Internet Programming with Java and JavaScript

OBJECTIVE:

The objective of this lab is to give the student hands–on experience with the Java and JavaScript. Each exercise begins with a statement of the exercise objective followed by a description of the environment and concepts. Commands required to complete the exercise are provided in a box following the description.

The User ID for all exercises in this lab is **netlab** (the password is **netlab**).

OUTLINE:

1. Java Basics: Demos and Documentation
2. Create a simple Java Applet.
3. Create a simple JavaScript script
4. Interfacing Java and JavaScript
5. Advanced Java Animation
6. Java Panels, Widgets and Events
7. Java Communications (Inter–Applet and Sockets)

FYI

The lab system is configured with software versions that are not currently shipped as part of AIX V4.2. But the software is generally available. Here's what you'll need:

AIX V4.1.3 (or later) with X–Windows installed

Java Development Toolkit (JDK) Version 1.0.2

(download from http://ncc.hursley.ibm.com/javainfo)

Netscape Navigator V3

(download from http://www.netscape.com)

Java Lab

1. Java Basics: Demos and Documentation

1.1 Run the JDK demos.

The demo code shipped with the JDK provides a quick way to ensure everything is properly installed on your system. The code is easy to read (and learn from) and, because of Java's object–oriented features, can be reused from your own Java programs.

The JDK executables are stored in **`/usr/lpp/Java/bin`**, so make this part of your path. You will also need to export your **`DISPLAY`** and compile the demos before executing them. The **`/usr/lpp/Java/demo`** directory includes scripts to compile and run the demos.

From a window, enter the following commands:

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

Replace mydisplay ***with the hostname for your system***.

Notice the 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 Netscape browser. Note the appletviewer has Menu button. Use this button to Quit an applet so the next demo will be started. If you get tired of seeing the demos, you can just press **`<Ctrl> c`** from the window where you entered the **`RunDemos`** command.

The first applet, Animator, simply displays a series of images in succession to create the effect of movement. Most Web animations either use this method, display a single image at different coordinates (creating the effect of movement) or successively enlarge or reduce a single image (creating the effect of growth or explosions).

To get a better idea of how it works, view the HTML and an image used by the Animator.

```
> cd /usr/lpp/Java/demo/Animator
```
View the HTML file used to call the Animator applet
```
> pg example1.html
```
Note this HTML file indicates the image source directory is 'images/Duke'
```
> ls images/Duke
```
Note there are 10 .gif files in this directory
```
>  export LANG=C
>  netscape &
```
From the Netscape Window, press the Open button and then enter:
```
  file:/usr/lpp/Java/demo/Animator/images/Duke/T10.gif
```
To close the image, click on the window and then press q

1.2 Use the Java documentation to get familiar with Java applet flow.

We're almost ready to start our own applet. But first, we need a few concepts. Figure 1 shows the steps for creating an applet (take a quick look at Figure 1, but don't start on the steps yet). In Figure 1, look at the **`helloWorld.java`** file. All Java source code is stored in files whose name ends in **`.java`**. (The Java compiler writes to files ending in **`.class`**.)

A key feature of Object Technology (OT) is reusability. Even the simplest applets take advantage of this. Notice the line creating the helloWorld class **`extends java.applet.Applet`**. Extending this Applet class is what allows us to avoid rewriting the GUI interface (i.e. the browser). Since applets, by definition, are Java programs invoked from a browser, all applets extend the Applet class. By extending the Applet class, applets need only implement those methods and variables they want to customize.

But how do we know what methods and variables are available in a class (such as Applet) for us to extend? The best way to answer this is to show it to you in the documentation. You can access the documentation online at http://java.sun.com/products/JDK/CurrentRelease/api. For this lab, you will access the documentation locally by clicking on the **Bookmarks** menu button of your browser and then selecting **Java Packages**.

From the Menu bar of the Netscape window, click on **Bookmarks** *and then select:* Java Packages

From the API documentation, notice the list of Java Packages. Almost everything in Java is either a *class*, an *object* or a *method*. Java groups *classes* into *packages*. We tell the compiler which class we are extending by giving the package and class name (i.e. '**`extends java.applet.Applet`**). So, to see what the Applet class provides, click on the **`java.applet`** link and then the **`Class Applet`** link. Notice the list of methods in the Applet class. Here's what you need to know:

- When the Web page containing your applet is referenced, the browser will call the **`init()`** and then the **`start()`** methods.
- If a 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()`** and then **`destroy()`**.

Feel free to click on any of these methods and read their description. In our helloWorld applet, we can override any of these methods simply by creating our own versions. For example, in **`init()`**, we would do things like sizing the window, setting background colors and formatting the layout. Whereas in **`start()`**, we'd want to paint the screen or activate an animation. Notice in Figure 1, the **`helloWorld.java`** contains an **`init()`** method, but no **`start()`**, **`stop()`** or **`destroy()`**. We didn't override these because the default actions were okay (helloWorld is a *very* simple applet).

However, **`helloWorld.java`** does have a **`paint()`** method. But **`paint()`** isn't listed as one of the methods for the Applet class. How does it get called? Well, just as our helloWorld class extends Applet, you'll notice (near the top of the screen for the Applet Class documentation), that Applet '**`extends Panel`**'. (Go ahead click on **`Panel`**.) And Panel '**`extends Container`**' , and (keep clicking) Container '**`extends Component`**'. So just as our helloWorld applet has access to all the public methods of Applet, Applet has access to all the public methods of Panel, Container and Component. Component includes a **`paint()`** method which is called by the browser anytime the window is updated. Input to **`paint()`** is a Graphics object representing the graphics area of this Component (our applet). So, for an applet, **`paint()`** is crucial.

If you're thinking...'this is alittle too abstract, maybe I should check the schedule for another session'...don't leave yet. We're just getting to the fun part, promise!

2. Create a simple Java Applet.

2.1 Write a simple Java applet to write a message to the browser window.

In this section we will start with an applet which simply writes a message to the browser window. Then we'll build on this by adding animation and multithreading.

Use Figure 1 and the following steps to build the simple helloWorld applet.

1. Create a **`java`** directory in your home directory. This is where you will store everything you create in this lab.
2. In your **`java`** directory, create the **`hello.html`** file to be opened by the browser.
3. Create the Java source file **`helloWorld.java`** as shown in Figure 1.
4. Compile **`helloWorld.java`** with the Java compiler (**`javac`**).
5. Set the **`Location`** in your browser to **`file:/u/netlab/java/hello.html`** and press **`Enter`** to test your work.

Now, we have a very simple Java and HTML example, and we understand the basics of applet flow and using the API documentation. Let's add a little buzz to our applet; let's add some animation.

2.2 Add animation to the helloWorld applet.

Since this is a Java lab (and not an art lab), we'll just steal the images for our animation from the JDK Animator demo. Use the following commands to copy the images to your **`java`** directory.

