produce image. */
/* On error, notify consumer. */

    public synchronized void addConsumer (ImageConsumer ic) {
        consumer = ic;
        try {
            produce();
        } catch (Exception e) {
            if (consumer != null)
                consumer.imageComplete (ImageConsumer.IMAGEERROR);
        }
        consumer = null;
    }

/* If consumer passed to routine is single consumer, return true, else false. */

    public synchronized boolean isConsumer (ImageConsumer ic) {
        return (ic == consumer);
    }

/* Disables consumer if currently consuming. */

    public synchronized void removeConsumer (ImageConsumer ic) {
        if (consumer == ic)
            consumer = null;
    }

/* Production is done by adding consumer. */

    public void startProduction (ImageConsumer ic) {
        addConsumer (ic);
    }

    public void requestTopDownLeftRightResend (ImageConsumer ic) {
        // Not needed. The data is always in this format.
    }

```

The previous group of methods implements the `ImageProducer` interface. They are quite simple, largely because of the way this `ImageProducer` generates images. It builds the image in memory before delivering it to the consumer; you must call the `readImage()` method (discussed shortly) before you can create an image with this consumer. Because the image is in memory before any consumers can register their interest, we can write an `addConsumer()` method that registers a consumer and delivers all the data to that consumer before returning. Therefore, we don't need to manage a list of consumers in a `Hashtable` or some other collection object. We can store the current consumer in an instance variable `ic` and forget about any others: only one consumer exists at a time. To make sure that only one consumer exists at a time, we synchronize the `addConsumer()`, `isConsumer()`, and `removeConsumer()` methods. Synchronization prevents another consumer from

registering itself before the current consumer has finished. If you write an `ImageProducer` that builds the image in memory before delivering it, you can probably use this code verbatim.

`addConsumer()` is little more than a call to the method `produce()`, which handles “consumer relations”: it delivers the pixels to the consumer using the methods in the `ImageConsumer` interface. If `produce()` throws an exception, `addConsumer()` calls `imageComplete()` with an `IMAGEERROR` status code. Here’s the code for the `produce()` method:

```

/* Production Process:
   Prerequisite: Image already read into store array. (readImage)
                  props / width / height already set (readImage)
   Assumes RGB Color Model - would need to filter to change.
   Sends Ppm Image data to consumer.
   Pixels sent one row at a time.
*/

private void produce () {
    ColorModel cm = ColorModel.getRGBdefault();
    if (consumer != null) {
        if (loadError) {
            consumer.imageComplete (ImageConsumer.IMAGEERROR);
        } else {
            consumer.setDimensions (width, height);
            consumer.setProperties (props);
            consumer.setColorModel (cm);
            consumer.setHints (PpmHints);
            for (int j=0;j<height;j++)
                consumer.setPixels (0, j, width, 1, cm, store[j], 0, width);
            consumer.imageComplete (ImageConsumer.STATICIMAGEDONE);
        }
    }
}

```

`produce()` just calls the `ImageConsumer` methods in order: it sets the image’s dimensions, hands off an empty `Hashtable` of properties, sets the color model (the default RGB model) and the hints, and then calls `setPixels()` once for each row of pixel data. The data is in the integer array `store[][]`, which has already been loaded by the `readImage()` method (defined in the following code). When the data is delivered, the method `setPixels()` calls `imageComplete()` to indicate that the image has been finished successfully.

```

/* Allows reading to be from internal byte array, in addition to disk/socket */

public void readImage (byte b[]) {
    readImage (new ByteArrayInputStream (b));
}

/* readImage reads image data from Stream */
/* parses data for PPM format */

```

```

/* closes inputStream when done          */

public void readImage (InputStream is) {
    long tm = System.currentTimeMillis();
    boolean raw=false;
    DataInputStream dis = null;
    BufferedInputStream bis = null;
    try {
        bis = new BufferedInputStream (is);
        dis = new DataInputStream (bis);
        String word;
        word = readWord (dis);
        if ("P6".equals (word)) {
            raw = true;
        } else if ("P3".equals (word)) {
            raw = false;
        } else {
            throw (new AWTException ("Invalid Format " + word));
        }
        width = Integer.parseInt (readWord (dis));
        height = Integer.parseInt (readWord (dis));
        // Could put comments in props - makes readWord more complex
        int maxColors = Integer.parseInt (readWord (dis));
        if ((maxColors < 0) || (maxColors > 255)) {
            throw (new AWTException ("Invalid Colors " + maxColors));
        }
        store = new int[height][width];
        if (raw) { // binary format (raw) pixel data
            byte row[] = new byte [width*3];
            for (int i=0;i<height;i++){
                dis.readFully (row);
                for (int j=0,k=0;j<width;j++,k+=3) {
                    int red = row[k];
                    int green = row[k+1];
                    int blue = row[k+2];
                    if (red < 0)
                        red +=256;
                    if (green < 0)
                        green +=256;
                    if (blue < 0)
                        blue +=256;
                    store[i][j] = (0xff<< 24) | (red << 16) |
                        (green << 8) | blue;
                }
            }
        }
        else { // ASCII pixel data
            for (int i=0;i<height;i++) {
                for (int j=0;j<width;j++) {
                    int red = Integer.parseInt (readWord (dis));
                    int green = Integer.parseInt (readWord (dis));
                    int blue = Integer.parseInt (readWord (dis));
                    store[i][j] = (0xff<< 24) | (red << 16) |
                        (green << 8) | blue;
                }
            }
        }
    }
}

```

```

        }
    }
} catch (IOException io) {
    loadError = true;
    System.out.println ("IO Exception " + io.getMessage());
} catch (AWTException awt) {
    loadError = true;
    System.out.println ("AWT Exception " + awt.getMessage());
} catch (NoSuchElementException nse) {
    loadError = true;
    System.out.println ("No Such Element Exception " + nse.getMessage());
} finally {
    try {
        if (dis != null)
            dis.close();
        if (bis != null)
            bis.close();
        if (is != null)
            is.close();
    } catch (IOException io) {
        System.out.println ("IO Exception " + io.getMessage());
    }
}
System.out.println ("Done in " + (System.currentTimeMillis() - tm)
    + " ms");
}

```

`readImage()` reads the image data from an `InputStream` and converts it into the array of pixel data that `produce()` transfers to the consumer. Code using this class must call `readImage()` to process the data before calling `createImage()`; we'll see how this works shortly. Although there is a lot of code in `readImage()`, it's fairly simple. (It would be much more complex if we were dealing with an image format that compressed the data.) It makes heavy use of `readWord()`, a utility method that we'll discuss next; `readWord()` returns a word of ASCII text as a string.

`readImage()` starts by converting the `InputStream` into a `DataInputStream`. It uses `readWord()` to get the first word from the stream. This should be either "P6" or "P3", depending on whether the data is in binary or ASCII. It then uses `readWord()` to save the image's width and height and the maximum value of any color component. Next, it reads the color data into the `store[][]` array. The ASCII case is simple because we can use `readWord()` to read ASCII words conveniently; we read red, green, and blue words, convert them into ints, and pack the three into one element (one pixel) of `store[][]`. For binary data, we read an entire scan line into the byte array `row[]`, using `readFully()`; then we start a loop that packs this scan line into one row of `store[][]`. A little additional complexity is in the inner loop because we must keep track of two arrays (`row[]` and `store[][]`). We read red, green, and blue components from `row[]`, converting Java's signed bytes to unsigned data by adding 256 to any negative values; finally, we pack these components into one element of `store[][]`.

