

What's ^{so} HOT about Java?

If you think this page reads more like a TV commercial than a technical article, welcome to Java.

Finding information about Java is certainly no chore. Java is portable and architecture-neutral for

any system implementing the Java Virtual Machine.

Programs written in Java can be distributed and executed on any client system that has a Java-enabled Web browser. Developers are freed from having to write multiple versions for multiple platforms and even from having to recompile for each platform.

Because the applets run on the client system, scalability and performance are no longer tightly tied to Web server systems. High performance requirements are also addressed through multithreading, the ability to link native code, and the Just-in-Time compiler.

Although understanding Java may come easily, true confidence is more likely to come through actually working with it. This

Website will guide you through working with the Java Development Toolkit for AIX. (Because Java is platform-independent, these exercises will work with other operating systems as well as the AIX/UNIX environment.)

You will learn to run demonstrations as well as to develop your own example applications and applets. You will see how

easy it is to format graphical user interfaces and handle user inputs, to build multithreaded animations, and to communicate among applets and to servers over the network.

“What’s So HOT About Java?” begins by thoroughly defining and describing Java, intended for readers needing a basic understanding of Java and how to apply it to their environment.

The examples provided in this article will provide step-by-step instructions for creating two Java applets, complete with sample code and illustrations.

**Is it portable
and interpreted
or high-
performance?...
Both!**

**Is Java an
object-oriented
programming
language or
a distributed
run-time
environment?...
Both!**

GETTING STARTED

Before we can really get started, we need to get the essentials out of the way. If you haven’t already installed the Java Development Toolkit (JDK) and a Java-enabled Web browser, you’ll need to do so. You can also download a Java-enabled Web Browser (HotJava), which is written in Java. Be sure to use Netscape Navigator 3 Web Browser, which is Java- and JavaScript-enabled. You won’t be able to use the Navigator 2, as it isn’t Java-enabled. The download sites for both the JDK and the Navigator include README files with step-by-step instructions for installing the products.

Once you install the JDK 1.0.2 and the Navigator 3, make sure everything is properly in place by invoking the JDK demos. Figure 1 includes commands to enter at a shell (ksh) command prompt to set up and run the JDK demos.

```
>export PATH=$PATH:/usr/lpp/Java/bin:.  
>cd /usr/lpp/Java/demo  
>export DISPLAY=mydisplay:0  
>CompileDemos  
>RunDemos
```

Figure 1. Commands to Enter at a Shell (ksh) Command Prompt to Set Up and Run the JDK Demos

On the export DISPLAY command, replace mydisplay:0 with your display’s actual name. Note that when you enter the RunDemos command, three windows (representing the three HTML files in the Animator demo) will be displayed. These demos use the appletviewer to run the applets. The JDK provides the appletviewer so you can view applets without a Java-enabled browser.

Later, when we start writing our own applets, we’ll use the Navigator browser instead of the appletviewer. If you haven’t already noticed, the appletviewer has an Applet menu button. Use this button to Quit an applet, so the next demo will start.

As you invoke the applets, you should notice a few things in preparation for our later examples. First, look at the Animator applet. If you’ve already gone to the next applet, just enter

What's So Hot About Java?

appletviewer Animator/example1.html. In Java, basic animation is performed in one of three ways:

- A series of images (GIF, JPEG, or XDM) are displayed in succession
- A single image is successively redrawn at different coordinates, creating the effect of movement
- A single image is successively enlarged or reduced, creating the effect of growth or of an explosion

The Animator applet uses the first method, displaying a series of images in succession. In fact, the Animator applet was written so you can use parameters to specify the directory where images are stored, the number of images to store, the time to pause between displaying each image, the audio files to include, and more. To get a better understanding, let's take a quick look at the applet source and images. You can view a single image in your browser by creating a file (call it /tmp/showgif.html) and inserting the line ``. Then, invoke the Netscape Browser (netscape &) and set the Location field on the menu to file:/tmp/showgif.html. If the netscape command fails, try entering `export LANG=C` and then restart netscape.

Now let's check out the applet source. Java source code is always stored in files with the postfix .java. Animator/Animator.java. Search for the field `getParameterInfo`. The first column of this array lists parameters that Animator accepts. To see how to set these parameters, display the Animator/example1.html file and note the param tags. By supplying your own set of image files, you can use the Animator applet to animate any Web page.

A FEW BASIC JAVA CONCEPTS

Now we're ready to start our own applet. Refer to Figure 2 for the steps to create an applet. But don't start this procedure yet.

First, you need to understand a few basic Java concepts. Look at the HelloWorld.java source file. A key feature of object technology (OT) is *reusability*. Even the simplest applet takes advantage of this feature. Notice the line creating the HelloWorld class "extends java.applet.Applet". The Applet class is what allows us to avoid rewriting the GUI interface already provided by the browser. Since applets by definition are Java programs invoked from a browser, all applets must extend the Applet class. By extending Applet, we need only to implement those methods and variables we want to override.

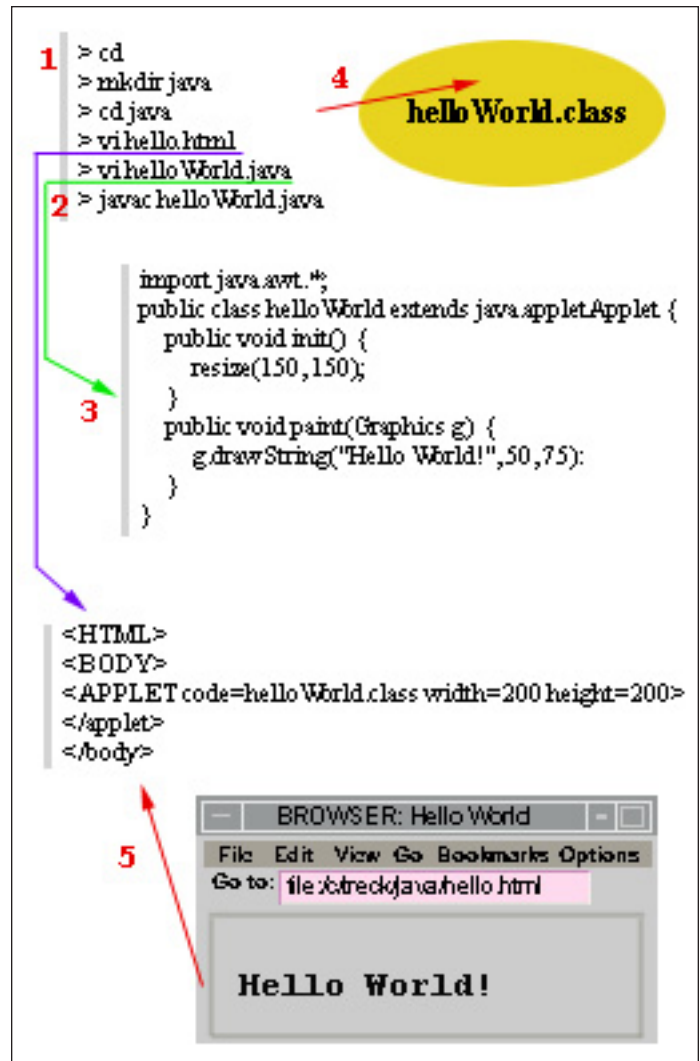


Figure 2. Five Steps to Creating an Applet

Now is a good time to view the API documentation, which is on the Sun site. The API documentation is very important, so I'd recommend adding it to your browser Bookmarks. Notice the list of Java Packages. In Java, almost everything is either a class, an object, or a method. Java groups classes into packages. The Abstract Window Toolkit (AWT or java.awt) package provides the windowing and GUI classes. The java.applet package provides classes required to write a program that can be invoked from within a browser.