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

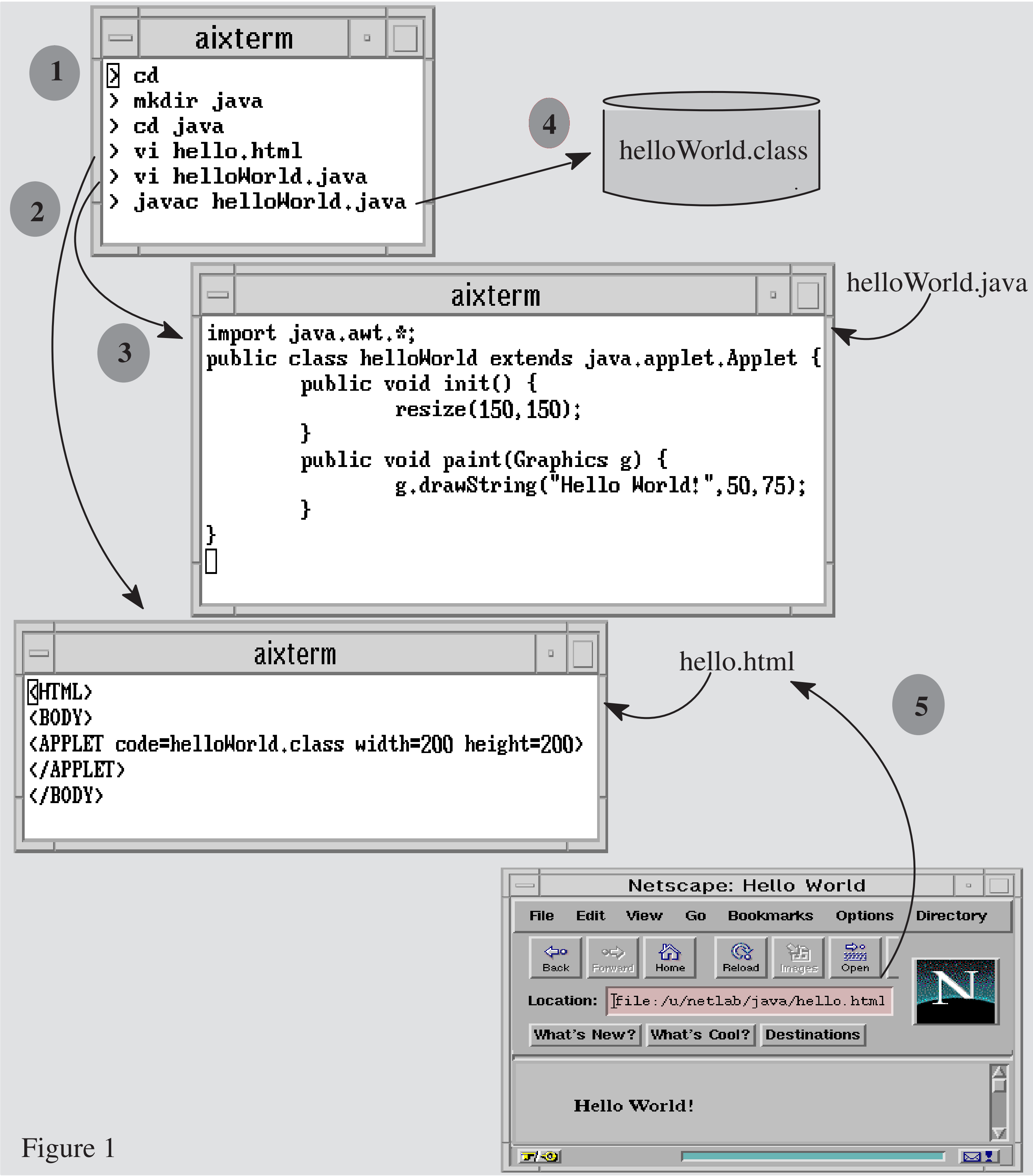

Figure 1

We could also steal the Animator class, but that wouldn't be as exciting as starting from scratch. Since we know animation is just displaying a series of images, we know we need a loop somewhere. But where? Not in **`init()`** because then we'd stay in **`init()`** forever. Putting it in **`paint()`** doesn't make sense either. What we really need is a way to spawn off our proposed loop, hmmm...sounds like a thread. So, to add animation, we really need to add multithreading.

The Thread Class is part of the **java.lang** package. Go back to the **Java Packages Bookmark**. This time, click on **java.lang** and then **Class Thread**. Like the Applet class, the Thread Class has **start()**, **stop()** and **destroy()** methods. It also has a **run()** method that is called by **start()**. Good. We can put our loop in the **run()** method.

The only problem is that Java does not allow multiple inheritance. So we have to pick, either we extend Thread or Applet, but not both. But, if we had 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 for behavior, just as a class is a template. The difference is that Interfaces pass *only* methods to their children. From the **Class Thread** page in the API documentation, notice the Thread Class '**implements Runnable**'. Click on **Runnable**. Note that the Runnable Interface implements only an abstract **run()** method. Okay. If helloWorld continues to extend Applet (so it keeps an interface to the browser) but also implements Runnable (so it adds the **run()** method), we have a place to put our loop. Perfect!

Figure 2 shows a modified version of **helloWorld.java** (for the remainder of this lab, all of the code in the figures is in the **/u/java/Figures** directory...so you can just copy it to your directory instead of typing it!). (Ignore the bold, commented numbers (1, 2, 3) in Figure 2, it's where we'll add stuff later.) Notice in Figure 2, that we've added a **run()** method, which has our loop. Also note, the **run()** method calls **repaint()** instead of **paint()**. The difference is that **paint()** simply paints the screen but, **repaint()** clears the screen and then calls **paint()**. Since we will be displaying a series of images, we want to clear, and then repaint, the screen. So, our applet *calls* **repaint()** and, to customize what is painted, our applet *overrides* **paint()**.

In the **run()** method's loop, we call **sleep()** so our images won't zoom by too fast. You've probably noticed the **sleep()** is enclosed in a **try** statement. This is because it is possible for a error to occur while the **sleep()** is processing. In Java, the correct terminology is 'The **sleep** method can *throw* the exception **InterruptedException**.' If a method can throw an exception, the Java compiler requires you either catch the exception (using the **try** statement), or throw it up to the caller (using the **throw** clause). Notice our **catch** statement does nothing. That's okay. At least we won't get a compiler error. In later examples, we'll handle our exceptions.

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 just as it did in our previous applet, but now, it also draws an image. The images are stored in the array, **frame**. These images 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 these images in the "waving" subdirectory under the directory where it found the **hello.html** file.