```

/* readWord returns a word of text from stream      */
/* Ignores PPM comment lines.                      */
/* word defined to be something wrapped by whitespace */

private String readWord (InputStream is) throws IOException {
    StringBuffer buf = new StringBuffer();
    int b;
    do { // get rid of leading whitespace
        if ((b=is.read()) == -1)
            throw new EOFException();
        if ((char)b == '#') { // read to end of line - ppm comment
            DataInputStream dis = new DataInputStream (is);
            dis.readLine();
            b = ' '; // ensure more reading
        }
    }while (Character.isSpace ((char)b));
    do {
        buf.append ((char) (b));
        if ((b=is.read()) == -1)
            throw new EOFException();
    } while (!Character.isSpace ((char)b)); // reads first space
    return buf.toString();
}
}

```

`readWord()` is a utility method that reads one ASCII word from an `InputStream`. A word is a sequence of characters that aren't spaces; space characters include newlines and tabs in addition to spaces. This method also throws out any comments (anything between # and the end of the line). It collects the characters into a `StringBuffer`, converting the `StringBuffer` into a `String` when it returns.

Example 12-5: PPMImageDecoder Test Program

```

import java.awt.Graphics;
import java.awt.Color;
import java.awt.image.ImageConsumer;
import java.awt.Image;
import java.awt.MediaTracker;
import java.net.URL;
import java.net.MalformedURLException;
import java.io.InputStream;
import java.io.IOException;
import java.applet.Applet;
public class ppmViewer extends Applet {
    Image image = null;
    public void init () {
        try {
            String file = getParameter ("file");
            if (file != null) {
                URL imageurl = new URL (getDocumentBase(), file);
                InputStream is = imageurl.openStream();
                PPMImageDecoder ppm = new PPMImageDecoder ();
                ppm.readImage (is);
            }
        }
    }
}

```

Example 12–5: PPMImageDecoder Test Program (continued)

```

        image = createImage (ppm);
        repaint();
    }
} catch (MalformedURLException me) {
    System.out.println ("Bad URL");
} catch (IOException io) {
    System.out.println ("Bad File");
}
}
public void paint (Graphics g) {
    g.drawImage (image, 0, 0, this);
}
}

```

The applet we use to test our `ImageProducer` is very simple. It creates a URL that points to an appropriate PPM file and gets an `InputStream` from that URL. It then creates an instance of our `PPMImageDecoder`; calls `readImage()` to load the image and generate pixel data; and finally, calls `createImage()` with our `ImageProducer` as an argument to create an `Image` object, which we draw in `paint()`.

12.4.2 *PixelGrabber*

The `PixelGrabber` class is a utility for converting an image into an array of pixels. This is useful in many situations. If you are writing a drawing utility that lets users create their own graphics, you probably want some way to save a drawing to a file. Likewise, if you're implementing a shared whiteboard, you'll want some way to transmit images across the Net. If you're doing some kind of image processing, you may want to read and alter individual pixels in an image. The `PixelGrabber` class is an `ImageConsumer` that can capture a subset of the current pixels of an `Image`. Once you have the pixels, you can easily save the image in a file, send it across the Net, or work with individual points in the array. To recreate the `Image` (or a modified version), you can pass the pixel array to a `MemoryImageSource`.

Prior to Java 1.1, `PixelGrabber` saves an array of pixels but doesn't save the image's width and height—that's your responsibility. You may want to put the width and height in the first two elements of the pixel array and use an offset of 2 when you store (or reproduce) the image.

Starting with Java 1.1, the grabbing process changes in several ways. You can ask the `PixelGrabber` for the image's size or color model. You can grab pixels asynchronously and abort the grabbing process before it is completed. Finally, you don't have to preallocate the pixel data array.

Constructors

public PixelGrabber (ImageProducer ip, int x, int y, int width, int height, int pixels[], int offset, int scansize)

The first `PixelGrabber` constructor creates a new `PixelGrabber` instance. The `PixelGrabber` uses `ImageProducer ip` to store the unscaled cropped rectangle at position (x, y) of size `width height` into the `pixels` array, starting at `offset` within `pixels`, and each row starting at increments of `scansize` from that.

As shown in Figure 12-5, the position $(x1, y1)$ would be stored in `pixels[]` at position $(y1 - y) * scansize + (x1 - x) + offset$. Calling `grabPixels()` starts the process of writing pixels into the array.

The `ColorModel` for the pixels copied into the array is always the default RGB model: that is, 32 bits per pixel, with 8 bits for alpha, red, green, and blue components.

public PixelGrabber (Image image, int x, int y, int width, int height, int pixels[], int offset, int scansize)

This version of the `PixelGrabber` constructor gets the `ImageProducer` of the `Image image` through `getSource()`; it then calls the previous constructor to create the `PixelGrabber`.

public PixelGrabber (Image image, int x, int y, int width, int height, boolean forceRGB) ★

This version of the constructor does not require you to preallocate the pixel array and lets you preserve the color model of the original image. If `forceRGB` is `true`, the pixels of `image` are converted to the default RGB model when grabbed. If `forceRGB` is `false` and all the pixels of `image` use one `ColorModel`, the original color model of `image` is preserved.

As with the other constructors, the `x`, `y`, `width`, and `height` values define the bounding box to grab. However, there's one special case to consider. Setting `width` or `height` to `-1` tells the `PixelGrabber` to take the `width` and `height` from the image itself. In this case, the grabber stores all the pixels below and to the right of the point (x, y) . If (x, y) is outside of the image, you get an empty array.

Once the pixels have been grabbed, you get the pixel data via the `getPixels()` method described in "Other methods." To get the `ColorModel`, see the `getColorModel()` method.

ImageConsumer interface methods

public void setDimensions (int width, int height)

In Java 1.0, the `setDimensions()` method of `PixelGrabber` ignores the `width` and `height`, since this was set by the constructor.

With Java 1.1, `setDimensions()` is called by the image producer to give it the dimensions of the original image. This is how the `PixelGrabber` finds out the image's size if the constructor specified -1 for the image's width or height.

public void setHints (int hints)

The `setHints()` method ignores the hints.

public void setProperties (Hashtable properties)

The `setProperties()` method ignores the properties.

public void setColorModel (ColorModel model)

The `setColorModel()` method ignores the model.

public void setPixels (int x, int y, int w, int h, ColorModel model, byte pixels[], int offset, int scansize)

The `setPixels()` method is called by the `ImageProducer` to deliver pixel data for some image. If the pixels fall within the portion of the image that the `PixelGrabber` is interested in, they are stored within the array passed to the `PixelGrabber` constructor. If necessary, the `ColorModel` is used to convert each pixel from its original representation to the default RGB representation. This method is called when each pixel coming from the image producer is represented by a byte.

public void setPixels (int x, int y, int w, int h, ColorModel model, int pixels[], int offset, int scansize)

The second `setPixels()` method is almost identical to the first; it is used when each pixel coming from the image producer is represented by an `int`.

public synchronized void imageComplete (int status)

The `imageComplete()` method uses `status` to determine if the pixels were successfully delivered. The `PixelGrabber` then notifies anyone waiting for the pixels from a `grabPixels()` call.

Grabbing methods

public synchronized boolean grabPixels (long ms) throws InterruptedException

The `grabPixels()` method starts storing pixel data from the image. It doesn't return until all pixels have been loaded into the pixels array or until `ms` milliseconds have passed. The return value is `true` if all pixels were successfully acquired. Otherwise, it returns `false` for the abort, error, or timeout condition encountered. The exception `InterruptedException` is thrown if another thread interrupts this one while waiting for pixel data.

public boolean grabPixels () throws InterruptedException

This `grabPixels()` method starts storing pixel data from the image. It doesn't return until all pixels have been loaded into the `pixels` array. The return value is `true` if all pixels were successfully acquired. It returns `false` if it encountered an abort or error condition. The exception `InterruptedException` is thrown if another thread interrupts this one while waiting for pixel data.

public synchronized void startGrabbing() ★

The `startGrabbing()` method provides an asynchronous means of grabbing the pixels. This method returns immediately; it does not block like the `grabPixels()` methods described previously. To find out when the `PixelGrabber` has finished, call `getStatus()`.

public synchronized void abortGrabbing() ★

The `abortGrabbing()` method allows you to stop grabbing pixel data from the image. If a thread is waiting for pixel data from a `grabPixels()` call, it is interrupted and `grabPixels()` throws an `InterruptedException`.

Other methods

public synchronized int getStatus() ★

public synchronized int status () ☆

Call the `getStatus()` method to find out whether a `PixelGrabber` succeeded in grabbing the pixels you want. The return value is a set of `ImageObserver` flags ORed together. `ALLBITS` and `FRAMEBITS` indicate success; which of the two you get depends on how the image was created. `ABORT` and `ERROR` indicate that problems occurred while the image was being produced.

`status()` is the Java 1.0 name for this method.

public synchronized int getWidth() ★

The `getWidth()` method reports the width of the image data stored in the destination buffer. If you set `width` to `-1` when you called the `PixelGrabber` constructor, this information will be available only after the grabber has received the information from the image producer (`setDimensions()`). If the width is not available yet, `getWidth()` returns `-1`.

The width of the resulting image depends on several factors. If you specified the width explicitly in the constructor, the resulting image has that width, no questions asked—even if the position at which you start grabbing is outside the image. If you specified `-1` for the width, the resulting width will be the difference between the `x` position at which you start grabbing (set in the constructor) and the actual image width; for example, if you start grabbing at `x=50` and the original image width is `100`, the width of the resulting image is `50`. If `x` falls outside the image, the resulting width is `0`.

public synchronized int getHeight() ★

The `getHeight()` method reports the height of the image data stored in the destination buffer. If you set height to -1 when you called the `PixelGrabber` constructor, this information will be available only after the grabber has received the information from the image producer (`setDimensions()`). If the height is not available yet, `getHeight()` returns -1.

The height of the resulting image depends on several factors. If you specified the height explicitly in the constructor, the resulting image has that height, no questions asked—even if the position at which you start grabbing is outside the image. If you specified -1 for the height, the resulting height will be the difference between the `y` position at which you start grabbing (set in the constructor) and the actual image height; for example, if you start grabbing at `y=50` and the original image height is 100, the height of the resulting image is 50. If `y` falls outside the image, the resulting height is 0.

public synchronized Object getPixels() ★

The `getPixels()` method returns an array of pixel data. If you passed a pixel array to the constructor, you get back your original array object, with the data filled in. If, however, the array was not previously allocated, you get back a new array. The size of this array depends on the image you are grabbing and the portion of that image you want. If size and image format are not known yet, this method returns `null`. If the `PixelGrabber` is still grabbing pixels, this method returns an array that may change based upon the rest of the image. The type of the array you get is either `int[]` or `byte[]`, depending on the color model of the image. To find out if the `PixelGrabber` has finished, call `getStatus()`.

public synchronized ColorModel getColorModel() ★

The `getColorModel()` method returns the color model of the image. This could be the default `RGB ColorModel` if a pixel buffer was explicitly provided, `null` if the color model is not known yet, or a varying color model until all the pixel data has been grabbed. After all the pixels have been grabbed, `getColorModel()` returns the actual color model used for the `getPixels()` array. It is best to wait until grabbing has finished before you ask for the `ColorModel`; to find out, call `getStatus()`.

Using PixelGrabber to modify an image

You can modify images by combining a `PixelGrabber` with `MemoryImageSource`. Use `getImage()` to load an image from the Net; then use `PixelGrabber` to convert the image into an array. Modify the data in the array any way you please; then use `MemoryImageSource` as an image producer to display the new image.

Example 12-6 demonstrates the use of the `PixelGrabber` and `MemoryImageSource` to rotate, flip, and mirror an image. (We could also do the rotations with a subclass of `ImageFilter`, which we will discuss next.) The output is shown in Figure 12-6. When working with an image that is loaded from a local disk or the network, remember to wait until the image is loaded before grabbing its pixels. In this example, we use a `MediaTracker` to wait for the image to load.

Example 12-6: Flip Source