In Java, we tell the compiler which class we're referencing by giving the package and class name (for example, java.applet.Applet). If you'll be using a lot of classes from a package, you can simplify your programs by "importing" a package or class. Notice that our HelloWorld applet starts with the

What's So Hot About Java?

statement, `import java.awt.*`, to import all the classes in the AWT package. Imported classes don't need to be prefixed with their package name. For example, if we had imported `java.applet.*`, our `helloWorld` class definition could simply extend `Applet` instead of `java.applet.Applet`. The `java.lang` package is the only package automatically imported by the Java compiler. We'll talk more about packages and interfaces in later sections.

In the API documentation, select the `java.applet` link, then the `Class Applet` link. The `Class Applet` page describes the variables and methods that are part of the `Applet` class. (Don't worry if this looks foreign right now, but if you plan to write applets on your own later, it's helpful to start learning how to use the API documentation.) Notice the list of methods included in the `Applet` class. The key points are:

- If the user leaves your page, the browser will call the `stop()` method.
- If the user returns to your page, the browser will call the `start()` method again.
- When the user exits the browser, the browser will call `stop()`, then `destroy()`.

In our `helloWorld` applet, we can override any of these methods by creating our own versions. In `init()`, we'd do things like sizing the window, setting background colors, and creating our applet's layout. Ours is a simple applet, so we just set the window size in `init()` and leave the `start()`, `stop()`, and `destroy()` methods as provided for now.

Notice that our applet has a `paint` method, but `paint()` wasn't listed as one of the methods for the `Applet` class. How does it get called? Since our `helloWorld` applet extends the `Applet` class, it's a subclass of `Applet`. As such, our `helloWorld` applet inherits the methods and variables implemented in `Applet`.

Referring to the API documentation for `Applet`, notice that the declaration for the `Applet` class indicates that it extends `Panel`; therefore, it's a subclass of `Panel`. (Go ahead, click on `Panel`.) `Panel` is a subclass of `Container`, which is a subclass of `Component`. `Component` includes a `paint()` method called by the browser any time the window is updated (resized, redisplayed, or altered in any way). Input to `paint()` is a `Graphics` object representing the graphics area of this component (our applet). So for an applet, `paint()` is crucial. Our simple applet just paints the string "Hello World!" but we'll build on this later.

About now you might be thinking, "This sounds a little too abstract. I mean, how was I supposed to know to go from `Applet` to `Panel` to `Container` to `Component` just to find the `paint()` method?" Well, the good news is that you need to know only a few of these fundamental relationships or "flows," and we go through them in this series. The other piece of good news is that as you become familiar with the API documentation, you'll more easily pick up on when to call a method to perform a task and when to override a method.

FIVE STEPS TO CREATING AN APPLET

NOTE: Be sure to get the source code provided in the downloadable package.

Getting back to our `helloWorld` example, use Figure 2 and the following steps to build your `helloWorld` applet.

1. Create a `java` directory in your home directory to store everything you create throughout this article.
2. In your home `java` directory, create `hello.html` for the browser to read.
3. Create the Java source file, `helloWorld.java`, as shown in Figure 2.
4. Compile the source file with the Java compiler (`javac`).
5. Set the Location in your browser to the full path for the `hello.html` file you just created and press Enter to test your work. *Voila!* Not earth shattering, but it's a start. Now that we have the basics, we can move on to the fun.

ADDING SOME BUZZ

By now, we have a simple applet and HTML example, and we understand the basics of applet flow and using the API documentation. Now, let's add some buzz to our applet; let's add some animation. Earlier, when we tested the `Animator` demo, we discussed the three basic ways of creating animations. For our `helloWorld` applet, we'll be displaying a series of

```
cd $HOME/java
mkdir images
cp /usr/lpp/Java/demo/Animator/images/Duke/*images
mv images/T10.gif images/T0.gif
```

Figure 3. Commands for Copying Images from the `Animator` Demo

What's So Hot About Java?

images. So you don't have to generate your own images, just use the commands in Figure 3 to copy the images from the Animator demo, for a series of 10 (0-9) image files.

Since we know the animation is just displaying a series of images, we know we need to add a loop somewhere. But where? Obviously, not in `init()`, because our program would then stay in `init()` forever. Putting it in `paint()` doesn't make much sense, either. In fact, our proposed loop doesn't really fit anywhere in the applet flow — `init()`, `start()`, `stop()`, `destroy()` — that we discussed. What we really need is a way to just spawn off our proposed loop. Sounds like a thread, right? So, to add animation, we really need to add multithreading.

Since multithreading is implemented at the language level in Java, the `Thread` class is a member of the `java.lang` package. Use the API documentation to view the `Thread` class in the `java.lang` package. Notice that like the `Applet` class, the `Thread` class has `start()`, `stop()`, and `destroy()` methods we can override. It also has a `run()` method that is called by `start()`. Good. So let's put our loop in the `run()` method. Java doesn't implement multiple inheritance, so if we chose to have `helloWorld` extend `Thread` instead of `Applet`, we'd lose our interface to the browser. Not so good.

This is where *Interfaces* come in. Java provides Interfaces as templates of behavior, just as a class is a template. But Interfaces pass *only* method descriptions to their children. If you are still in the API documentation for the `Thread` class, notice that it implements the `Runnable` Interface. Click on `Runnable`, noting that `Runnable` defines only an abstract `run()` method. Okay. If `helloWorld` continues to extend `Applet` (so it keeps its interface to the browser) but also implements `Runnable` (so it adds the `run()` method), we have a place to put our loop. Perfect! Now we know that we can access Java's multithreading capabilities in two ways. We can either define our class as a subclass of `Thread`, or we can implement the `Runnable` interface. Figure 4 shows our modified version of `helloWorld`, which now implements the `Runnable` interface.

(Ignore the big "1." It's not part of the code, so we'll explain it later.) Notice that we've also added a `run()` method, which has our proposed loop. Let's take a closer look at this loop in the `run()` method. See how it uses `repaint()` instead of `paint()` to paint the screen? The difference between `repaint()` and `paint()` is that `repaint()` calls `paint()`. `Repaint` is called whenever the image or window is changed or a new area is exposed. `Repaint()`

clears the previous image from the screen, then calls `paint()` to actually paint the image on the screen. Since we want to cycle through a series of images, we used `repaint()` to clear, then paint the screen.