Finally, notice that we've added a **start()** and **stop()** method, so when users go to the **hello.html** page, the animation thread will be started. When they leave the **hello.html** page, however, our thread will stop (instead of wasting cycles by displaying

```
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];
/* 1 */
  public void init() {
    resize(150,150);
/* 2 */
    for (int i = 0; i<10; i++)
      frame[i] = getImage(getDocumentBase(),"waving/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();
    }
  }

/* 3 */
  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 2

images that no one will see). If the user comes back to our **hello.html** page, the browser will restart the animation.

Use the following commands to save your old helloWorld and build the new helloWorld applet.

```
> cd $HOME/java
> mv helloWorld.java helloWorld.textonly.java
> cp /u/java/Figures/Fig02.helloWorld.java helloWorld.java
> javac helloWorld.java
```

Stop and restart the Netscape Browser to have the new Applet loaded.
From the Browser set Location *to:*
file:/u/netlab/java/hello.html

At last! We've got 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 sometimes flicker? Remember, the **`repaint()`** method clears the screen and then paints the image. So each pixel used to display the "Hello World!" message is set to grey (the background color) and then to black, each time the image is repainted. We perceive this set and reset as a flicker. What appeared as jerky movements at the beginning was due the time it takes for the browser to load the images.

We'll learn more about animation, including how to reduce flicker, in the advanced Java sections, but first, it's time for a little JavaScript.

3. Create a simple JavaScript script.

3.1 Create scripts to write a message to the browser window.

In this section we will start with a script which simply writes a message to the browser window. Then we'll build on this by creating scripts to prompt for input, process forms and even control our Java Applet.

If you skipped Section 2, you will need to enter the following commands to build the base environment. This will provide you with a sample Java Applet (helloWorld) and HTML (**`hello.html`**) file.

```
> cd
> mkdir java
> cd java
> tar –xvf /u/java/base.for.3.tar
```

If you are new to Java and JavaScript, it is important to clarify that JavaScript is not just a 'script version' or 'simplified version' of Java. JavaScript and Java are alike in that they are both Object Oriented and in that they allow you to control browser behavior...but they address very different requirements. Java is a powerful programming language which provides classes for graphics, network communications and multithreading. JavaScript can control browser behavior and content, but does not have any graphics, networking or multithreading capabilities. Also unlike Java, JavaScript cannot run outside the context of a browser.

So let's take a look at what JavaScript (JS) can do. Figure 3 contains our first JS script. Note that it looks like a standard HTML file. That's because JavaScript is a scripting language (understood by the browser) which can be embedded in HTML files and even in URLs. The HTML tag **`<SCRIPT>`** is used to indicate the following statements (up to the **`</SCRIPT>`** tag) are written in the language JavaScript.

```
<HTML>
<BODY>
<SCRIPT LANGUAGE="JavaScript">
document.write("<H1>Hello World!</H1>");
</SCRIPT>
</BODY>
</HTML>
```

Figure 3

The **`document`** object references the document being displayed by the browser. In other words, the method **`document.write()`** is used to write HTML statements to the document the browser is to display; thus dynamically building a Web page. Simple enough, let's get started.

Create a file called **`simpleJS.html`** and enter the statements from Figure 3 into this file.

> \> cd $HOME/java
>
> \> vi simpleJS.html
>
> *Enter the statements from Figure 3 and then press* <Shift> ZZ *to exit the editor.*
>
> *If you haven't already done so, start the Netscape browser.*
>
> \> netscape &
>
> *Set the* Location *field of the Browser to:*
>
> file:/u/netlab/java/simpleJS.html

Hmmm. That was definitely less painful than the comparable Java applet from Section 2 (and I know it ran faster!). However, just like the Java applet in Section 2, it's pretty dull. Let's see what we can add.

3.2 Create a JavaScript script to prompt the user and collect input.

Have you ever seen Web pages that prompt you for a User ID or password? Well, the prompt itself can be done with standard HTML, but we need a program or script to process the input. If a CGI script is used, the data must be sent back to the server to process. But, if JavaScript is used, the input could be processed at the client. Let's give it a try.

Figure 4 contains a script called **`yourName.html`**. When the **`yourName.html`** page is referenced, the browser will display a window prompting the user for his name. Notice **`yourName.html`** contains 2 script sections. One in the **`HEAD`** section and one in the **`BODY`** section. The **HEAD** contains the prologue to the Web page, the **BODY** contains the content of the Web page. So, if we need to create JavaScript functions, these would go into the **HEAD** section. JavaScript in the **BODY** (as we saw in the **simpleJS.htm**l), builds the Web page content.

```
<HTML>
<HEAD>
<SCRIPT>
function sayHello(yourName)
{
  var now = new Date();
  document.write("<H1>Hello " + yourName + "!</H1>");
  document.write("<BODY><P>You've been staring at this since "
           + now + "</BODY>");
}
</SCRIPT>
</HEAD>
<BODY>
<SCRIPT>
  yourName = window.prompt("Enter Your Name:","");
  while (yourName == "" || yourName == null)
    {
    window.confirm("I'm SERIOUS!");
    yourName = window.prompt("Enter Your Name:","");
    }
  sayHello(yourName);
</SCRIPT>
</BODY>
</HTML>
```

Figure 4

The JavaScript **window** object is used to create or manipulate browser windows. In other words, **window.prompt** in the **BODY** section of **yourName.html** creates a window prompting the user for input, i.e., their name. **Window.confirm** creates a window requiring the user to answer a Yes/No question. In our **yourName.html** script, the confirmation window is displayed only if the user tries to get by without entering their name. Once our script has the user's name, it calls the **sayHello()** function in the **HEAD**, which calls the JavaScript **Date()** method and writes a message to the screen.

Create a file called **yourName.html** and enter the script from Figure 4. Then use the browser to test your work. Remember, all the code in the figures is in **/u/java/Figures**.

> cp /u/java/Figures/Fig04.yourName.html yourName.html
> vi yourName.html
> *Read through the script to ensure you understand it and then exit the editor.*
> *From the Netscape browser, set the* Location *field to:*
> file:/u/netlab/java/yourName.html

That was pretty simple, but if you are developing interactive web pages, you will probably want more than just 1 piece of information. You may even want to give the user a list of choices to select from, instead of a free–text field. All of this can be done with standard HTML. The HTML **`FORM`** tag allows you to build a form where a user can enter text, pick from a list or click buttons. This information could be processed locally (via JavaScript) or sent back to the server to be processed (via CGI scripts).

4. Interfacing Java and JavaScript.

4.1 Create a Web page with an HTML form.

By the time we finish this script, it will interface with the Java helloWorld applet written in section 2.2, but we'll keep it simple by taking it one step at a time.

We'll start by just looking at the syntax for a **`FORM`**. In the HTML file in Figure 5, the **`HEAD`** section has a **`SCRIPT`** section with nothing but a comment in it. That's ok, we get to this later. The **`BODY`** section has a heading (denoted by **`H1`**) and then a **`FORM`**. On this **`FORM`** there will be a list, from which the user can select either a "Waving Duke" or a "Tumbling Duke" before pressing the **`GO!`** button to submit the **`FORM`**. In HTML, the **`NAME`** field of the **`FORM`** allows you to refer to this form. The **`ACTION`** field provides the URL of the page to be displayed when the **`FORM`** is submitted.

To allow a user to select a choice from a list, the HTML **`SELECT`** tag is used. The **`SIZE`** indicates how many entries of the list will be displayed to the user when the page is displayed. Each choice on the list is denoted by the **`OPTION`** tag. Finally, the **`INPUT`** tag is used to define other input mechanisms such as buttons or free–text fields. Because our **`INPUT`** type is **`submit`**, it is a button which completes the **`FORM`** when pressed.

To test this simple **`FORM`**, enter the following commands:

> cp /u/java/Figures/Fig05.pickDuke.html pickDuke.html
> *From the Netscape browser, set the* Location *field to:*
> file:/u/netlab/java/pickDuke.html

When you open **`pickDuke.html`** in the browser, you should select one of the Dukes and then press the **`GO!`** button. You probably noticed that regardless of which Duke you pressed, the same (waving) Duke was displayed. That's because we're not finished.