```
import java.applet.*;
import java.awt.*;
import java.awt.image.*;
public class flip extends Applet {
    Image i, j, k, l;
    public void init () {
        MediaTracker mt = new MediaTracker (this);
        i = getImage (getDocumentBase(), "ora-icon.gif");
        mt.addImage (i, 0);

        try {
            mt.waitForAll();
            int width = i.getWidth(this);
            int height = i.getHeight(this);
            int pixels[] = new int [width * height];
            PixelGrabber pg = new PixelGrabber
                (i, 0, 0, width, height, pixels, 0, width);
            if (pg.grabPixels() && ((pg.status() &
                ImageObserver.ALLBITS) !=0)) {
                j = createImage (new MemoryImageSource (width, height,
                    rowFlipPixels (pixels, width, height), 0, width));
                k = createImage (new MemoryImageSource (width, height,
                    colFlipPixels (pixels, width, height), 0, width));
                l = createImage (new MemoryImageSource (height, width,
                    rot90Pixels (pixels, width, height), 0, height));
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

The `try` block in Example 12-6 does all the interesting work. It uses a `PixelGrabber` to grab the entire image into the array `pixels[]`. After calling `grabPixels()`, it checks the `PixelGrabber` status to make sure that the image was stored correctly. It then generates three new images based on the first by calling `createImage()` with a `MemoryImageSource` object as an argument. Instead of using the original array, the `MemoryImageSource` objects call several utility methods to manipulate the array: `rowFlipPixels()`, `colFlipPixels()`, and `rot90Pixels()`. These methods all return integer arrays.

```
public void paint (Graphics g) {
    g.drawImage (i, 10, 10, this); // regular
    if (j != null)
```

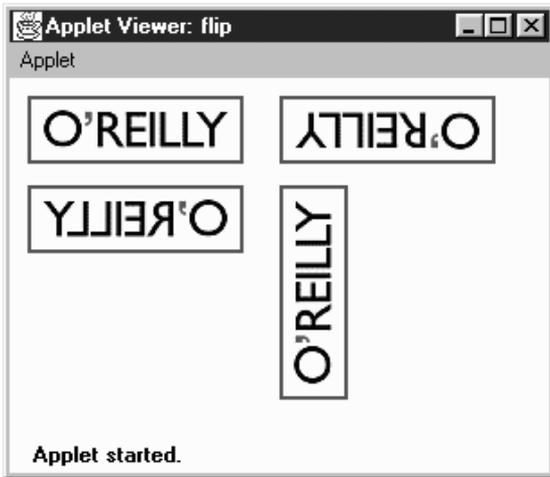


Figure 12-6: Flip output

```

        g.drawImage (j, 150, 10, this); // rowFlip
    if (k != null)
        g.drawImage (k, 10, 60, this); // colFlip
    if (l != null)
        g.drawImage (l, 150, 60, this); // rot90
    }
    private int[] rowFlipPixels (int pixels[], int width, int height) {
        int newPixels[] = null;
        if ((width*height) == pixels.length) {
            newPixels = new int [width*height];
            int newIndex=0;
            for (int y=height-1;y>=0;y--)
                for (int x=width-1;x>=0;x--)
                    newPixels[newIndex++]=pixels[y*width+x];
        }
        return newPixels;
    }
}

```

`rowFlipPixels()` creates a mirror image of the original, flipped horizontally. It is nothing more than a nested loop that copies the original array into a new array.

```

    private int[] colFlipPixels (int pixels[], int width, int height) {
        ...
    }
    private int[] rot90Pixels (int pixels[], int width, int height) {
        ...
    }
}

```

`colFlipPixels()` and `rot90Pixels()` are fundamentally similar to

`rowFlipPixels()`; they just copy the original pixel array into another array, and return the result. `colFlipPixels()` generates a vertical mirror image; `rot90Pixels()` rotates the image by 90 degrees counterclockwise.