We don't want our images to zoom by too fast, so we've added a `sleep()`, so there will be a slight pause each time an image is displayed. You've probably also noticed that `sleep()` is enclosed in a try statement, because it's possible for an error to occur while `sleep()` is processing. In Java, the correct terminology is "The `sleep` method can throw the exception `InterruptedException`." The try statement says that Java should try to sleep, but if it's interrupted before the `sleep()` completes, it should do whatever is specified in the catch statement. The Java compiler tries to protect processes from stumbling into errors that programmers didn't consider. If a method throws an exception, the Java compiler requires you either to catch the exception (using the try statement) or to pass it up to the caller (using the throw clause). Notice that our catch statement does nothing. That's okay. At least we made the conscious decision not to do any error handling. In later examples, we'll handle exceptions.

PAINTING THE PICTURE

Let's move on to the `paint()` method, which will be called by `repaint()` from our loop. The `paint()` method draws the "Hello World!" string, then draws the current frame. These frames were loaded when the applet was initialized by the `init()` method. The `init()` method uses `getImage()` to load images based on a URL. We used the `getDocumentBase()` to tell the browser to look for the images in a sub-directory under the directory where it found the `hello.html` file.

Notice that we've added a `start()` and `stop()` method, so that when users go to the `hello.html` page, the animation thread will be started. When they leave the `hello.html` page, however, our thread won't waste cycles by displaying images that no one will see. If the user comes back to our `hello.html` page, the browser will restart our animation thread.

When you finish updating the `helloWorld.java` file, compile it (`javac helloWorld.java`). Use the browser to open the `hello.html` file, but don't forget to stop and restart the browser each time you update the applet. Our tiny applet now has multithreaded animation buzz!

But wait a minute here. Why does the little man (Duke) sometimes paint so slowly? And why does the "Hello World!" message seem to flicker? Remember, the `repaint()` method clears the screen

What's So Hot About Java?

```
import java.awt.*;
import java.applet.*;

public class helloWorld extends Applet implements Runnable {
    Thread animationThread=null;
    protected int cur = 0, delay=150;
    protected Image frame[] = new Image[10];

    public void init() {
        resize (150,150);
        for (int i = 0; i<10; i++)
            frame[i] = getImage(getDocumentBase(),
                                "images/T"+i+".gif");
    }

    public void start() {
        animationThread = new Thread (this);
        animationThread.start ();
    }

    public void run () {
        while (true) {
            for (cur=1; cur<10; cur++) {
                repaint();
                try { // slow down the image changes
                    Thread.sleep(delay);
                } catch (InterruptedException e) {}
            }
            cur=0;
            repaint();
        }
    }

1 public void paint (Graphics g) {
    g.drawString(new String("Hello World!"), 70,50);
    g.drawImage(frame[cur], 25, 55, this);
}

    public void stop() {
        if (animationThread != null &&
            animationThread.isAlive())
            animationThread.stop();
    }
}
```

Figure 4. Modified Version of helloWorld Implementing the Runnable Interface

What's So Hot About Java?

and then paints the image, so each pixel used to display the “Hello World!” message is set to grey (the background color), then to black, each time the image is repainted. We perceive this set and reset as a flicker.

FIVE WAYS TO REDUCE FLICKER

You can do five things to smooth your animations:

- Avoid complex calculations in `paint()`
- Clear only what is necessary
- Minimize the paint area
- Make sure images are loaded before calling `paint()`
- Double-buffer images

We'll cover these in order. First, it's important to realize that `paint()` will be called frequently. Its only responsibility should be to draw the image. Because our `helloWorld` applet is so simple, it was easy to limit our `paint()` to simply drawing the image. As your applets become more complex, **try to precalculate as much as possible**. For example, PC game programmers often use lookup tables, instead of calculating trigonometric functions, for exactly this reason.

Next, **clear only what is necessary** to help minimize the paint area. In our applet, there's really no reason to clear the area where the “Hello World” message is painted, as the message never changes. Since the `repaint()` method calls the `update()` method to clear and then `paint()` the screen, we can clear and paint only specific areas by overriding the `update()` method. To test this, try inserting the `update()` method (Figure 5) into our new `helloWorld` class (the big “1” in Figure 4). Re-compile and run. Notice that although Duke still flickers, the “Hello World!” message seems pretty steady. That's because, instead of clearing the entire image, we clear and paint only the part where Duke is drawn. To **minimize the paint area** even more, we'd need to use *clipping*. Clipping is beyond the

```
public void update (Graphics g) {  
    g.setColor (getBackground());  
    g.fillRect (25,55,60,80);  
    g.setColor (getForeground());  
    paint (g);  
}
```

Figure 5. `update()` Message to Insert into the New `helloWorld` Class

scope of this article, but is explained in Lemay and Perkins' *Teach Yourself Java in 21 Days*.

The fourth method, **making sure images are loaded before calling** `paint()`, is really noticed only at the start of the animation. This is because the `getImage()` method called in `init()` returns immediately; image loading takes place in the background. In other words, the image may not be loaded when `paint()` is called. Java loads images asynchronously, so if an image is not actually needed until later, the user won't be unnecessarily delayed.

If you noticed the images painting slowly the first time, or if it looked like the motion was jerky at first (because some images were being skipped), you can use Java's `MediaTracker` to ensure images are loaded before they are painted. This can cause a delay before the animation is started, but once the animation does start, it'll be smoother. To test this, try replacing the `init()` method in Figure 4 with the `init()` method in Figure 6. Be sure to recompile `helloWorld.java` and restart the browser to invoke the new version. You probably noticed the delay before the animation started, so bear in mind the tradeoff associated with pre-loading images. Note that the wait doesn't have to be in `init()`. If you choose to use `MediaTracker`, you can minimize delays by performing the wait directly before the animation is started.

```
public void init() {  
    int i;  
    resize(150,150);  
    MediaTracker tracker = new MediaTracker(this);  
    for (i=0; i<=10; i++) {  
        frame[i] = getImage(getDocumentBase(),  
                             "images/T" + i + ".gif");  
        tracker.addImage(frame[i],i);  
    }  
    showStatus("loading images");  
    try {  
        tracker.waitForAll();  
    } catch (InterruptedException e) {}  
    showStatus("Images loaded");  
}
```

Figure 6. Alternate `init()` Method

Finally, **double-buffering provides the best alternative for reducing flicker**. Double-buffering takes advantage of the fact that it's faster to display

What's So Hot About Java?

a frame than to draw an image on the screen. Instead of drawing an image on the screen, the application draws the image to a frame that is not being displayed. Once the image is drawn, the application simply displays the frame where the image was drawn. To test this, make the following changes in `helloWorld.java`:

1. Replace the `paint()` and `update()` methods with those provided in Figure 7.1.
2. Replace the line “`public void init() {}`” with the lines provided in Figure 7.2.

```
public void update(Graphics g) {
    paint(g);
}
public void paint(Graphics g) {
    bufGraphic.setColor(getBackground());
    bufGraphic.fillRect(0,0,150,150);
    bufGraphic.setColor(getForeground());
    bufGraphic.drawString(new String ("Hello
        World!"), 70,50);
    bufGraphic.drawImage(frame[cur], 25, 55,
        this);
    g.drawImage(bufImage, 0,0, this);
}
```

Figure 7.1. First Set of Changes to `helloWorld.java` for Double-Buffering

```
protected Image bufImage;
protected Image Graphics bufGraphics;

public void init() {
    bufImage = createImage(150,150);
    bufGraphic = bufImage.getGraphics();
}
```

Figure 7.2. Second Set of Changes to `helloWorld.java` for Double-Buffering

You probably noticed that the `update()` method in Figure 7.1 doesn't clear the screen anymore. Clearing the screen is no longer required, because we are replacing the full frame. This and the number of lines in the `paint()` method seem to contradict our previous statements about reducing the paint area and making the `paint()` method as simple as possible. Although this method does require more work (*i.e.*, consumes more memory and cycles), the extra work is not visible to the user.