```
<HTML>
<HEAD>
<SCRIPT>
// This is where we’ll put JavaScript functions later
</SCRIPT>
</HEAD>
<BODY>
<SCRIPT>
<CENTER>
<H1>Which Duke Would You Like to See?</H1>
<FORM NAME=”dForm” ACTION=”hello.html”>
<SELECT NAME=”dSelect” SIZE=1”>
<OPTION VALUE=”waver”> Waving Duke
<OPTION VALUE=”tumbler”> Tumbling Duke
</SELECT>
<INPUT TYPE=”submit” VALUE=”Go!”>
</FORM>
</CENTER>
</BODY>
</HTML>
```

Figure 5

This time, when you press the **`GO!`** button and Duke is displayed, look at the **Location** field of the browser. Notice the name ends with **`?dSelect=waver`** or **`?dSelect=tumbler`**, depending on which one you actually selected. That’s because the browser appends the **`FORM`** input info to the URL (delimited from the file name by a **`?`**) so it can be processed by a server–side script. For example, a CGI script would see this as a variable, **`WWW_dSelect`** and the value would be either “waver” or “tumbler”.

The ability to pass input from a **`FORM`** back to the server to, for example, perform database operations, is very useful. But there are also times when you would like to maintain variables which are accessible from the client–side. Netscape calls them *Cookies*. Cookies are just variables which are accessible for the duration of a browser session. So, for our example, instead of passing the input back to the server to process, we could save it in a Cookie and use it for client–side processing. Sounds like a nice place to store whether Duke should wave or tumble.

4.2 Modify the JS script to store and process Netscape Cookies (variables).

First, we need to change our Java Applet so Duke can tumble. We will add an input argument to be received from an HTML file, which specifies whether Duke should wave or tumble. We can get the ’tumbling’ images from the JDK demos. Use the following steps:

First, get the images

```
> cd $HOME/java
> mkdir tumbling
> cp /usr/lpp/Java/demo/TumblingDuke/images/tumble/* tumbling
> mv tumbling/T17.gif tumbling/T0.gif
```

Now, modify helloWorld.java to understand the new parameter

```
> vi helloWorld.java
```

*after the comment /*1*/, enter the line:*

```
protected String dir = "waving";
```

*after the comment /*2*/, enter the lines:*

```
String iTemp = getParameter("INDEX");
if  ( iTemp != null && Integer.valueOf(iTemp).intValue() == 1)
     dir = "tumbling";
```

Change the line with "getImage(..." to:

```
frame[i]=getImage(getDocumentBase(), dir+"/T"+i+".gif");
```

So the name of the images directory will be determined by the INDEX *parameter.*

Press <Shift> ZZ to exit the editor.

Compile the new version of helloWorld.java

```
javac helloWorld.java
```

Okay, now, if the HTML file which invokes the helloWorld Java Applet includes an **INDEX** parameter, the helloWorld will use it to display the correct Duke. So we have to change the HTML file to include an **INDEX** parameter. We were using the **hello.html** file to invoke the helloWorld applet. But it is static. Basically, we need to be able to build this file dynamically so we can enter the appropriate **INDEX** when the user selects it. Sounds like a job for JavaScript! Figure 6 shows an HTML file (**dukes.html**) with JavaScript to dynamically build the HTML required. Netscape Cookies can be accessed by calling **document.cookie()**. The only problem is that *ALL* the cookies are in the returned string. So our script uses **document.cookie.indexOf()** to find the **DINDEX** cookie, and then the **substring()** method to find the value of the **DINDEX** (the value will be after the string "DINDEX=", which has a length of 7; note, we are expecting only single digit indexes). The value is stored in the variable **index**, which is then used to build the **<PARAM NAME=INDEX VALUE=...>** tag in the HTML that is being generated with the **document.write()** method.

```
<HTML>
<TITLE> Duke </TITLE>
<BODY>
<SCRIPT LANGUAGE=”JavaScript”>

// Find the index in the Cookie
indexString = document.cookie.indexOf(”DINDEX”);
index = document.cookie.substring(indexString+7,indexString+8);

// Build the appropriate HTML
document.write(”<BODY><APPLET CODE=helloWorld.class “);
document.write(”WIDTH=200 HEIGHT=200>”);
document.write(”<PARAM NAME=INDEX VALUE=” + index + ”>”);
document.write(”</BODY>”);
</SCRIPT>
</BODY>
</HTML>
```

Figure 6

Enter the HTML in Figure 6 into a file called **`dukes.html`**

> \> cp /u/java/Figures/Fig06.dukes.html dukes.html
> \> vi dukes.html
> *Read through the script to ensure you understand it and then exit the editor.*

Now, all we have to do is to store the user’s input into a Cookie before going to this new **`dukes.html`** file. This is pretty simple with JavaScript. JavaScript adds an **`onClick`** field to the HTML **`INPUT`** tag. We can use the onClick field to store the current value of the **`SELECT`** portion of the **`FORM`**. We’ll also need to change the **`ACTION`** field for the **`FORM`** to point to our new **`dukes.html`** file. Use the following commands:

Modify the pickDuke.html file to store the input from the <FORM>

> vi pickDuke.html

Insert the following JavaScript in the <SCRIPT> section of the <HEAD>

```
function setDuke(index){
     document.cookie="DINDEX=" + index;
}
```

Change the ACTION field of the <FORM> to:

```
ACTION="dukes.html"
```

Change the <INPUT...> tag to:

```
<INPUT TYPE="submit" VALUE="GO!"
onClick="setDuke(dForm.dSelect.selectedIndex)">
```

Exit the editor.

Stop and restart the Netscape browser, then set the Location *field to:*

file:/u/netlab/java/pickDuke.html

When the user clicks on the **`GO!`** button, the browser will call our **`setDuke()`** function before going to the **`dukes.html`** page to store the input in the cookie, **`DINDEX`**. The **`dukes.html`** page will retreive the index from **`DINDEX`** and use it to set the **`INDEX`** parameter in the dynamically generated HTML. Our Java Applet, helloWorld, will get the value of the input parameter, **`INDEX`**, and use it to determine which directory to use for images. Impressive.

But maybe we need alittle tighter interaction between Java and JavaScript. Maybe we need to be able to send a message from our JavaScript script to our Java applet. Piece o' cake.

4.3 Modify the JS script to communicate with the Java applet.

A JavaScript script can communicate with a Java applet by calling any of the applet's public methods. Let's try it out. Remember the dynamic HTML built by **`dukes.html`**? How about if we add a button to stop the animation? Easy enough. We just add a **`FORM`** with a stop button. When the user clicks on the button, our script will send a message to the helloWorld applet by calling its **`stop()`** method. To do this, we will have to name our applet by adding a **`NAME`** field to the **`APPLET`** tag (we'll name our applet **`hApplet`**). Use the following to make these changes.

Modify the dukes.html file to include a button on the generated HTML

> vi dukes.html

First, on the APPLET tag before CODE=, insert the following NAME field :

```
NAME=hApplet
```

Then, directly before the statement `document.write(</BODY>);`, *insert:*

```
document.write("<FORM>");
document.write("<INPUT TYPE=\"button\" VALUE=\"Stop It!\"");
document.write("onClick=\"document.hApplet.stop();\">");
document.write("</FORM>);
```

Press <Shift> ZZ to exit the editor.

From the Netscape browser, set the Location *field to:*

file:/u/netlab/java/pickDuke.html

Now, when you click on the “Stop It!” button, our JavaScript script tells the Java applet to stop by calling it’s **`stop()`** method.

5. Advanced Java Animation

In Section 2 we built the simple, but animated helloWorld applet. The problem was, it flickered and movements were jerky. There are 5 basic ways to make your animations appear more fluid.

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

5.1 Minimizing the paint area.

As shown in Figure 2, our **`paint()`** method did not include any complex calculations, so the first bullet does not apply to our example. However, minimizing the paint area and clearing only what is necessary, does apply. For example, for each frame of the animation, the entire image was cleared, then “Hello World!” is re–drawn and the new frame is drawn. There is really no reason to re–draw “Hello World!” each time, as the message never changes. It is the **`repaint()`** method that calls **`update()`** to clear and then paint the screen. So, by overriding **`update()`**, we can clear and paint only specific areas. To test this, try inserting the **`update()`** method in Figure 7 into the **`helloWorld.java`** source file. The **`update()`** method should be placed after the **`/*3*/`** comment shown in Figure 2. Recompile **`helloWorld.java`** and invoke it from the browser.

In one window, view the **`update()`** *method from Figure 7*

> view /u/java/Figures/Fig07.update.insert

From a second window, edit the helloWorld.java file:

> vi helloWorld.java

Using your mouse, paste the **`update()`** *method into the helloWorld.java file following the* **`/*3*/`** *comment.*

Press <Esc> and then <Shift> ZZ in each window to exit the editors.

Recompile helloWorld.java

> javac helloWorld.java

Stop and then restart the Netscape browser, then set the Location *field to:*

file:/u/netlab/java/hello.html

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

Figure 7

Notice that although Duke still flickers, the "Hello World!" message seems pretty steady. To minimize the paint area even more, we'd need to use clipping, which is beyond the scope of this lab.

5.2 Make sure images are loaded before calling paint().

Going back to our list of ways to reduce flicker, the fourth bullet, ensuring images are loaded, is really only noticed at the start of the animation. This is because the **`getImage()`** method called in **`init()`** (see Figure 2) returns immediately; image loading takes place in the background. In other words, the image may not be loaded when **`paint()`** attempts to draw it. Java's Media Tracker can be used to ensure the images are loaded before the browser attempts to draw them. This can cause a delay before the animation is started, but once the animation starts, it will be smoother.

Test this by replacing the **`init()`** method in Figure 2 with the **`init()`** method in Figure 8. Be sure to recompile **`helloWorld.java`** and restart the browser to invoke the new version.

In one window, view the **`init()`** *method from Figure 8*
> view /u/java/Figures/Fig08.init.replace
From a second window, edit the helloWorld.java file:
> vi helloWorld.java
    Delete the current **`init()`** *method.*
    Paste the new **`init()`** *method where you just deleted the prior version.*
    Press <Esc> and then <Shift> ZZ in each window to exit the editors.
Recompile helloWorld.java
> javac helloWorld.java
Stop and then restart the Netscape browser, then set the Location *field to:*
    file:/u/netlab/java/hello.html

```
public void init() {
  int i;
  resize(150,150);
  MediaTracker tracker = new MediaTracker(this);

  for (i = 0; i<10; i++) {
    frame[i] = getImage(getDocumentBase(),"waving/T"+i+".gif");
    tracker.addImage(frame[i],i);
  }

  showStatus("loading images");
  try {
    tracker.waitForAll();
  } catch (InterruptedException e) {}
  showStatus("Images loaded");
}
```

Figure 8

You probably noticed the delay before the animation started on this test, so bear in mind that this is the tradeoff associated with preloading images.

5.3 Using Double–Buffering to reduce flicker.

Double–buffering provides the best alternative for reducing flicker by taking advantage of the fact that it's faster to display a frame than to draw an image on the screen. So, to implement double–buffering, an application draws the image to a frame that is not being displayed, and then simply displays this frame. To test this, make the following changes to helloWorld.java.

In one window, view the **`paint()`** *and* **`update()`** *methods from Figure 9*

\> view /u/java/Figures/Fig09.update.and.paint.replace

From a second window, edit the helloWorld.java file:

\> vi helloWorld.java

Delete the current **`paint()`** *and* **`update()`** *methods, then paste the new versions from Figure 9, where you just deleted the previous ones.*

Delete the line **`'public void init() {'`** *and then insert the lines provided in Figure 10 (*`/u/java/Figures/Fig10.init.partial.replace`*).*

Press <Esc> and then <Shift> ZZ in each window to exit the editors.

Recompile helloWorld.java

\> javac helloWorld.java

Stop and then restart the Netscape browser, then set the Location *field to:*

file:/u/netlab/java/hello.html

