
Version 1.0 Beta 1

Converting Applets that Use Images

This page will describe the basic requirements for porting an applet that used images in the Alpha3 HotJava browser to the new portable Beta image API. It can also be used as a tutorial for writing new applets that will access, lay out, draw, and or modify images in the new Beta image API.

There is one important new feature in this release. A new class was added to the java.awt package to help you in tracking the asynchronous loading of images. See the documentation on the MediaTracker class for more information and an example.

For sake of avoiding information overload where it is not needed, this document will break applets down into 4 basic types:

1. Applets which simply draw images loaded from a URL.
2. Applets which modify their layout to fit images loaded from a URL.
3. Applets which render to an off screen image for double buffering
4. Applets which create new images as the result of calculations or which modify existing images using image filters.

How do I draw an image loaded from a URL?

The simplest method for drawing an image has changed very little from the HotJava Alpha3 API. The following code illustrates how this is done:

```
public class SimpleImage extends Applet {
    Image myimg;
    public void init() {
        myimg = getImage(getDocumentBase(), "myimg.gif");
    }
    public void paint(Graphics g) {
        // This call draws the image at its normal size at the
        // upper left corner of the applet.
        g.drawImage(myimg, 0, 0, this);
        // The next call draws the image scaled to fit into a
        // 100x100 rectangle at the upper left corner of the applet.
        g.drawImage(myimg, 0, 0, 100, 100, this);
    }
}
```

This example shows all that is needed to simply render an image using the new Beta

image API. If that is all an applet needs to do with images, then the example can be used as a template for the applet's code. What follows is an explanation of how this template works for those who are curious or for those who need an introduction to more advanced use of images. The two changes from Alpha3 visible in the code are the presence of the getDocumentBase() call when getting the image and the presence of an extra paramenter in the drawImage() calls.

There are now two methods provided in the Applet API to get a handle to an image given a URL or an "href" specification. These methods are:

- Applet.getImage(URL url);
- Applet.getImage(URL url, String href);

The Alpha3 Applet API provided a method which would take a simple "href" specification and the browser would first try to interpret that href relative to the URL of where the applet was loaded. If that failed, then it would try to interpret that href relative to the URL of where the document containing the applet was loaded. Since this search was potentially time consuming and since it could lead to confusing behavior caused by the environment in which the applet's code was stored, this interface was eliminated in favor of the explicit interfaces listed above. Two new methods were added to provide easy access to the two most common URLs which an applet may wish to use as a base for loading its images:

- Applet.getDocumentBase();
- Applet.getCodeBase();

These methods can be used as shown in the init() method of the example above.

There are now two methods provided by the Graphics class used to draw an image. These methods are:

- Graphics.drawImage(Image img, int x, int y, (ImageObserver) this);
- Graphics.drawImage(Image img, int x, int y, int width, int height, (ImageObserver) this);

The final parameter was added so that images could be loaded asynchronously as they are needed. The drawImage method will always return immediately even if the image being drawn has not yet been loaded. In order to ensure that an Applet will be able to draw the final image after it has been completely loaded and scaled to the desired size, an ImageObserver parameter was added to the drawImage calls. Since the Applet class already implements the method needed to support this ImageObserver interface, and since the default implementation of that method automatically calls the applet's repaint method when new data is available for the images that the applet is drawing, most applets should not normally need to be concerned about this interface other than to pass in "this" as the final parameter to all drawImage calls.

How do I modify my layout to fit images loaded from a URL?

Most applets will probably have a fixed layout. Either the code for the applet will be specific to the images and other media involved in displaying the applet and it will contain values which specify the layout and positions of those graphics, or the applet will parse the layout of its graphics from values specified in the applet parameters which appear in the HTML file.

Some more sophisticated applets will want to investigate the size of the available fonts in the environment in which they are run or they may want to automatically adapt their size or layout to the size of the various images and other media that their parameters instruct them to load. If an applet wants to dynamically control its layout in this manner based on the size of images, then it will have to wait until the images have been loaded before it can determine its layout.

In general, since modifying the size of an applet will probably involve recalculating the layout of the HTML page on which it appears, it is much preferred for an applet to adapt itself to whatever size was specified in the attributes of the `<APPLET>` tag which invoked it than to change that size after it has loaded.