PANELS, WIDGETS, EVENTS

We could certainly do a lot more with animations, but many other neat Java features await us, so let's move on to a new applet; we'll call it `whiteBoard`.

Before starting on the `whiteBoard` applet, we need to discuss Java's Abstract Window Toolkit in more detail. In our `helloWorld` applet, we noted that because `Applet` was a subclass of `Panel` and `Component`, we could override inherited methods to *draw* on the browser window — as if the page being displayed by the browser window was our applet's drawing board. As your applets become more sophisticated, you'll need more control over the layout of this “drawing board.” You may already be aware that you can divide a Web page using HTML table and frame commands. Additionally, Java provides a Layout Manager so your applet can easily add panels, containers, and components within this frame or table entry. These panels and components can even be nested within one another. The browser communicates to these panels and components through *events*.

Our `whiteBoard` applet will start off looking very similar to the `DrawTest` demo provided with the JDK, which allows users to draw pictures on their browser windows. This new applet will also give us practice

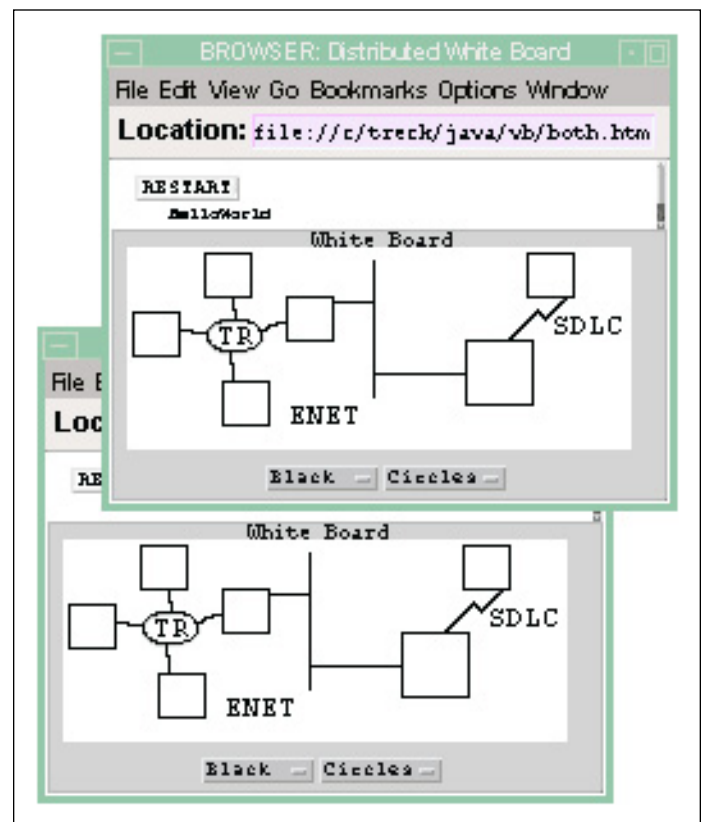


Figure 8. Final `WhiteBoard` Applet

What's So Hot About Java?

with panels and events. In later sections, we'll add code to demonstrate frames, inter-applet communications, and socket communications. The resulting applet (shown in Figure 8) will let two users connect to a server and draw on a distributed whiteBoard. In other words, as either user draws on the whiteBoard, updates will be displayed on both users' browsers.

Before we can start on our whiteBoard, we need to decide how the page will be formatted.

DECIDING ON PAGE FORMAT

Java's Layout Manager includes Border, Card, Flow, Grid, and GridBag layouts. You can view an example of each layout in the API documentation by selecting the `java.awt` package, then selecting the class for each layout. Our whiteBoard applet's main component is the drawing board itself, but we'll also need a heading and controls. The `BorderLayout` suits our needs pretty well.

We know the browser will send messages to our applet as *events*. Actions like pressing a key, pressing or releasing the mouse button, or dragging the mouse are all events. But pressing the mouse button on the whiteBoard itself should be handled differently from pressing the mouse button while in our controls area. We can separate these areas and events by making the drawing board one object and the controls a separate object. We'll refer to these as the whiteBoard Panel (`wbPanel`) and the Controls Panel (`wbControls`). You'll notice in Figure 8 that our resulting applet uses the `FlowLayout` for the Controls Panel, so we've nested a `FlowLayout` within a `BorderLayout`.

Now, let's consider our `wbPanel` in more detail. We'll let users draw shapes with lines and circles, and we'll also let them enter text, but we'll start by just allowing lines. We'll need, however, to keep track of what is on the board, its location, and its color. We can simplify things by creating a `Shape` class. Because we don't know how many Shapes will be drawn, we'll use Java's `Vector` class to create a dynamic array of Shapes.

In Figure 9, you'll see the `Shape` and the `wbPanel` classes. The `Shape` class includes a method named `Shape`. A method that has the same name as its class is called a *constructor*. Classes are to objects as blueprints are to houses. Constructors are methods that create an object from a class. An applet invokes a constructor by preceding the class name with the `new` operator. A constructor method initializes the new object and its variables and, in general, does anything the object needs to do to initialize itself. In the case of `Shape`, the constructor records the coordinates and color of the `Shape`. The keyword `this` simply

```
import java.awt.*;
import java.util.Vector;

class Shape extends Object {
    public int x1, y1, x2, y2;
    public Color c;

    public Shape(int x1, int y1, int x2,
                int y2, Color c) {
        this.x1 = x1;
        this.y1 = y1;
        this.x2 = x2;
        this.y2 = y2;
        this.c = c;
    }
}

class wbPanel extends Panel {
    int x1,y1, x2,y2, x1,y1;
1 public static Vector shapes;

    public wbPanel() {
        setBackground(Color.white);
        setFont(new Font("TimesRoman",
                          Font.BOLD,16));
2 shapes = new Vector();
    }
    public boolean handleEvent(Event e) {
        switch (e.id) {
            case Event.MOUSE_DOWN:
                x1 = e.x;
                y1 = e.y;
                x2 = -1;
                return true;
            case Event.MOUSE_UP:
                Shape s = new Shape(x1,y1,e.x,
                                    e.y,getForeground());
3 shapes.addElement(s);
                x2 = x1 = -1;
                repaint();
                return true;
            default
                return false;
        }
    }
    public void paint(Graphics g) {
4 int nShapes = shapes.size();
        g.setColor(getForeground());
        g.setPaintMode();
        for (int i=0; i < nShapes; i++) {
5 Shape p = (Shape)shapes.elementAt(i);
            g.setColor((Color)p.c);
            g.drawLine(p.x1, p.y1, p.x2, p.y2);
        }
    }
}
```