```
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 9

```
protected Image bufImage;
protected Graphics bufGraphic;

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

Figure 10

You probably noticed that the **`paint()`** method in Figure 9 does more work than our previous version (contradicting our earlier warnings about making **`paint()`** 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.

6. Java Panels, Widgets and Events

Our helloWorld applet has been drawing on the browser window much like someone would draw on a 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. In addition, Java provides a Layout Manager so your applet can add panels, containers and components within this frame or table entry. The browser communicates to these panels and components through *events*.

In this section we will begin a new applet, whiteBoard, which provides examples of how to use panels, widgets and events.

Before starting our whiteBoard, we need to decide on a layout. Select the **Bookmark** for **Java Packages** to go back to the Java documentation. Select the **java.awt** package and then select each type of **Layout** (Border, Card, Flow, GridBag & Grid) to view the layouts available to us. Our whiteBoard's main component will be the drawing board itself, but we'll also need a heading and some controls. The **`BorderLayout`** suits our needs pretty well.

We'll call the drawing board panel, **`wbPanel`**. We'll call the controls panel, **`wbControls`**. Let's think about **`wbPanel`**. We want users to draw lines, circles and text, but we'll start with just lines. We'll need to track what is on the board, it's location and color, so we'll simplify things by creating a **`Shape`** object. Since we don't know how many **`Shape`**s will be drawn, we can use Java's **`Vector`** class to create a dynamic array of **`Shape`**s.

In Figure 11, you can see the **`Shape`** and **`wbPanel`** classes. The **`Shape`** class includes a method which is also named **`Shape`**. A method which has the same name as it's class is called a *constructor*. Constructors are methods that create an object from a class. They initialize the new object and it's variables. In the case of **`Shape`**, the constructor records the location and color of the shape. The keyword **`this`** simply refers to the object within which the **`this`** is used.

Now look at the **`wbPanel`** class in Figure 11. Notice that it has an **`handleEvent()`** method. This is something new. The **`handleEvent()`** method is called whenever any event occurs within the component. Users will draw lines by pressing the mouse button (**`MOUSE_DOWN`**) at the starting point, dragging it to the end point, and then releasing the mouse button (**`MOUSE_UP`**). **`HandleEvent()`** is what will process these events. For the events **`handleEvent()`** processes itself, it returns a **`true`**. For events it does not handle, it returns a **`false`** so Java will simply take default actions.

Figure 12 shows the **`whiteBoard.java`** file which includes the **`whiteBoard`** and **`wbControls`** classes. Notice the **`whiteBoard`** class itself is very simple. It simply sets the layout, creates **`wbPanel`** and **`wbControls,`** and then places these objects in the layout.

The **`wbControls`** is slightly more complicated because it needs to allow user input. **`WbControls`** adds a **`Choice`** button (much like JavaScript's **`Select`**), so when the user makes a selection, it is passed to the **`action()`** method with an argument specifying the user's selection. This is easier than processing lower–level events like **`MOUSE_DOWN`**.

Copy the **`wb.java`** (Figure 11) and **`whiteBoard.java`** (Figure 12) files to your **`java`** directory. Create an HTML file to invoke the whiteBoard applet by making a copy of the **`hello.java`** file (call it **`wboard.html`**). In **`wboard.html`**, replace the **`helloWorld`** with **`whiteBoard`**, and delete the **`width`** and **`height`** fields. Then test your work.

```
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, xl,yl;
  public static Vector shapes;                               /* 1 */

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

Figure 11

```
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 12

Create the wb.java and whiteBoard.java files in your **`java`** *directory.*
> cp /u/java/Figures/Fig11.wb.java wb.java
> cp /u/java/Figures/Fig12.whiteBoard.java whiteBoard.java
Build the wboard.html file
> cp hello.html wboard.html
> vi wboard.html
Replace **`helloWorld`** *with* **`whiteBoard`**
Delete the **`width`** *and* **`height`** *fields*
Press <Esc> and then <Shift> ZZ to exit the editor.
Compile the new Java files.
> javac wb.java whiteBoard.java
Stop and then restart the Netscape browser, then set the Location *field to:*
file:/u/netlab/java/wboard.html

With this early version, you should be able to set the ink color and draw lines. The line doesn't display until you release the mouse button, but on the final version, it will display as you are drawing it. The final version also allows drawing circles and text on the whiteBoard, but for now, we want to keep it simple.

7. Java Communications (Inter–Applet and Sockets)

In our last exercise, we built a whiteBoard applet which created 2 panels, **`wbPanel`** and **`wbControls`**. Referring back to Figure 12, you'll notice whiteBoard's **`init()`** passes the **`wbPanel`** panel (or more appropriately, the **`wbPanel`** object) to **`wbControls`**. This was to allow the **`wbControls`** object to communicate changes in ink color to the **`wbPanel`** object. Also in Figure 12, you'll notice that **`wbControls`** communicates this change in ink simply by calling **`wbPanel`**'s **`setForeground()`** method. Painless, right? The good news is that inter–applet communications is just as easy.

7.1 Inter–Applet Communications

Here's the plan. We'll build an HTML file so we can display both helloWorld and whiteBoard on a single Web page. We'll add a RESTART button to the helloWorld applet to restart the animation and clear whiteBoard's drawings. In other words, when RESTART is pressed, helloWorld will have use inter–applet communications to tell whiteBoard to clear it's **`wbPanel`**.

To start, we need a common set of methods that both whiteBoard and helloWorld can call, but we don't want to force these two classes into the same superclass. Instead, we can build an **`abstract`** class (much like the **`abstract run()`** method we used to implement multithreading) which will be accessible to both whiteBoard and helloWorld.

Figure 13 illustrates the new **`abstract wb`** class which will represent the contents of our whiteBoard. Since **`wb`** represents whiteBoard's contents, we moved our **`Shape`** vector into this new **wb** class. When whiteBoard creates its **wbPanel**, it will call **setWb()** to initialize the contents. Notice the **synchronized** modifier used with each **wb** method. The **synchronized** modifier prevents more than one thread from executing the enclosed block of code at the same time.

```
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) {
    shapes.addElement(s);
  }
}
```

Figure 13

Insert the **wb** class in Figure 13 into **wb.java** after the **import** statement.

In one window, view the **wb** *class from Figure 13*
> view /u/java/Figures/Fig13.wb.insert
From a second window, edit the wb.java file:
> vi wb.java
Insert the **abstract wb** *class from Figure 13 directly after the* **import** *statements.*
Press <Esc> and then <Shift> ZZ to exit the editor.

You'll have to make a few other changes to utilize this new **wb** class. Refer to the original **wb.java** in Figure 11. Notice the bold, commented numbers 1 through 5. These numbers mark lines that will be changed. Make the following changes to **wb.java** and **whiteBoard.java**:

Edit the wb.java file:
> vi wb.java
Delete the lines marked by **`/*1*/`** *and* **`/*2*/`**.
Replace the line marked by **`/*3*/`** *with*
```
wb.addShape(s);
```
*Prefix the word "***`shapes`***" with "***`wb.`***" on the lines marked by* **`/*4*/`** *and* **`/*5*/`**.
Press <Esc> and then <Shift> ZZ to exit the editor.
> vi whiteBoard.java
After the line marked by **`/*1*/`**, *insert the statement:*
```
wb.setWb(wp);
```
This will initialize the whiteBoard.
Press <Esc> and then <Shift> ZZ to exit the editor.

This completes the changes to the whiteBoard code. The next step is to add the RESTART button to the helloWorld applet and recompile all the Java code.

View the **`handleEvent()`** *method in Figure 14:*
> view /u/java/Figures/Fig14.handleEvent.insert
In a second window, edit the helloWorld.java file:
> vi helloWorld.java
After the **`stop()`** *method, insert the* **`handleEvent()`** *method from Figure 14.*
After the call to **`resize(150,150)`** *in the* **`init()`** *method, add the line:*
```
add( new Button("RESTART"));
```
Press <Esc> and then <Shift> ZZ in to exit each editor.
Recompile the Java code:
> javac wb.java whiteBoard.java helloWorld.java