Grabbing data asynchronously

To demonstrate the new methods introduced by Java 1.1 for `PixelGrabber`, the following program grabs the pixels and reports information about the original image on mouse clicks. It takes its data from the image used in Figure 12-6.

```
// Java 1.1 only
import java.applet.*;
import java.awt.*;
import java.awt.image.*;
import java.awt.event.*;

public class grab extends Applet {
    Image i;
    PixelGrabber pg;
    public void init () {
        i = getImage (getDocumentBase(), "ora-icon.gif");
        pg = new PixelGrabber (i, 0, 0, -1, -1, false);
        pg.startGrabbing();
        enableEvents (AWTEvent.MOUSE_EVENT_MASK);
    }
    public void paint (Graphics g) {
        g.drawImage (i, 10, 10, this);
    }
    protected void processMouseEvent(MouseEvent e) {
        if (e.getID() == MouseEvent.MOUSE_CLICKED) {
            System.out.println ("Status: " + pg.getStatus());
            System.out.println ("Width: " + pg.getWidth());
            System.out.println ("Height: " + pg.getHeight());
            System.out.println ("Pixels: " +
                (pg.getPixels() instanceof byte[] ? "bytes" : "ints"));
            System.out.println ("Model: " + pg.getColorModel());
        }
        super.processMouseEvent (e);
    }
}
```

This applet creates a `PixelGrabber` without specifying an array, then starts grabbing pixels. The grabber allocates its own array, but we never bother to ask for it since we don't do anything with the data itself: we only report the grabber's status. (If we wanted the data, we'd call `getPixels()`.) Sample output from a single mouse click, after the image loaded, would appear something like the following:

```
Status: 27
Width: 120
Height: 38
Pixels: bytes
Model: java.awt.image.IndexColorModel@1ed34
```

You need to convert the status value manually to the corresponding meaning by looking up the status codes in `ImageObserver`. The value 27 indicates that the 1, 2, 8, and 16 flags are set, which translates to the `WIDTH`, `HEIGHT`, `SOMEBITS`, and `FRAMEBITS` flags, respectively.

12.5 *ImageFilter*

Image filters provide another way to modify images. An `ImageFilter` is used in conjunction with a `FilteredImageSource` object. The `ImageFilter`, which implements `ImageConsumer` (and `Cloneable`), receives data from an `ImageProducer` and modifies it; the `FilteredImageSource`, which implements `ImageProducer`, sends the modified data to the new consumer. As Figure 12-1 shows, an image filter sits between the original `ImageProducer` and the ultimate `ImageConsumer`.

The `ImageFilter` class implements a “null” filter that does nothing to the image. To modify an image, you must use a subclass of `ImageFilter`, by either writing one yourself or using a subclass provided with AWT, like the `CropImageFilter`. Another `ImageFilter` subclass provided with AWT is the `RGBImageFilter`; it is useful for filtering an image on the basis of a pixel’s color. Unlike the `CropImageFilter`, `RGBImageFilter` is an abstract class, so you need to create your own subclass to use it. Java 1.1 introduces two more image filters, `AreaAveragingScaleFilter` and `ReplicateScaleFilter`. Other filters must be created by subclassing `ImageFilter` and providing the necessary methods to modify the image as necessary.

`ImageFilters` tend to work on a pixel-by-pixel basis, so large `Image` objects can take a considerable amount of time to filter, depending on the complexity of the filtering algorithm. In the simplest case, filters generate new pixels based upon the color value and location of the original pixel. Such filters can start delivering data before they have loaded the entire image. More complex filters may use internal buffers to store an intermediate copy of the image so the filter can use adjacent pixel values to smooth or blend pixels together. These filters may need to load the entire image before they can deliver any data to the ultimate consumer.

To use an `ImageFilter`, you pass it to the `FilteredImageSource` constructor, which serves as an `ImageProducer` to pass the new pixels to their consumer. The following code runs the image `logo.jpg` through an image filter, `SomeImageFilter`, to produce a new image. The constructor for `SomeImageFilter` is called within the constructor for `FilteredImageSource`, which in turn is the only argument to `createImage()`.

```
Image image = getImage (new URL (
    "http://www.ora.com/images/logo.jpg"));
Image newOne = createImage (new FilteredImageSource (image.getSource(),
    new SomeImageFilter()));
```

12.5.1 *ImageFilter* Methods

Variables

protected ImageConsumer consumer;

The actual `ImageConsumer` for the image. It is initialized automatically for you by the `getFilterInstance()` method.

Constructor

public ImageFilter ()

The only constructor for `ImageFilter` is the default one, which takes no arguments. Subclasses can provide their own constructors if they need additional information.

ImageConsumer interface methods

public void setDimensions (int width, int height)

The `setDimensions()` method of `ImageFilter` is called when the width and height of the original image are known. It calls `consumer.setDimensions()` to tell the next consumer the dimensions of the filtered image. If you subclass `ImageFilter` and your filter changes the image's dimensions, you should override this method to compute and report the new dimensions.

public void setProperties (Hashtable properties)

The `setProperties()` method is called to provide the image filter with the property list for the original image. The image filter adds the property filters to the list and passes it along to the next consumer. The value given for the filters property is the result of the image filter's `toString()` method; that is, the `String` representation of the current filter. If `filters` is already set, information about this `ImageFilter` is appended to the end. Subclasses of `ImageFilter` may add other properties.

public void setColorModel (ColorModel model)

The `setColorModel()` method is called to give the `ImageFilter` the color model used for most of the pixels in the original image. It passes this color model on to the next consumer. Subclasses may override this method if they change the color model.

public void setHints (int hints)

The `setHints()` method is called to give the `ImageFilter` hints about how the producer will deliver pixels. This method passes the same set of hints to the next consumer. Subclasses must override this method if they need to provide different hints; for example, if they are delivering pixels in a different order.

```
public void setPixels (int x, int y, int width, int height, ColorModel model, byte pixels[],
int offset, int scansize)
```

```
public void setPixels (int x, int y, int width, int height, ColorModel model, int pixels[],
int offset, int scansize)
```

The `setPixels()` method receives pixel data from the `ImageProducer` and passes all the information on to the `ImageConsumer`. `(x, y)` is the top left corner of the bounding rectangle for the pixels. The bounding rectangle has size `width height`. The `ColorModel` for the new image is `model`. `pixels` is the byte or integer array of the pixel information, starting at `offset` (usually 0), with scan lines of size `scansize` (usually `width`).

```
public void imageComplete (int status)
```

The `imageComplete()` method receives the completion status from the `ImageProducer` and passes it along to the `ImageConsumer`.

If you subclass `ImageFilter`, you will probably override the `setPixels()` methods. For simple filters, you may be able to modify the pixel array and deliver the result to `consumer.setPixels()` immediately. For more complex filters, you will have to build a buffer containing the entire image; in this case, the call to `imageComplete()` will probably trigger filtering and pixel delivery.