Figure 9. Shape and wbPanel Classes in the wb.java File

What's So Hot About Java?

```
import java.awt.*;
import java.applet.*;
import wbPanel;

public class whiteBoard extends Applet {
    Label wbTitle = new Label ("White Board",
                               Label.CENTER);

    public void init() {
        setLayout(new BorderLayout());
        wbPanel wp = new wbPanel();
        wbTitle.setFont(new Font("TimesRoman",
                                Font.BOLD,18));
        add ("North", wbTitle);
        add ("Center", wp);

1      add("South",new wbControls(wp));
    }
}

class wbControls extends Panel {
    wbPanel target;

    public wbControls(wbPanel target) {
        this.target = target;
        Choice ink = new Choice();
        ink.addItem("Black");
        ink.addItem("Blue");
        ink.addItem("Red");
        add(ink);
    }

    public boolean action(Event e, Object arg) {
        if (e.target instanceof Choice) {
            String choice = (String)arg;
            if (choice.equals("Red")) {
                target.setForeground(Color.red);
            } else if (choice.equals("Blue")) {
                target.setForeground(Color.blue);
            } else if (choice.equals("Black")) {
                target.setForeground(Color.black);
            }
        }
        return true;
    }
}
```

Figure 10. The whiteBoard.java File, Including the whiteBoard and wbControls Classes

refers to the object within which the `this` is used. For example, the `Shape` constructor is setting the coordinates and color of *this* instance of `Shape` to the values passed when its constructor is called.

Figure 11 illustrates the new `wb` class. When `whiteBoard` creates its `wbPanel`, it will call the `setWb()` method. Since `wb` represents the contents of the `whiteBoard`, we moved our vector of shapes into the abstract `wb` class. Notice the `synchronized` modifier used with each of the `wb` class's methods. The `synchronized` keyword prevents more than one thread from executing the enclosed block of code at the same time. Insert the `wb` class in Figure 11 into `wb.java` after the import statements.

```
public abstract class wb {
    private static wbPanel whiteBoard;
    public static Vector shapes;

    public static synchronized void setWb
        (wbPanel wp) {
        shapes = new Vector ();
        whiteBoard = wp;
    }

    public static synchronized void clearWb () {
        shapes = new Vector ();
        whiteBoard.repaint();
    }

    public static synchronized void
        addShape(Shape s) {
        shape.addElement(s);
    }
}
```

Figure 11. The New wb Class

You'll need to make a few other changes to utilize our new `wb` class. Refer to the original `wb.java` file in Figure 9. Notice the big green numbers "1" through "5." These numbers mark the lines that will be changed. Since we moved the shapes vector under the protection of `wb`, we need to remove its declaration from `wbPanel`, so in the `wb.java` file:

1. Delete the line at "1" and at "2" in Figure 9.
2. Replace the line at "3" with the statement `"wb.addShape(s);"` ;
3. Prefix the word "shapes" with "wb." in the line at "4" and at "5."

In the `whiteBoard.java` file (Figure 10), add the statement `"wb.setWb(wp);"` at the line indicated by the big "1." This will initialize the `whiteBoard`. In the `helloWorld.java` file, you can:

What's So Hot About Java?

1. Insert the line “add(new Button(“RESTART”));” after the call to `resize` in `helloWorld`'s `init()` method.
2. Insert the `handleEvent()` method illustrated in Figure 12 into the `helloWorld` class after the `stop()` method. `HandleEvent()` will restart the animation and clear the `whiteBoard` (via inter-applet communications) when “RESTART” is clicked.

```
public boolean handleEvent (Event evt) {
    switch(evt.id) {
        case Event.ACTION_EVENT:
            if ((“RESTART”).equals(evt.arg)) {
                if (animationThread.isAlive())
                    animationThread.stop();
                animationThread = new Thread(this);
                animationThread.start();
                wb.clearWb();
                return true;
            } else return super.handleEvent(evt);
        default:
            return super.handleEvent(evt);
    }
}
```

Figure 12. The `handleEvent` Method

Now look at the `wbPanel` class. Notice that it also has a constructor. In addition, it has a method we haven't seen before, the `action()` method. The `action()` method is part of the `Component` class, and it's called whenever any action occurs within the component. Our `whiteBoard` will let users draw lines by pressing the mouse button (`MOUSE_DOWN`) at the starting point, dragging the mouse to the end point, then releasing the mouse button (`MOUSE_UP`). Each time `repaint()` is called for our `wbPanel`, the `update()` method will refresh the background, and `paint()` will redraw each shape.

Figure 10 shows the `whiteBoard.java` file, which includes the `whiteBoard` and `wbControls` classes. Notice that the `whiteBoard` class itself is very simple. In fact, it includes only an `init()` method, which sets the layout, then creates and adds the heading, `wbPanel`, and `wbControls` components.

The `wbControls` component is slightly more involved, because it needs to allow user input. Notice that by adding a `Choice` button to our `wbControls` panel, our `action()` method doesn't have to deal with `MOUSE_DOWN` events. Instead, the event is

passed to the `action()` method as a `Choice`, with an argument specifying the user's selection.

If you haven't already done so, create the `wb.java` (Figure 9) and `whiteBoard.java` (Figure 10) files, and then compile them with `javac`. You'll need an HTML file for the browser to open. Simply create a copy of the `hello.html` file created earlier and call it `wboard.html` (`cp hello.html wboard.html`). In `wboard.html`, replace `helloWorld` with `whiteBoard` and delete the height and width fields. To test, set the location field of your browser to the `wboard.html` file.

INTER-APPLET COMMUNICATIONS

To implement the `whiteBoard` applet, our `wbControls` object needed to communicate changes in ink color to the `wbPanel` object. This communication between panels was done simply by having `wbControls` call `wbPanel`'s `setForeground()` method. Painless, right? The good news is that inter-applet communications is just as easy. Since we now have two applets, `helloWorld` and `whiteBoard`, we can demonstrate Inter-Applet communications.

Here's the plan. We'll display both the `helloWorld` and `whiteBoard` applets on a single Web page. We'll add a `RESTART` button to our `helloWorld` applet to restart the animation and clear the `whiteBoard`'s drawings. In other words, when the `RESTART` button is pressed, the `helloWorld` applet will have to tell the `whiteBoard` applet to clear its `wbPanel`.

To make our plan work, we need something to represent our `whiteBoard`'s contents. We need a few methods that both `helloWorld` and `wbPanel` can call to manipulate the `whiteBoard`'s contents. The problem is, we don't want to try to force these two classes into the same superclass. We solved a similar problem earlier by using Interfaces. Interfaces are a set of abstract methods that act as a common shared repository. If we use an abstract class to represent

```
<HTML>
<HEAD>
<TITLE>Distributed White Board</TITLE>
</HEAD>
<FRAMESET ROWS="30%, 70%">
<FRAME SRC="hello.html" NAME="hello">
<FRAME SRC="wboard.html" NAME="wboard">
</FRAMESET>
</HTML>
```

Figure 13. The `both.html` File

What's So Hot About Java?

our whiteBoard's contents, then both whiteBoard and helloWorld will have access. This means neither applet needs to change, except to invoke our new methods.