```
public boolean handleEvent(Event evt) {
  switch(evt.id) {
      case Event.ACTION_EVENT:
      if ((”RESTART”).equals(evt.arg)) {
        if (animationThread != null &&
            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 14

The only part left is to create the HTML file which will display both helloWorld and whiteBoard on the same Web page. Create the **`both.html`** file as shown in Figure 15. Open this HTML file in your browser, draw a few lines on the whiteBoard and then test the new RESTART button.

Create the both.html file in your **`java`** *directory.*

> cp /u/java/Figures/Fig15.both.html both.html

Stop and then restart the Netscape browser, then set the Location *field to:*

file:/u/netlab/java/both.html

Draw a few lines and then click on the RESTART button.

```
<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 15

7.2 Adding Socket Communications

Unlike inter–applet communications, applets running on different systems can’t just call each other’s methods. Additionally, Java security restricts applets from making network connections, except to the host that served the applet. So, to make our whiteBoard distributed, we need to implement a whiteBoard server to forward messages to and from client whiteBoard applets.

This whiteBoard server (call it wbServer) should run as a daemon on the Web Server so it will always be available for connecting clients. Therefore, unlike our helloWorld and whiteBoard applets, our wbServer will not be invoked from a browser. Java programs that are not invoked from a browser are called Java applications.

Java applications must have a **`main()`** method, which is called when the program is invoked. Also, just as an applet has a specific flow, applications using socket interfaces must have a specific flow. Figure 16 depicts a simple flow in which a server listens for and connects to incoming clients. A server associates itself with a particular port by *binding* itself to that port and then waits to *accept* incoming clients.

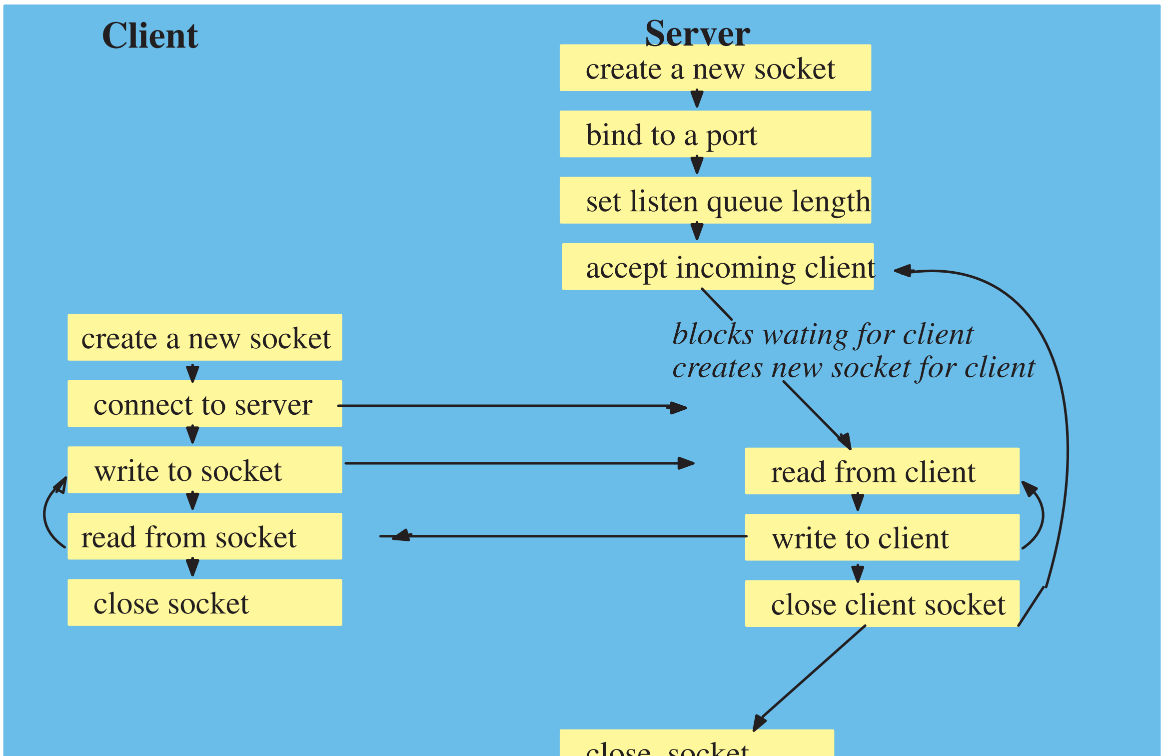

Figure 16

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 that client. The new socket is passed to the server that is bound to the requested port. All communication to this particular client is handled through this new socket. The process is repeated for each client.

Because the flow is different for servers and clients, Java implements a **`Socket`** class and a **`ServerSocket`** class. A **`ServerSocket`** listens on a particular logical port; when a client requests a connection to that port on the server system, a new **`Socket`** is created and returned to the server from the **`ServerSocket.accept()`** method.

Figure 17 illustrates how our whiteBoard clients will communicate to the wbServer. The idea is that whenever a user draws a shape, whiteBoard will add the shape to the local **`wbPanel`** and send a message to wbServer. The wbServer will forward this message to the socket for the remote whiteBoard client. The remote whiteBoard will update its **`wbPanel`** to reflect the change. Because the message representing the new shape is only 20 bytes, very little network traffic is generated and the update appears almost simultaneously.

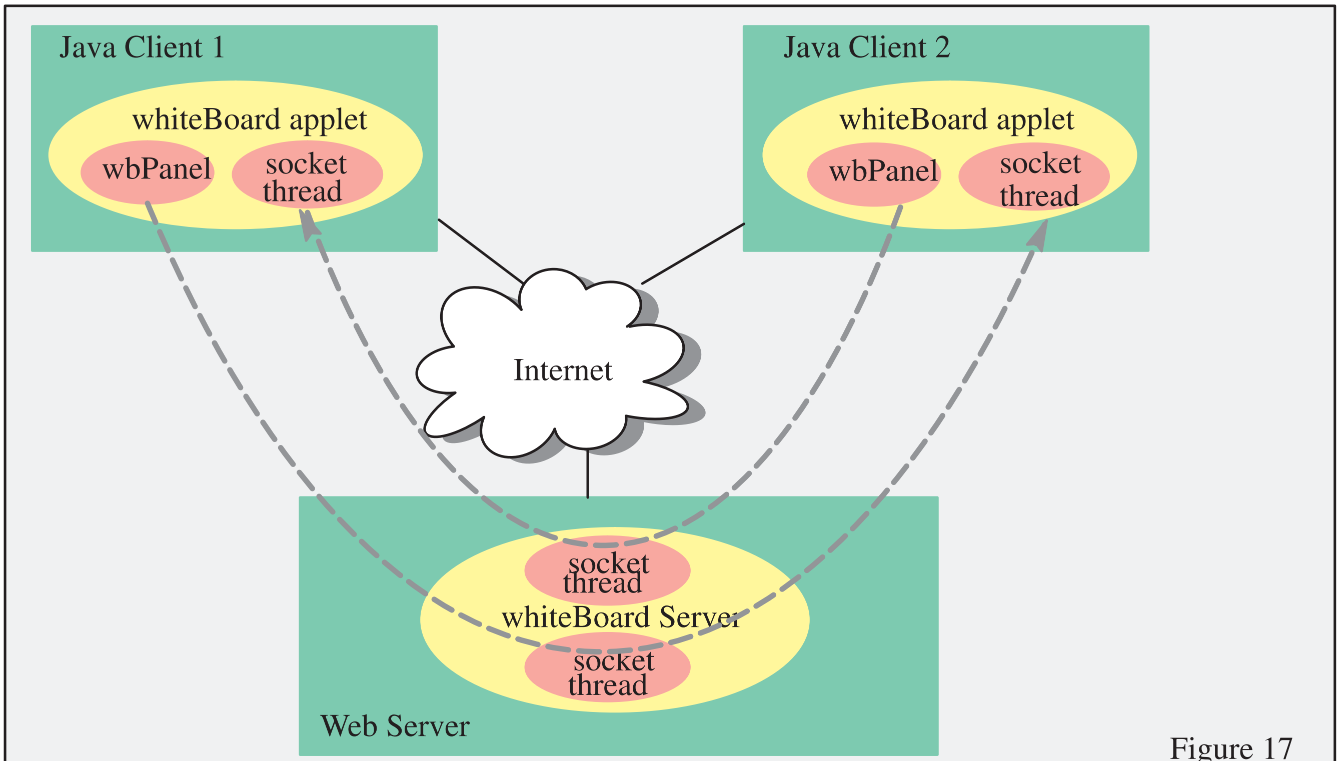

Figure 17

To create the wbServer, you'll need to do the following:

Copy the files for the wbServer application:
\> cp /u/java/Figures/Fig18.wbServer.java wbServer.java
\> cp /u/java/Figures/Fig19.serverSockThread.java serverSockThread.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;

    try {
      listener = new ServerSocket(port);
    } catch (IOException e) {
      System.out.println("ServerSocket failed");
      return;
    }

    while (true) {
      try { // get a pair of clients
        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(c2,c1);
      t2.start();
    }
  }
}
```