Whether an applet wishes to change its size, or whether it will simply change the scaling and location of various images and graphics that it draws based on the sizes of its media, it will need to deal with the fact that image dimensions are not known until some unspecified time after the applet has been created due to the asynchronous nature in which the data for images is loaded. Most Alpha3 applets would be able to query the image sizes in their init methods and resize themselves based on that information, but that method will no longer work in the new image APIs. Additionally, the problem could not be solved by the default implementation of the imageUpdate method as was done for simpler applets in the previous example since there is no standard method that Applets use to reevaluate their size.

There are 3 ways of dealing with this issue:

1. The applet could adjust the size of the media to fit its given dimensions.
2. The applet could use the MediaTracker class to wait for all of the image data for particular subsets of its images to become available.
3. The applet could implement the ImageObserver interface and observe the status of the images as they are loaded and constructed.

Adjusting the size of the media to fit the Applet Specify a size for the applet in the <APPLET> tag and honor that fixed size in the applet code. In particular, dynamically determine the location and scaling of the images on the fly in the paint method to fit the size that was specified in the <APPLET> tag.

```
/*
 * This applet draws an image scaled to its width and height
 * as specified in the <APPLET> tag.
 */
Image myimg;
```

```
public void init() {
    myimg = getImage(getDocumentBase(), "myimg.gif");
}

public void paint(Graphics g) {
    int appw = size().width;
    int apph = size().height;
    if ((checkImage(myimg, null) & ERROR) != 0) {
        // The image had a problem - just draw a big red rect
        g.setColor(Color.red);
        g.fillRect(0, 0, appw, apph);
        return;
    }
    // Scale the image to the width and height of the applet
    g.drawImage(myimg1, 0, 0, appw, apph, this);
}
```

Delaying layout until the image data becomes available Use the MediaTracker to track the loading of the applet's images and then perform the layout when all of the necessary images have been fully loaded. If you are going to wait for the data to be loaded, then you should fork a thread in your start method and wait for the images to load in that thread. If you call any of the methods in the MediaTracker class which block waiting for information (such as waitForAll() or waitForID()) in the thread which executes your init, start, paint, or update methods then you will probably block a system thread of the browser in which you are running and delay the rest of the applets from being processed.

> There is an example of how to use this class in the documentation for the MediaTracker class

Implementing the ImageObserver interface to track image sizes Provide a custom implementation of the ImageObserver interface which calls a custom layout method when size information is available and use a state variable to communicate the status of the calculated layout to the paint and update methods so that they don't draw the final results until the layout is completely determined.

```
/*
 * This applet draws a big image scaled to its width and height as
 * specified in the <APPLET> tag, and a small image scaled by the
 * same ratio as the big image and positioned in the center of it.
 */
Image bigimg, smallimg;
int smallx, smally, smallw, smallh;
boolean sizeknown = false;
boolean errored = false;

public void init() {
    bigimg = getImage(getDocumentBase(), "bigimg.gif");
    smallimg = getImage(getDocumentBase(), "smallimg.gif");
    positionImages();
}

public boolean imageUpdate(Image theimg, int infoflags,
                           int x, int y, int w, int h) {
```

```
        if (infoflags & (ERROR)) {
            errored = true;
        }
        if (infoflags & (WIDTH | HEIGHT) != 0) {
            positionImages();
        }
        boolean done = (infoflags & (ERROR | FRAMEBITS | ALLBITS)) != 0;
        // Repaint immediately if we are done, otherwise batch up
        // repaint requests every 100 milliseconds
        repaint(done ? 0 : 100);
        return !done;
    }

    public synchronized void positionImages() {
        int bigw = bigimg.getWidth(this);
        int bigh = bigimg.getHeight(this);
        int smallw = bigimg.getWidth(this);
        int smallh = bigimg.getHeight(this);
        if (bigw < 0 || bigh < 0 || smallw < 0 || smallh < 0) {
            return;
        }
        smallw = smallw * bigw / size().width;
        smallh = smallh * bigh / size().height;
        smallx = (bigw - smallw) / 2;
        smally = (bigh - smallh) / 2;
        sizeknown = true;
    }

    public synchronized void paint(Graphics g) {
        int appw = size().width;
        int apph = size().height;
        if (errored) {
            // The images had a problem - just draw a big red rectangle
            g.setColor(Color.red);
            g.fillRect(0, 0, appw, apph);
            return;
        }
        // Scale the big image to the width and height of the applet
        g.drawImage(bigimg, 0, 0, appw, apph, this);
        if (sizeknown) {
            // Scale the small image to the central region calculated above.
            g.drawImage(smallimg, smallx, smally, smallw, smallh, this);
        }
    }
```

How do I use an off screen image buffer for double buffering?