Now, recompile all three files: `wb.java`, `whiteBoard.java`, and `helloWorld.java`.

We're almost there. Since we need to combine the helloWorld and whiteBoard applets on a single page, we need a new HTML file. Create the `both.html` file as shown in Figure 13. Open this HTML file in your browser, draw a few lines on the whiteBoard, then test the new RESTART button.

The next segment will feature client-server network (socket) communications.

THE PLAN

Is adding network communications as simple as adding inter-applet communications? Well, not exactly. If the applets happen to be running on different systems, we can't just invoke the method of another applet.

Because Java's security features restrict applets from making network connections (except to the host that served the applet), we need to implement a whiteBoard server program to forward messages to and from client whiteBoard applets. Unlike our helloWorld and whiteBoard applets, our server won't be invoked from a browser. Instead, the server should run as a daemon on the Web Server so it will always be available for connecting clients. Note that Java programs not invoked from a browser are called Java applications.

Java applications must have a `main()` method, which is called when the program is invoked. Any command line arguments supplied when the program is started are passed as input arguments to the `main()` method, just as in C and C++.

Just as an applet has a specific flow, a pair of programs using socket interfaces also have a specific flow. Figure 14 depicts a simple flow in which a server listens for and connects to incoming clients. A server associates itself with a particular logical port by *binding* itself to that port. The server waits to *accept* incoming clients. To communicate with a server, a client *connects* to the port to which the server is bound. For each incoming client, TCP/IP creates a new socket to represent the client. This new socket is passed to the server that is bound to the requested port to handle all communication to this particular client.

The process is repeated for each incoming client. Note that server processes block while waiting

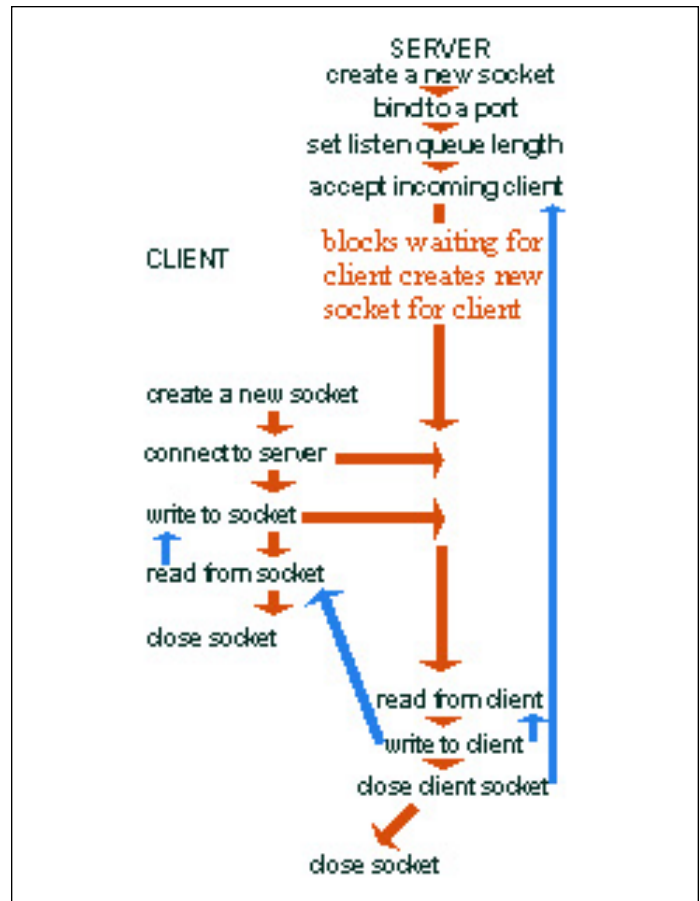


Figure 14. A Simple Flow, in Which a Server Listens for and Connects to Incoming Clients

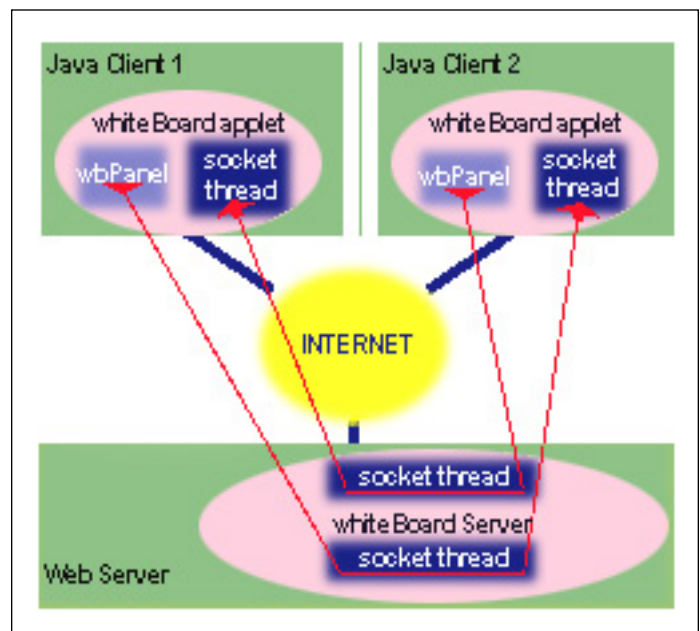


Figure 15. How Clients Interface to Each Other Through the Server

What's So Hot About Java?

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

public class wbServer {
    static ServerSocket listener;
    static Socket c1, c2;

    public static void main(String args[]) {
        DataInputStream in;
        DataOutputStream out;
        int port = 8888;

1      try {
            listener = new ServerSocket(port);
        } catch (IOException e) {
            System.out.println("ServerSocket
            failed"); return;
        }
        while (true) {
            try { // get a pair of client
                c1 = listener.accept();
                c2 = listener.accept();
            } catch (IOException e) {
                System.out.println
                ("Accept failed"); return;
            }
            serverSockThread t1 = new
            serverSockThread(c1,c2); t1.start();
            serverSockThread t2 = new
            serverSockThread(c1,c2); t2.start();
        }
    }
}
```

Figure 16. Source Code to Be Inserted into the wbServer.java File

for incoming clients. And when reading messages from the socket, both the server and the client processes block.

Because the socket interface will cause our client and server to block at various points, we'll need to implement multithreading in both the client and server. Figure 15 illustrates how the clients will interface to each other through the server.

You'll need to do the following:

- Create a whiteBoard Server (call it wbServer). Figure 16 contains the source code to be inserted into our wbServer.java file. Notice that the server simply accepts two clients, then creates two threads. Each thread will listen for messages from one client and forward those messages to the other client.