Figure 18

```
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 19

You'll also need to modify the client–side (whiteBoard) to handle socket messages. First, create the **`sockThread`** class which will read incoming messages from the wbServer. Then modify the whiteBoard applet to create a new thread (**`sockThread`**) when the applet is initialized and to stop this thread when the applet is destroyed. Finally, to ensure that when shapes are added to the local whiteBoard, they are also sent to the remote whiteBoard, the **`addShape()`** method will need to be modified to accept a parameter indicating whether the shape was drawn locally (set flag to **`true`**) or remotely (set flag to **`false`**). Use the following commands to implement these changes:

Create a sockThread class to read incoming messages from wbServer:

> cp /u/java/Figures/Fig20.sockThread.java sockThread.java

View the new whiteBoard class in Figure 21

> view /u/java/Figures/Fig21.whiteBoard.replace

In a second window, edit whiteBoard.java

> vi whiteBoard.java

Delete the whiteBoard class.

Insert the code from Figure 21 directly after the **`import`** *statements in whiteBoard.java*

Press <Esc> and <Shift> ZZ to exit both editors.

View the new wb class in Figure 22

> view /u/java/Figures/Fig22.wb.replace

In a second window, edit wb.java

> vi wb.java

Delete the wb class

Insert the code from Figure 22 directly after the **`import`** *statements in wb.java*

In the **`handleEvent()`** *method, replace the line "***`wb.addShape(s)`***" with:*

```
wb.addShape(s, true);
```

Press <Esc> and <Shift> ZZ to exit both editors.

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

public class sockThread extends Thread {
  wbPanel wp;
  DataInputStream in;

  sockThread(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);
      wp.repaint();
    }
  }
}
```

Figure 20

```
import java.net.*;

public class whiteBoard extends Applet {
  sockThread sockthread=null;
  Label wbTitle = new Label(”White Board”,Label.CENTER);

  public void init() {
    Socket sock;

    setLayout(new BorderLayout());
    wbPanel wp = new wbPanel();
    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 sockThread(wp,wb.in);
    sockthread.start();
  }

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

Figure 21

You may have noticed the **serverSockThread** uses the **DataInputStream** to filter the socket input. This filter is used so that if an error occurs or the remote disconnects, the **readInt()** method will throw an **IOException** or **EOFException**, respectively. We've simplified our code by catching the superclass for both of these exceptions, **Throwable**. By catching only **Throwable**, we are telling Java to do the same thing, regardless of which exception is thrown.


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

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 setWb(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 {
        out.writeInt(s.x1);
        out.writeInt(s.y1);
        out.writeInt(s.x2);
        out.writeInt(s.y2);
        out.writeInt(s.c.getRGB());
      } catch (IOException e) {
        wb.sockClose();
        return;
      }
    }
  }
}
```

Figure 22

You may have also noticed the **`finally`** keyword in **`serverSockThread`**'s **`cleanWb()`** method. **`Finally`** denotes statements which should 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, and then from a separate window, start 2 Netscape browsers. In both browsers, set the **Location** to the **`both.html`** HTML file created in Section 6.

Compile the modified Java code.

> javac wb.java serverSockThread.java wbServer.java

> javac sockThread.java whiteBoard.java

Start the wbServer application:

> java wbServer

In a second window, start 2 Netscape browsers

> netscape &

> netscape &

Set the Location *of both browsers to*

file:/u/netlab/java/both.html

Test your work by drawing lines on either of the whiteBoards; lines drawn on either browser should appear on both whiteBoards.

Where do you go from here? Well, the final copy of whiteBoard provided in **`/u/java/Final`** shows images as they are being drawn and allows users to draw circles and text, as well as lines, so you may want to review these enhancements. For more info and samples, try visiting the following Web Sites:

- IBM Centre for Java Technology (http://ncc.hursley.ibm.com/javainfo)
- Sun Java Home Page (http://java.sun.com)