The API for double buffering is very similar to the HotJava Alpha3 API for double buffering. Both involve the creation of an off screen image buffer and a graphics object which uses it as the target for the lengthy rendering operation, and both involve copying that image to the screen using the same API as for drawing regular images.

```
    public class DoubleBuffer extends Applet {
        Image offscrImg;
        Graphics offscrG;
```

```
    public void init() {
        offscrImg = createImage(size().width, size().height);
        offscrG = offscrimg.getGraphics();
    }
    public void paint(Graphics g) {
        // Draw the new animation frame into the offscreen image.
        // Note: this is just a simple grid of lines...
        int appw = size().width;
        int apph = size().height;
        offscrG.setColor(getBackground());
        offscrG.fillRect(0, 0, appw, apph);
        offscrG.setColor(Color.black);
        for (int x = 0; x < appw; x += 10) {
            offscrG.drawLine(x, 0, x, apph);
        }
        offscrG.setColor(Color.white);
        for (int y = 0; y < apph; y += 10) {
            offscrG.drawLine(0, y, appw, y);
        }
        g.drawImage(offscrImg, 0, 0, this);
    }
}
```

The only differences between the Alpha3 API and the new Beta API are the way in which the graphics object is created and the extra parameter to the drawImage call which was discussed in the first section above. In Alpha3, a Graphics object for the offscreen image was created using a constructor from the standard Graphics class [`new Graphics(offscrImg);`]. In Beta, the Image object provides a factory method for generating a new Graphics object which can use that image as a destination drawable surface.

```
Image image = createImage(width, height);
Graphics g = image.getGraphics();
```

It is an error to call the getGraphics() method on an image that was not created by the createImage (int width, int height) method.

How do I create new images or modify existing images?

A number of applets have been written which rely on calculations to produce images on the fly or which perform filtering operations on an existing image to produce a number of new variants of an image on the fly. There are a number of utility classes available which help to perform most of the work involved in these operations such as RGBImageFilter and MemoryImageSource. The API documentation for those classes includes examples of how to use those classes in conjunction with the createImage() call to define a simple memory image and a simple image filter. Further examples can be found in the DitherTest applet, the MoleculeViewer applet, and the ImageMap applet. Even with the existance of the utility classes, it is still a good idea for an applet programmer to understand the ImageProducer and ImageConsumer interfaces upon which the Beta image architecture is based.

The ImageProducer interface is implemented by objects which know how to produce the raw image data for an image. Examples of such objects include those that load images from files and URLs, objects which produce image data from an array of pixels, and objects which filter the output of some other existing ImageProducer object.

The ImageProducer interface defines a number of methods for registering objects which implement the ImageConsumer interface. The ImageConsumer objects implement an interface which the ImageProducer uses to deliver the raw image data. This raw image data consists of 6 different pieces:

1. The dimensions of the raw image (width and height). [see ImageConsumer.setDimensions]
2. A Hashtable which defines an extensible list of properties of the image. [see ImageConsumer.setProperties]
3. A default ColorModel object which will describe how to produce color information from raw pixels for the majority of the pixels in the image. [see ImageConsumer.setColorModel]
4. A set of hints which inform the ImageConsumer of the order in which the ImageProducer expects to deliver the pixels. [see ImageConsumer.setHints]
5. The pixels for the image itself, formatted as any number of arrays of bytes and/or integers, each tagged with a ColorModel object. [see ImageConsumer.setPixels]
6. The last piece of information for most images (or the final piece of information for each frame of a multi–frame image) is that the image (or frame) is done. [see ImageConsumer.imageComplete]