```
import java.awt.Color.*;
import java.io.*;
import java.net.Socket;

public class serverSockThread extends Thread {
    DataInputStream in;
    DataOutputStream out;
    Socket sIn, sOut;

    serverSockThread(Socket In, Socket Out) {
        sIn = In;
        sOut = Out;
        try {
            in = new DataInputStream
            (In.getInputStream());
            out = new DataOutputStream
            (Out.getOutputStream());
        } catch (IOException e) {
            System.out.println
            (Socket Stream error"); return;
        }
    }
    public void cleanupSock() {
        try {
            sIn.close();
        } catch (IOException e) {
        } finally {
            try {
                sOut.close();
            } catch (IOException e) {
            } finally {
                stop();
            }
        }
    }
    public void run(); {
        int i;
        while (true) {
            try {
                i = in.readInt();
                out.writeInt(i);
            } catch (Throwable e) {
                cleanupSock();
                return;
            }
        }
    }
}
```

Figure 17. Source Code to Be Inserted into the ServerSockThread.java File

What's So Hot About Java?

- Create a `serverSocketThread` class to receive messages from a client and forward those messages to the other client. Figure 17 contains the source code to be inserted in the `serverSocketThread.java`.
- Create a `socketThread` class to read incoming messages from `wbServer`. These messages represent shapes to be drawn on the `wbPanel`, and therefore, these messages will be added to `wbPanel` using the `addShape()` method. Figure 18, includes the Java code for the `socketThread.java` source file.

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

public class socketThread extends Thread {
    wbPanel wp;
    DataInputStream in;
    socketThread(wbPanel wp, DataInputStream in) {
        this.wp = wp;
        this.in = in;
    }
    public void run() {
        int x1,y1,x2,y2,c;
        while (wb.sockAvail) {
            try {
                x1 = in.readInt();
                y1 = in.readInt();
                x2 = in.readInt();
                y2 = in.readInt();
                c = in.readInt();
            } catch (Throwable e) {
                wb.sockClose();
                return;
            }
            wb.addshape(new Shape(x1,y1,x2,y2,
                                new Color(c)),false);
            wb.repaint();
        }
    }
}
```

Figure 18. Java Code for the `socketThread.java` Source File

- Modify the `whiteBoard` applet to create a new thread (`socketThread`) to handle the socket when the applet is initialized and to stop the thread when the applet is destroyed. Figure 19 contains the new `whiteBoard` class to replace the `whiteBoard` class defined in `whiteBoard.java`.

- Modify the `wb` class so that when shapes are added to the `wbPanel`, they'll be added to both the local and the remote `wbPanel`. First, replace the `wb` class defined in `wb.java` with the new `wb` class in Figure 20. Next, modify the `wb.addShape(s)` in "`wb.java`" to "`wb.addShape(s,true)`".
- Ensure that no other applications are using the port 8888 by issuing the command "`netstat -an grep 8888`". If the port is being used, select another port number (find a number greater than 5000 that is not being used in the `/etc/services` file) and replace the 8888 used in `whiteBoard.java` and `wbServer.java`.

```
import java.net.*;

public class whiteBoard extends Applet {
    socketThread sockthread=null;
    Label wbTitle = new Label
        ("WhiteBoard",Label.CENTER);

    public void init() {
        Socket sock;
        setLayout(new BorderLayout());
        wbPanel wp = new wpPanel();
        wbTitle.setFont(new
            Font("TimesRoman",Font.BOLD,18));
        add("North", wbTitle);
        add("Center", wp);
        add("South",new wbControls(wp));
        try {
            sock = new
                Socket((getCodeBase()).
                    getHost(),8888);
        } catch (Throwable e) {
            return;
        }
        wb.setWb(wp,sock);
        sockthread = new socketThread(wp,wb.in);
        sockthread.start();
    }

    public void destroy() {
        wb.sockClose();
        if (sockthread != null &&
            sockthread.isAlive())
            sockthread.stop();
        System.exit(0);
    }
}
```

Figure 19. New `whiteBoard` Class to Replace the `whiteBoard` Class Defined in `whiteBoard.java`

What's So Hot About Java?

Note the import statements in Figures 19 and 20 are in addition to those import statements already included in `whiteBoard.java` and `wb.java`.

You may have noticed that the `serverSocketThread` uses the `DataInputStream` to filter the socket input. As described in the API documentation, if an error occurs on the socket or if the remote disconnects, the `readInt()` method will throw a `IOException` or `EOFException`, respectively. However, note that `serverSocketThread` catches only `Throwable`. `Throwable` is the superclass for both `IOException` and `EOFException`. By catching `Throwable`, we are telling Java to do the same thing, regardless of which exception is thrown.

You may have also noticed the `finally` keyword in `serverSocketThread`'s `cleanWb` method. `Finally` denotes statements to be executed regardless of whether an exception was generated.

When you complete the changes described, compile each of the modified Java source files. Start the `wbServer` by entering the command `java wbServer`. From a separate C window, start two Netscape browsers (`netscape &`) and set the `Location` field on both browsers to the `both.html` file created in the inter-applet communications example. Test your work by drawing lines on either of the whiteboards; lines drawn on either browser should appear on both browsers.

LINKING NATIVE METHODS

We've seen that Java provides a very powerful set of class libraries for building Internet applets and applications. However, it's possible that you may require a special capability provided by your operating system but not implemented in the Java class libraries. Java allows native methods to be implemented. Of course, there is a catch.

First, once you link a native method into your Java application, the application is no longer hardware-independent. In other words, you'll need to implement the feature for every operating system on which your application is to run.

The second problem is that, for security reasons, native methods cannot be linked into applets. So if you link native methods, your Java program will have to run as an application. In other words, it cannot be loaded from a browser over the Net.

The good news is that the JDK does provide tools to simplify linking native methods. For our example, assume that we've decided that our `wbServer` should not have its IP port (8888) hard-coded. Instead, we'd like to be more flexible and allow the system administrator to configure the port.

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

public abstract class wb {
    private static wbPanel whiteBoard;
    public static Vector shapes;
    public static Socket socket;
    public static DataInputStream in;
    public static DataOutputStream out;
    public static boolean sockAvail;

    public static synchronized void setW
        (wbPanel wp, Socket s) {
        shapes = new Vector();
        whiteBoard = wp;
        socket = s;
        sockAvail = true;
        try {
            in = new DataInputStream
                (socket.getInputStream());
            out = new DataOutputStream
                (socket.getOutputStream());
        } catch (IOException e) {
            return;
        }
    }

    public static synchronized void clearWb() {
        shapes = new Vector();
        whiteBoard.repaint();
    }

    public static synchronized void sockClose() {
        try {
            socket.close();
        } catch (IOException e) {
        } finally {
            sockAvail = false;
        }
    }

    public static synchronized void addShape
        (Shape s, boolean local) {
        shapes.addElement(s); // add shape to
        local whiteBoard