Cloneable interface methods

```
public Object clone ()
```

The `clone()` method creates a clone of the `ImageFilter`. The `getFilterInstance()` function uses this method to create a copy of the `ImageFilter`. Cloning allows the same filter instance to be used with multiple `Image` objects.

Other methods

```
public ImageFilter getFilterInstance (ImageConsumer ic)
```

`FilteredImageSource` calls `getFilterInstance()` to register `ic` as the `ImageConsumer` for an instance of this filter; to do so, it sets the instance variable `consumer`. In effect, this method inserts the `ImageFilter` between the image's producer and the consumer. You have to override this method only if there are special requirements for the insertion process. This default implementation just calls `clone()`.

```
public void resendTopDownLeftRight (ImageProducer ip)
```

The `resendTopDownLeftRight()` method tells the `ImageProducer` `ip` to try to resend the image data in the top-down, left-to-right order. If you override this method and your `ImageFilter` has saved the image data internally, you may want your `ImageFilter` to resend the data itself, rather than asking the `ImageProducer`. Otherwise, your subclass may ignore the request or pass it along to the `ImageProducer` `ip`.

Subclassing ImageFilter: A blurring filter

When you subclass `ImageFilter`, there are very few restrictions on what you can do. We will create a few subclasses that show some of the possibilities. This `ImageFilter` generates a new pixel by averaging the pixels around it. The result is a blurred version of the original. To implement this filter, we have to save all the pixel data into a buffer; we can't start delivering pixels until the entire image is in hand. Therefore, we override `setPixels()` to build the buffer; we override `imageComplete()` to produce the new pixels and deliver them.

Before looking at the code, here are a few hints about how the filter works; it uses a few tricks that may be helpful in other situations. We need to provide two versions of `setPixels()`: one for integer arrays, and the other for byte arrays. To avoid duplicating code, both versions call a single method, `setThePixels()`, which takes an `Object` as an argument, instead of a pixel array; thus it can be called with either kind of pixel array. Within the method, we check whether the pixels argument is an instance of `byte[]` or `int[]`. The body of this method uses another trick: when it reads the `byte[]` version of the pixel array, it ANDs the value with `0xff`. This prevents the byte value, which is signed, from being converted to a negative `int` when used as an argument to `cm.getRGB()`.

The logic inside of `imageComplete()` gets a bit hairy. This method does the actual filtering, after all the data has arrived. Its job is basically simple: compute an average value of the pixel and the eight pixels surrounding it (i.e., a 3x3 rectangle with the current pixel in the center). The problem lies in taking care of the edge conditions. We don't always want to average nine pixels; in fact, we may want to average as few as four. The `if` statements figure out which surrounding pixels should be included in the average. The pixels we care about are placed in `sumArray[]`, which has nine elements. We keep track of the number of elements that have been saved in the variable `sumIndex` and use a helper method, `avgPixels()`, to compute the average. The code might be a little cleaner if we used a `Vector`, which automatically counts the number of elements it contains, but it would probably be much slower.

Example 12-7 shows the code for the blurring filter.

Example 12-7: Blur Filter Source

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

public class BlurFilter extends ImageFilter {
    private int savedWidth, savedHeight, savedPixels[];
    private static ColorModel defaultCM = ColorModel.getRGBdefault();

    public void setDimensions (int width, int height) {
        savedWidth=width;
```

Example 12–7: Blur Filter Source (continued)

```

        savedHeight=height;
        savedPixels=new int [width*height];
        consumer.setDimensions (width, height);
    }

```

We override `setDimensions()` to save the original image's height and width, which we use later.

```

    public void setColorModel (ColorModel model) {
        // Change color model to model you are generating
        consumer.setColorModel (defaultCM);
    }

    public void setHints (int hintflags) {
        // Set new hints, but preserve SINGLEFRAME setting
        consumer.setHints (TOPDOWNLEFTRIGHT | COMPLETESCANLINES |
            SINGLEPASS | (hintflags & SINGLEFRAME));
    }

```

This filter always generates pixels in the same order, so it sends the hint flags `TOPDOWNLEFTRIGHT`, `COMPLETESCANLINES`, and `SINGLEPASS` to the consumer, regardless of what the image producer says. It sends the `SINGLEFRAME` hint only if the producer has sent it.

```

    private void setThePixels (int x, int y, int width, int height,
        ColorModel cm, Object pixels, int offset, int scansize) {
        int sourceOffset = offset;
        int destinationOffset = y * savedWidth + x;
        boolean bytearray = (pixels instanceof byte[]);
        for (int yy=0;yy<height;yy++) {
            for (int xx=0;xx<width;xx++)
                if (bytearray)
                    savedPixels[destinationOffset++]=
                        cm.getRGB(((byte[])pixels)[sourceOffset++]&0xff);
                else
                    savedPixels[destinationOffset++]=
                        cm.getRGB(((int[])pixels)[sourceOffset++]);
            sourceOffset += (scansize - width);
            destinationOffset += (savedWidth - width);
        }
    }

```

`setThePixels()` saves the pixel data for the image in the array `savedPixels[]`. Both versions of `setPixels()` call this method. It doesn't pass the pixels along to the image consumer, since this filter can't process the pixels until the entire image is available.

```

    public void setPixels (int x, int y, int width, int height,
        ColorModel cm, byte pixels[], int offset, int scansize) {
        setThePixels (x, y, width, height, cm, pixels, offset, scansize);
    }

```

```

public void setPixels (int x, int y, int width, int height,
    ColorModel cm, int pixels[], int offset, int scansize) {
    setThePixels (x, y, width, height, cm, pixels, offset, scansize);
}

public void imageComplete (int status) {
    if ((status == IMAGEABORTED) || (status == IMAGEERROR)) {
        consumer.imageComplete (status);
        return;
    } else {
        int pixels[] = new int [savedWidth];
        int position, sumArray[], sumIndex;
        sumArray = new int [9]; // maxsize - vs. Vector for performance
        for (int yy=0;yy<savedHeight;yy++) {
            position=0;
            int start = yy * savedWidth;
            for (int xx=0;xx<savedWidth;xx++) {
                sumIndex=0;
                // xx yy
                sumArray[sumIndex++] = savedPixels[start+xx]; // center center
                if (yy != (savedHeight-1)) // center bottom
                    sumArray[sumIndex++] = savedPixels[start+xx+savedWidth];
                if (yy != 0) // center top
                    sumArray[sumIndex++] = savedPixels[start+xx-savedWidth];
                if (xx != (savedWidth-1)) // right center
                    sumArray[sumIndex++] = savedPixels[start+xx+1];
                if (xx != 0) // left center
                    sumArray[sumIndex++] = savedPixels[start+xx-1];
                if ((yy != 0) && (xx != 0)) // left top
                    sumArray[sumIndex++] = savedPixels[start+xx-savedWidth-1];
                if ((yy != (savedHeight-1)) && (xx != (savedWidth-1)))
                    // right bottom
                    sumArray[sumIndex++] = savedPixels[start+xx+savedWidth+1];
                if ((yy != 0) && (xx != (savedWidth-1))) //right top
                    sumArray[sumIndex++] = savedPixels[start+xx-savedWidth+1];
                if ((yy != (savedHeight-1)) && (xx != 0)) //left bottom
                    sumArray[sumIndex++] = savedPixels[start+xx+savedWidth-1];
                pixels[position++] = avgPixels(sumArray, sumIndex);
            }
            consumer.setPixels (0, yy, savedWidth, 1, defaultCM,
                pixels, 0, savedWidth);
        }
        consumer.imageComplete (status);
    }
}

```

`imageComplete()` does the actual filtering after the pixels have been delivered and saved. If the producer reports that an error occurred, this method passes the error flags to the consumer and returns. If not, it builds a new array, `pixels[]`, which contains the filtered pixels, and delivers these to the consumer.

Previously, we gave an overview of how the filtering process works. Here are some details. `(xx, yy)` represents the current point's x and y coordinates. The point `(xx, yy)` must always fall within the image; otherwise, our loops are constructed incorrectly. Therefore, we can copy `(xx, yy)` into the `sumArray[]` for averaging without any tests. For the point's eight neighbors, we check whether the neighbor falls in the image; if so, we add it to `sumArray[]`. For example, the point just below `(xx, yy)` is at the bottom center of the 33 rectangle of points we are averaging. We know that `xx` falls within the image; `yy` falls within the image if it doesn't equal `savedHeight-1`. We do similar tests for the other points.

Even though we're working with a rectangular image, our arrays are all one-dimensional so we have to convert a coordinate pair `(xx, yy)` into a single array index. To help us do the bookkeeping, we use the local variable `start` to keep track of the start of the current scan line. Then `start + xx` is the current point; `start + xx + savedWidth` is the point immediately below; `start + xx + savedWidth-1` is the point below and to the left; and so on.

`avgPixels()` is our helper method for computing the average value that we assign to the new pixel. For each pixel in the `pixels[]` array, it extracts the red, blue, green, and alpha components; averages them separately, and returns a new ARGB value.

```
private int avgPixels (int pixels[], int size) {
    float redSum=0, greenSum=0, blueSum=0, alphaSum=0;
    for (int i=0;i<size;i++)
        try {
            int pixel = pixels[i];
            redSum += defaultCM.getRed (pixel);
            greenSum += defaultCM.getGreen (pixel);
            blueSum += defaultCM.getBlue (pixel);
            alphaSum += defaultCM.getAlpha (pixel);
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println ("Ooops");
        }
    int redAvg = (int)(redSum / size);
    int greenAvg = (int)(greenSum / size);
    int blueAvg = (int)(blueSum / size);
    int alphaAvg = (int)(alphaSum / size);
    return ((0xff << 24) | (redAvg << 16) |
            (greenAvg << 8) | (blueAvg << 0));
}
```

Producing many images from one: dynamic ImageFilter

The `ImageFilter` framework is flexible enough to allow you to return a sequence of images based on an original. You can send back one frame at a time, calling the following when you are finished with each frame:

```
consumer.imageComplete(ImageConsumer.SINGLEFRAMEDONE);
```

After you have generated all the frames, you can tell the consumer that the sequence is finished with the `STATICIMAGEDONE` constant. In fact, this is exactly what the new animation capabilities of `MemoryImageSource` use.

In Example 12-8, the `DynamicFilter` lets the consumer display an image. After the image has been displayed, the filter gradually overwrites the image with a specified color by sending additional image frames. The end result is a solid colored rectangle. Not too exciting, but it's easy to imagine interesting extensions: you could use this technique to implement a fade from one image into another. The key points to understand are:

- This filter does not override `setPixels()`, so it is extremely fast. In this case, we want the original image to reach the consumer, and there is no reason to save the image in a buffer.
- Filtering takes place in the image-fetching thread, so it is safe to put the filter-processing thread to sleep if the image is coming from disk. If the image is in memory, filtering should not sleep because there will be a noticeable performance lag in your program if it does. The `DynamicFilter` class has a delay parameter to its constructor that lets you control this behavior.
- This subclass overrides `setDimensions()` to save the image's dimensions for its own use. It needs to override `setHints()` because it sends pixels to the consumer in a nonstandard order: it sends the original image, then goes back and starts sending overlays. Likewise, this subclass overrides `resendTopDownLeft-Right()` to do nothing because there is no way the original `ImageProducer` can replace all the changes with the original `Image`.
- `imageComplete()` is where all the fun happens. Take a special look at the status flags that are returned.

Example 12-8: DynamicFilter Source

```
import java.awt.*;
import java.awt.image.*;
public class DynamicFilter extends ImageFilter {
    Color overlapColor;
    int delay;
    int imageWidth;
    int imageHeight;
    int iterations;
    DynamicFilter (int delay, int iterations, Color color) {
        this.delay = delay;
        this.iterations = iterations;
        overlapColor = color;
    }
    public void setDimensions (int width, int height) {
        imageWidth = width;
```

Example 12–8: DynamicFilter Source (continued)

```

        imageHeight = height;
        consumer.setDimensions (width, height);
    }
    public void setHints (int hints) {
        consumer.setHints (ImageConsumer.RANDOMPIXELORDER);
    }
    public void resendTopDownLeftRight (ImageProducer ip) {
    }
    public void imageComplete (int status) {
        if ((status == IMAGEERROR) || (status == IMAGEABORTED)) {
            consumer.imageComplete (status);
            return;
        } else {
            int xWidth = imageWidth / iterations;
            if (xWidth <= 0)
                xWidth = 1;
            int newPixels[] = new int [xWidth*imageHeight];
            int iColor = overlapColor.getRGB();
            for (int x=0;x<(xWidth*imageHeight);x++)
                newPixels[x] = iColor;
            int t=0;
            for (;t<(imageWidth-xWidth);t+=xWidth) {
                consumer.setPixels(t, 0, xWidth, imageHeight,
                    ColorModel.getRGBdefault(), newPixels, 0, xWidth);
                consumer.imageComplete (ImageConsumer.SINGLEFRAMEDONE);
                try {
                    Thread.sleep (delay);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            int left = imageWidth-t;
            if (left > 0) {
                consumer.setPixels(imageWidth-left, 0, left, imageHeight,
                    ColorModel.getRGBdefault(), newPixels, 0, xWidth);
                consumer.imageComplete (ImageConsumer.SINGLEFRAMEDONE);
            }
            consumer.imageComplete (STATICIMAGEDONE);
        }
    }
}

```

The `DynamicFilter` relies on the default `setPixels()` method to send the original image to the consumer. When the original image has been transferred, the image producer calls this filter's `imageComplete()` method, which does the real work. Instead of relaying the completion status to the consumer, `imageComplete()` starts generating its own data: solid rectangles that are all in the `overlapColor` specified in the constructor. It sends these rectangles to the consumer by calling

`consumer.setPixels()`. After each rectangle, it calls `consumer.imageComplete()` with the `SINGLEFRAMEDONE` flag, meaning that it has just finished one frame of a multi-frame sequence. When the rectangles have completely covered the image, the method `imageComplete()` finally notifies the consumer that the entire image sequence has been transferred by sending the `STATICIMAGEDONE` flag.

The following code is a simple applet that uses this image filter to produce a new image:

```
import java.applet.*;
import java.awt.*;
import java.awt.image.*;
public class DynamicImages extends Applet {
    Image i, j;
    public void init () {
        i = getImage (getDocumentBase(), "rosey.jpg");
        j = createImage (new FilteredImageSource (i.getSource(),
            new DynamicFilter(250, 10, Color.red));
        }
    public void paint (Graphics g) {
        g.drawImage (j, 10, 10, this);
    }
}
```

One final curiosity: the `DynamicFilter` doesn't make any assumptions about the color model used for the original image. It sends its overlays with the default RGB color model. Therefore, this is one case in which an `ImageConsumer` may see calls to `setPixels()` that use different color models.

12.5.2 *RGBImageFilter*

`RGBImageFilter` is an abstract subclass of `ImageFilter` that provides a shortcut for building the most common kind of image filters: filters that independently modify the pixels of an existing image, based only on the pixel's position and color. Because `RGBImageFilter` is an abstract class, you must subclass it before you can do anything. The only method your subclass must provide is `filterRGB()`, which produces a new pixel value based on the original pixel and its location. A handful of additional methods are in this class; most of them provide the behind-the-scenes framework for funneling each pixel through the `filterRGB()` method.

If the filtering algorithm you are using does not rely on pixel position (i.e., the new pixel is based only on the old pixel's color), AWT can apply an optimization for images that use an `IndexColorModel`: rather than filtering individual pixels, it can filter the image's color map. In order to tell AWT that this optimization is okay, add a constructor to the class definition that sets the `canFilterIndexColorModel` variable to `true`. If `canFilterIndexColorModel` is `false` (the default) and an `IndexColorModel` image is sent through the filter, nothing happens to the image.

Variables

protected boolean canFilterIndexColorModel

Setting the `canFilterIndexColorModel` variable permits the `ImageFilter` to filter `IndexColorModel` images. The default value is `false`. When this variable is `false`, `IndexColorModel` images are not filtered. When this variable is `true`, the `ImageFilter` filters the colormap instead of the individual pixel values.

protected ColorModel newmodel

The `newmodel` variable is used to store the new `ColorModel` when `canFilterIndexColorModel` is `true` and the `ColorModel` actually is of type `IndexColorModel`. Normally, you do not need to access this variable, even in subclasses.

protected ColorModel origmodel

The `origmodel` variable stores the original color model when filtering an `IndexColorModel`. Normally, you do not need to access this variable, even in subclasses.

Constructors

public RGBImageFilter ()—called by subclass

The only constructor for `RGBImageFilter` is the implied constructor with no parameters. In most subclasses of `RGBImageFilter`, the constructor has to initialize only the `canFilterIndexColorModel` variable.

ImageConsumer interface methods

public void setColorModel (ColorModel model)

The `setColorModel()` method changes the `ColorModel` of the filter to `model`. If `canFilterIndexColorModel` is `true` and `model` is of type `IndexColorModel`, a filtered version of `model` is used instead.

public void setPixels (int x, int y, int w, int h, ColorModel model, byte pixels[], int off, int scansize)

public void setPixels (int x, int y, int w, int h, ColorModel model, int pixels[], int off, int scansize)

If necessary, the `setPixels()` method converts the `pixels` buffer to the default `RGB ColorModel` and then filters them with `filterRGBPixels()`. If `model` has already been converted, this method just passes the `pixels` along to the consumer's `setPixels()`.

Other methods

The only method you care about here is `filterRGB()`. All subclasses of `RGBImageFilter` *must* override this method. It is very difficult to imagine situations in which you would override (or even call) the other methods in this group. They are helper methods that funnel pixels through `filterRGB()`.

```
public void substituteColorModel (ColorModel oldModel, ColorModel newModel)
```

`substituteColorModel()` is a helper method for `setColorModel()`. It initializes the protected variables of `RGBImageFilter`. The `origModel` variable is set to `oldModel` and the `newModel` variable is set to `newModel`.

```
public IndexColorModel filterIndexColorModel (IndexColorModel icm)
```

`filterIndexColorModel()` is another helper method for `setColorModel()`. It runs the entire color table of `icm` through `filterRGB()` and returns the filtered `ColorModel` for use by `setColorModel()`.

```
public void filterRGBPixels (int x, int y, int width, int height, int pixels[], int off,
int scansize)
```

`filterRGBPixels()` is a helper method for `setPixels()`. It filters each element of the `pixels` buffer through `filterRGB()`, converting pixels to the default RGB `ColorModel` first. This method changes the values in the `pixels` array.

```
public abstract int filterRGB (int x, int y, int rgb)
```

`filterRGB()` is the one method that `RGBImageFilter` subclasses must implement. The method takes the `rgb` pixel value at position (x, y) and returns the converted pixel value in the default RGB `ColorModel`. Coordinates of $(-1, -1)$ signify that a color table entry is being filtered instead of a pixel.

A transparent image filter that extends RGBImageFilter

Creating your own `RGBImageFilter` is fairly easy. One of the more common applications for an `RGBImageFilter` is to make images transparent by setting the alpha component of each pixel. To do so, we extend the abstract `RGBImageFilter` class. The filter in Example 12-9 makes the entire image translucent, based on a percentage passed to the class constructor. Filtering is independent of position, so the constructor can set the `canFilterIndexColorModel` variable. A constructor with no arguments uses a default alpha value of 0.75.

Example 12-9: TransparentImageFilter Source

```
import java.awt.image.*;
class TransparentImageFilter extends RGBImageFilter {
    float alphaPercent;
    public TransparentImageFilter () {
        this (0.75f);
    }
    public TransparentImageFilter (float aPercent)
        throws IllegalArgumentException {
        if ((aPercent < 0.0) || (aPercent > 1.0))
            throw new IllegalArgumentException();
        alphaPercent = aPercent;
        canFilterIndexColorModel = true;
    }
}
```

Example 12–9: TransparentImageFilter Source (continued)

```

    public int filterRGB (int x, int y, int rgb) {
        int a = (rgb >> 24) & 0xff;
        a *= alphaPercent;
        return ((rgb & 0x00ffffff) | (a << 24));
    }
}

```

12.5.3 CropImageFilter

The `CropImageFilter` is an `ImageFilter` that crops an image to a rectangular region. When used with `FilteredImageSource`, it produces a new image that consists of a portion of the original image. The cropped region must be completely within the original image. It is never necessary to subclass this class. Also, using the 10 or 11 argument version of `Graphics.drawImage()` introduced in Java 1.1 precludes the need to use this filter, unless you need to save the resulting cropped image.

If you crop an image and then send the result through a second `ImageFilter`, the pixel array received by the filter will be the size of the original `Image`, with the `offset` and `scansize` set accordingly. The `width` and `height` are set to the cropped values; the result is a smaller `Image` with the same amount of data. `CropImageFilter` keeps the full pixel array around, partially empty.

Constructors

public CropImageFilter (int x, int y, int width, int height) ★

The constructor for `CropImageFilter` specifies the rectangular area of the old image that makes up the new image. The `(x, y)` coordinates specify the top left corner for the cropped image; `width` and `height` must be positive or the resulting image will be empty. If the `(x, y)` coordinates are outside the original image area, the resulting image is empty. If `(x, y)` starts within the image but the rectangular area of size `width height` goes beyond the original image, the part that extends outside will be black. (Remember the color black has pixel values of 0 for red, green, and blue.)

ImageConsumer interface methods

public void setProperties (Hashtable properties) ★

The `setProperties()` method adds the `croprect` image property to the properties list. The bounding `Rectangle`, specified by the `(x, y)` coordinates and `width height` size, is associated with this property. After updating properties, this method sets the properties list of the consumer.

public void setDimensions (int width, int height) ★

The `setDimensions()` method of `CropImageFilter` ignores the `width` and `height` parameters to the function call. Instead, it relies on the size parameters in the constructor.

public void setPixels (int x, int y, int w, int h, ColorModel model, byte pixels[], int offset, int scansize) ★

public void setPixels (int x, int y, int w, int h, ColorModel model, int pixels[], int offset, int scansize) ★

These `setPixels()` methods check to see what portion of the `pixels` array falls within the cropped area and pass those pixels along.

Cropping an image with CropImageFilter

Example 12-10 uses a `CropImageFilter` to extract the center third of a larger image. No subclassing is needed; the `CropImageFilter` is complete in itself. The output is displayed in Figure 12-7.

Example 12-10: Crop Applet Source