        if (local == true && sockAvail) {
            try { // send shape to partner
                whiteBoard
                    .writeInt(s.x1);
                    .writeInt(s.y1);
                    .writeInt(s.x2);
                    .writeInt(s.y2);
                    .writeInt(s.c.getRGB());
            } catch (IOException e) {
                wb.sockClose();
                return;
            }
        }
    }
}
```

Figure 20. New `wb` Class to Replace the `wb.java`

What's So Hot About Java?

The standard for configuring IP ports is to associate the port number with a service name in the `/etc/services` file. The server then calls the `getservbyname()` subroutine to convert the service name into a port number. Since the Java class libraries do not implement this feature, we'll need to create and link a native method.

Using this requirement as a premise, Figure 21 illustrates the steps required to create and link the proposed native method on AIX. Start by creating a `getService.java` file containing the `getService` class. Note the static initializer in the `getService` class. Static denotes the code in the braces (`{}`) is to be executed exactly once, when the class is loaded. The Java stub (`getservice.java`) class must include this static method to load the shared library containing the native method. Notice how `loadLibrary` loads `getsvc`, which matches the library name (`libgetsvc.so`) in our makefile (Figure 22). The `getService.java` file also contains the declaration for our native `byName()` method.

In the second step, generate the `getService` class file by compiling the `getService.java` file. We use this class file in Steps 2 and 3 (see Figure 2) to generate the C header file and the C stub file.

In Step 4, we modify our `wbServer` to create the `getService` object and call the `byName()` method we declared in Step 2. This code replaces the line denoted by the big green "1" in the `main()` method of our `wbServer` class.

Now we've completed the steps that are generic to linking native methods in any Java implementation. The remaining steps are AIX-specific. We'll discuss those next.

AIX-SPECIFIC STEPS

In Step 5, we create an export file, `getService.exp`. The export file is required by the AIX linker (`ld`) command to indicate external symbols to be made available for another executable (Java) to import. Note that we exported the two symbols defined in the C stub file generated by `javah`.

Finally, in Step 6, we create the native method itself. Notice several key points about this function:

1. To match the stub file `getService.h` created by `javah` in Step 3, function name must be the format `class_ method`. Our function name, then, is `getService_byName`.
2. Be sure to include `#include "getService.h"`, the C header file generated by `javah`.

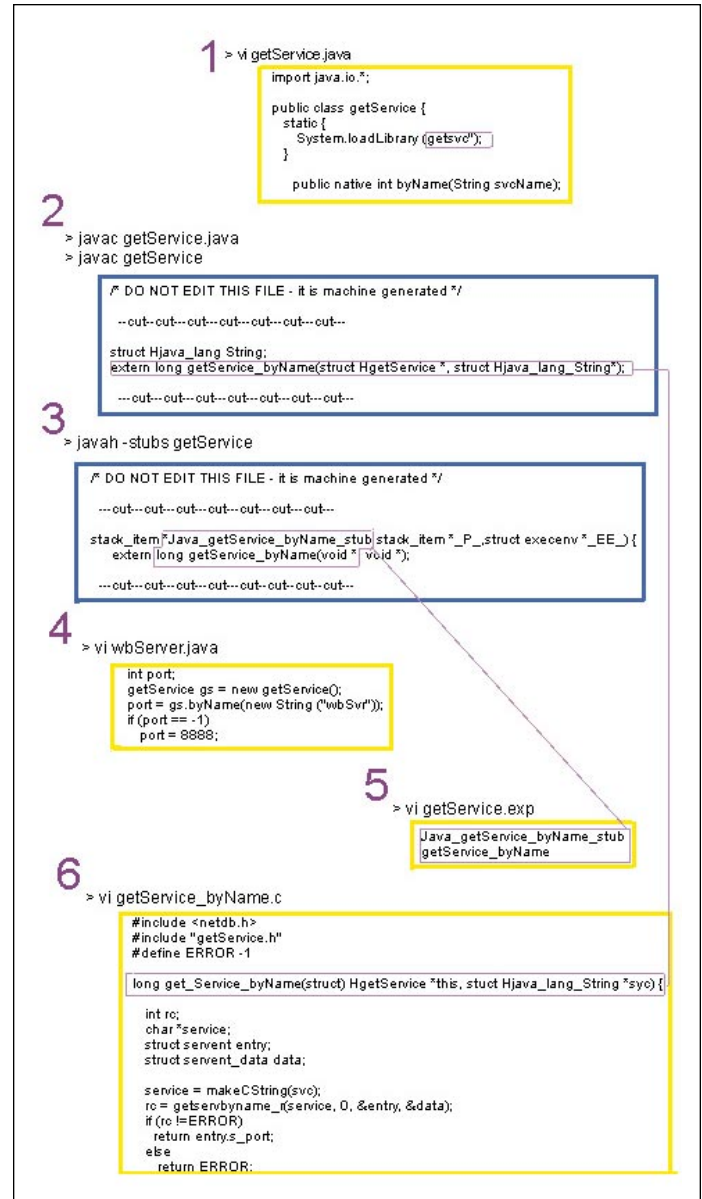


Figure 21. Steps for Creating and Linking a Native Method on AIX

3. The input arguments and returned parameter must be defined to match the declaration in the C header file `getService.h` generated by `javah`.
4. The `makeCString()` function was used to get a character pointer to the `String` input argument passed by the Java class. The `makeCString` and other useful methods for mapping Java constructs to C are declared in the `/usr/lpp/Java/include/native.h` file. The `native.h` file is included in the `getService.h` file generated by `javah`.

What's So Hot About Java?

Now that we've written all our C and Java source code, our only remaining step is to compile and link the code. Figure 22 provides a contents of a makefile that simplifies this requirement. Note that we've set up several variables so that our makefile can be easily modified to compile and link other native methods.

If you have not already done so, enter the text from Figure 22 into a file called `makefile`. Once this is complete, enter the command `make`, to build our new `wbServer`. Finally, insert the line "`wbSvr 8888/tcp`" into the `/etc/services` file. Test the `wbServer` and `whiteBoard` clients as described in the "Network Communications" posted earlier.

This concludes our series on Java examples. So where do you go from here? Well, the final copy of `whiteBoard` provided with the download package shows images as they are being drawn and allows users to draw circles and text as well as lines, so incorporate these enhancements into your applets. Try dressing up the `helloWorld` message by using the `Blink` demo provided with the `JDK`. Be sure to visit the `IBM Java Home Page` at <http://ncc.hursley.ibm.com/javainfo> to see what's in store for `JDK V1.1`. And remember — this is only the beginning!

```
JAVA_HOME = /user/lpp/Java
NATIVE = getService_byName
STUB = getService
CFLAGS = -I$(JAVA_HOME)/include/aix_pt
        -I$(JAVA_HOME)/include
LIB = libgetsvc.so
IMP = -bI:/usr/lpp/Java/include/java.exp
EXP = -bE:getService.exp
LDFLAGS = -bM:SRE - bnoentry -lc_r
        /usr/lib/libc.a

$(LIB): $(STUB).o $(NATIVE).o
    ld $(IMP) $(EXP) $(LDFLAGS) -o $(LIB)
    $(STUB.o $(NATIVE).o
    mv $(LIB) $(JAVA_HOME)/lib/aix_pt

$(STUB).o: $(STUB).c $(STUB).class
    xlc_r $(CFLAGS) -c $(STUB).c

$(NATIVE).o: $(NATIVE).c $(STUB).class
    xlc_r $(CFLAGS) -c $(NATIVE).c

$(STUB).h: $(STUB).class
    javah $(STUB)

$(STUB).c: $(STUB).class
    javah -stubs $(STUB)

$(STUB).class: $(STUB).java
    javac $(STUB).java
    javah $(STUB)

clean:
    rm $(JAVA_HOME)/lib/aix_pt/$(LIB)
    $(STUB).class $(STUB).h \
    $(STUB).c $(STUB).o $(NATIVE).o $(LIB)
```

Figure 22. Contents of a makefile