```
import java.applet.*;
import java.awt.*;
import java.awt.image.*;
public class Crop extends Applet {
    Image i, j;
    public void init () {
        MediaTracker mt = new MediaTracker (this);
        i = getImage (getDocumentBase(), "rosey.jpg");
        mt.addImage (i, 0);
        try {
            mt.waitForAll();
            int width      = i.getWidth(this);
            int height     = i.getHeight(this);
            j = createImage (new FilteredImageSource (i.getSource(),
                new CropImageFilter (width/3, height/3,
                width/3, height/3));
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    public void paint (Graphics g) {
        g.drawImage (i, 10, 10, this);           // regular
        if (j != null) {
            g.drawImage (j, 10, 90, this);      // cropped
        }
    }
}
```



Figure 12–7: Image cropping example output.

TIP You can use `CropImageFilter` to help improve your animation performance or just the general download time of images. Without `CropImageFilter`, you can use `Graphics.clipRect()` to clip each image of an image strip when drawing. Instead of clipping each `Image` (each time), you can use `CropImageFilter` to create a new `Image` for each cell of the strip. Or for times when an image strip is inappropriate, you can put all your images within one image file (in any order whatsoever), and use `CropImageFilter` to get each out as an `Image`.

12.5.4 *ReplicateScaleFilter*

Back in Chapter 2 we introduced you to the `getScaledInstance()` method. This method uses a new image filter that is provided with Java 1.1. The `ReplicateScaleFilter` and its subclass, `AreaAveragingScaleFilter`, allow you to scale images before calling `drawImage()`. This can greatly speed your programs because you don't have to wait for the call to `drawImage()` before performing scaling.

The `ReplicateScaleFilter` is an `ImageFilter` that scales by duplicating or removing rows and columns. When used with `FilteredImageSource`, it produces a new image that is a scaled version of the original. As you can guess, `ReplicateScaleFilter` is very fast, but the results aren't particularly pleasing aesthetically. It is great if you want to magnify a checkerboard but not that useful if you want to scale an image of your Aunt Polly. Its subclass, `AreaAveragingScaleFilter`, implements a more time-consuming algorithm that is more suitable when image quality is a concern.

Constructor

public ReplicateScaleFilter (int width, int height)

The constructor for `ReplicateScaleFilter` specifies the size of the resulting image. If either parameter is -1, the resulting image maintains the same aspect ratio as the original image.

ImageConsumer interface methods

public void setProperties (Hashtable properties)

The `setProperties()` method adds the rescale image property to the properties list. The value of the rescale property is a quoted string showing the image's new width and height, in the form "<width>x<height>", where the width and height are taken from the constructor. After updating properties, this method sets the properties list of the consumer.

public void setDimensions (int width, int height)

The `setDimensions()` method of `ReplicateScaleFilter` passes the new width and height from the constructor along to the consumer. If either of the constructor's parameters are negative, the size is recalculated proportionally. If both are negative, the size becomes width height.

public void setPixels (int x, int y, int w, int h, ColorModel model, int pixels[], int offset, int scansize)

public void setPixels (int x, int y, int w, int h, ColorModel model, byte pixels[], int offset, int scansize)

The `setPixels()` method of `ReplicateScaleFilter` checks to see which rows and columns of pixels to pass along.

12.5.5 AreaAveragingScaleFilter

The `AreaAveragingScaleFilter` subclasses `ReplicateScaleFilter` to provide a better scaling algorithm. Instead of just dropping or adding rows and columns, `AreaAveragingScaleFilter` tries to blend pixel values when creating new rows and columns. The filter works by replicating rows and columns to generate an image that is a multiple of the original size. Then the image is resized back down by an algorithm that blends the pixels around each destination pixel.

AreaAveragingScaleFilter methods

Because this filter subclasses `ReplicateScaleFilter`, the only methods it includes are those that override methods of `ReplicateScaleFilter`.

Constructors

public AreaAveragingScaleFilter (int width, int height) ★

The constructor for `AreaAveragingScaleFilter` specifies the size of the resulting image. If either parameter is -1, the resulting image maintains the same aspect ratio as the original image.

ImageConsumer interface methods

public void setHints (int hints) ★

The `setHints()` method of `AreaAveragingScaleFilter` checks to see if some optimizations can be performed based upon the value of the `hints` parameter. If they can't, the image filter has to cache the pixel data until it receives the entire image.

public void setPixels (int x, int y, int w, int h, ColorModel model, byte pixels[], int offset, int scansize) ★

public void setPixels (int x, int y, int w, int h, ColorModel model, int pixels[], int offset, int scansize) ★

The `setPixels()` method of `AreaAveragingScaleFilter` accumulates the pixels or passes them along based upon the available hints. If `setPixels()` accumulates the pixels, this filter passes them along to the consumer when appropriate.

12.5.6 Cascading Filters

It is often a good idea to perform complex filtering operations by using several filters in a chain. This technique requires the system to perform several passes through the image array, so it may be slower than using a single complex filter; however, cascading filters yield code that is easier to understand and quicker to write—particularly if you already have a collection of image filters from other projects.

For example, assume you want to make a color image transparent and then render the image in black and white. The easy way to do this task is to apply a filter that converts color to a gray value and then apply the `TransparentImageFilter` we developed in Example 12-9. Using this strategy, we have to develop only one very simple filter. Example 12-11 shows the source for the `GrayImageFilter`; Example 12-12 shows the applet that applies the two filters in a daisy chain.

Example 12–11: GrayImageFilter Source

```
import java.awt.image.*;
public class GrayImageFilter extends RGBImageFilter {
    public GrayImageFilter () {
        canFilterIndexColorModel = true;
    }
}
```

Example 12–11: GrayImageFilter Source (continued)

```

    public int filterRGB (int x, int y, int rgb) {
        int gray = (((rgb & 0xff0000) >> 16) +
                    ((rgb & 0x00ff00) >> 8) +
                    (rgb & 0x0000ff)) / 3;
        return (0xff000000 | (gray << 16) | (gray << 8) | gray);
    }
}

```

Example 12–12: DrawingImages Source

```

import java.applet.*;
import java.awt.*;
import java.awt.image.*;
public class DrawingImages extends Applet {
    Image i, j, k, l;
    public void init () {
        i = getImage (getDocumentBase(), "rosej.jpg");
        GrayImageFilter gif = new GrayImageFilter ();
        j = createImage (new FilteredImageSource (i.getSource(), gif));
        TransparentImageFilter tf = new TransparentImageFilter (.5f);
        k = createImage (new FilteredImageSource (j.getSource(), tf));
        l = createImage (new FilteredImageSource (i.getSource(), tf));
    }
    public void paint (Graphics g) {
        g.drawImage (i, 10, 10, this);           // regular
        g.drawImage (j, 270, 10, this);         // gray
        g.drawImage (k, 10, 110, Color.red, this); // gray - transparent
        g.drawImage (l, 270, 110, Color.red, this); // transparent
    }
}

```

Granted, neither the `GrayImageFilter` or the `TransparentImageFilter` are very complex, but consider the savings you would get if you wanted to blur an image, crop it, and then render the result in grayscale. Writing a filter that does all three is not a task for the faint of heart; remember, you can't subclass `RGBImageFilter` or `CropImageFilter` because the result does not depend purely on each pixel's color and position. However, you can solve the problem easily by cascading the filters developed in this chapter.