Introduction to Java

by Marshall Brain

Comment on this article

Get the Acrobat PDF of this article Get Acrobat Reader

Introduction

Java allows the developer to create executable code that runs in the browser's memory space. Because Java is a complete programming language, it allows you to do almost anything you like. From within a Java program you can draw, paint bitmaps, get user events and respond to them, load URLs, etc.

If you look at the Java applets being used on the web today, you will find that a web developer uses Java for four primary tasks:

- Creating animated graphics, including bitmap-loops, moving bitmaps and animated drawings.
- Creating graphical objects (bar charts, graphs, diagrams, etc.) that you do not want to download from the server through a CGI script. Typically a graph is created by having a CGI script look at a database, producing a GIF, and then sending it back to the browser. You might not want to do this either because it would take too long to download a large GIF or because you do not want to have to manufacture a GIF on the server.
- Creating new "controls". If you want to create a pushbutton control with special properties, or a more complicated control that looks like a graphics equalizer, you could create it with Java. Since you can draw and get events, you can create any kind of control you like.
- Creating applications made up of collections of controls like edit areas, buttons, check boxes, etc. This is similar to what you can do with JavaScript, but the program is completely self-contained and compiled so that you do not reveal the source code.

Java is a complete object oriented programming language derived from C++. It has all of the OO advantages of C++, but does away with a number of the more unpleasant aspects like pointers and memory allocation in the name of sanity, robustness and security. Java the language also comes with a wide-ranging collection of libraries (also known as *packages*)that extend the language. There is a library of user interface objects called AWT, an I/O library, a network library, etc. You can use Java to create both applets that are loaded over the web and executed inside a browser, as well as stand-alone applications. We will focus only on browser applets in this tutorial, but because of Java's application capability it would be possible to use Java as your sole development environment.

If you would like to learn Java, it is quite helpful if you already know C++ and OO concepts. The focus of this tutorial is not on the language itself but what you can do with the language.

The Simplest Program

Before you can compile and test your own Java applets you need two things: the Java compiler and a Java-aware browser. The Java compiler comes with quite a bit of documentation covering the language and the class hierarchies shipped with the language, and this documentation will be essential reference material for you. The compiler package also contains quite a bit of sample code that can also be useful.

Assuming that you have the compiler and a Java-aware browser set up, then use a text editor and enter the following "hello world" program:

```
//hello.java

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

public class hello extends Applet
{
   public void paint(Graphics g)
   {
      for (int x = 0 ; x < size().height ; x++)
      {
         g.drawString("Hello World!", 10, x*15);
      }
   }
}
```

To compile this program, save the text to a file named "hello.java". At a command line type the following command:

```
   javac hello.java
```

Note: Case matters on the file name, even in Windows 9x and Windows NT.

This command will compile the Java code and create a file called "hello.class" containing the binary object code for the applet. Note that you will save yourself a lot of complaints from the compiler if the class name in the code and the name part of the file name (in this case, "hello") are the same.

What the compiler created is an *applet*. An applet is a binary file that contains executable code. To run this executable code you have to load it into a Java-aware browser. You can do that with the following HTML document, *which you should place in the same directory as your applet*:

```
<title>Hello World Demo</title>
<hr>
<applet code=hello.class width="300" height="120">
</applet>
<hr>
```

Load this document into the browser and you should see the Hello World applet running in a small 300 by 120 window in an HTML page.

Understanding the Program

Let's walk through the program line-by-line and see how it works. The code has been annotated with line numbers in the following listing:

```
//hello.java

1 import java.awt.*;
2 import java.applet.*;

3 public class hello extends Applet
4 {
5    public void paint(Graphics g)
6    {
7       for (int x = 0 ; x < size().height ; x++)
8       {
9          g.drawString("Hello World!", 10, x*15);
10      }
11   }
12}
```

The first two lines "import" needed libraries in much the same way that a include statement would in C or C++. In Java, however, libraries are in a way hierarchical, and you can load just the pieces that you need. Here we have loaded the applet and the AWT pieces from the java libraries.

At line 3 the code declares a new class named hello. The file name and the class name should be synchronized (since the applet is named hello the file needs to be hello.java), as mentioned in the previous section. The class inherits from the Applet class and extends it. The applet class has a method named paint (actually, the paint method comes from the Component superclass of Applet). This method is called on exposure. It receives a reference to the Graphics class for the applet.

The Graphics class for the applet is the applet's "window" in the browser. When you draw with the Graphics class, the drawing appears in the applet's window. At line 5 the paint method has been overridden and it loops 10 times drawing the word "Hello World!" in Line 9.

At this point it would be worth your while to go look up the Graphics class in the documentation. The class contains a number of useful drawing functions for lines, circles, boxes, etc. Try out a few of these functions on your own to get comfortable with them. Simply place calls to the different drawing commands in the paint method and see what happens. Also try out the "set" functions in the Graphics class. For example, the following code demonstrates how to set the color and the font:

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

public class hello extends Applet
{
   public void paint(Graphics g)
   {
      g.setColor(Color.red);
      g.setFont(new Font("Helvetica", Font.PLAIN, 24));
      for (int x = 0 ; x < size().height ; x++)
      {
         g.drawString("Hello World!", 10, x*15);
      }
   }
}
```

To see this program in action, click here.

Note: In our experience, it is best to reload the HTML page into a fresh version of the browser on each run when you are developing Java code. The "Reload" button, at least in version 2.0 of NetScape, does not seem to reload the applet.

The AWT Library

Java's AWT library allows you to create user interfaces inside of Java applets. The AWT library contains all of the controls that you would expect to find in a simple interface development library: buttons, edit areas, check boxes and so on. The library also has several different containers that you use to arrange your controls on the screen, along with many other features.

The following code demonstrates how to create a push button and place both into a container called a BorderLayout.

```
// button_test.java

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

public class button_test extends Applet
{
   int y;
   Button b;

   public button_test()
   {
      y = 10;
      setLayout(new BorderLayout());

      b = new Button("Apply");
      add("North", b);
   }
   public void paint(Graphics g)
   {
      for (int x = 0 ; x < size().height ; x+=10)
      {
            g.drawString("Hello World!", y, x);
      }
   }
}
```

This program is not remarkably different from the original Hello World program. The difference lies in the constructor for the applet. In the constructor, the code initializes the variable "y", sets a layout for the applet, creates a new button control that displays the word "Apply" on its face, and then adds the button to the layout. A BorderLayout has five attachment points: North, South, East, West and Center. Once you have attached things, the layout handles the details of positioning the objects depending on the applet's size on the screen. In this case we have placed the button to the North (top) of the layout. The rest of the window will contain the text generated in the paint method.

Compile and run this program using the steps described above. You will find that the applet paints the button and then fills the rest of the applet's window with "Hello World."

Look up the Button control and the BorderLayout container in the documentation so that you begin to get a feeling for the options. Lookup the setLayout and add methods in the Applet class documentation. Note how the applet itself "owns" the layout and adds things to it. Look up several of the other controls in the documentation and try them in place of (not in addition to - in place of) the button control.

Getting Events

When you click the button in the previous example it does not do anything. To make it do something you need to handle its events. The easiest way to do that is to add an event handling function to the applet's class (although another way is to inherit a new button from the base button class and handle the event there).

The following code demonstrates an extremely simple event handler:

```
// button_test.java

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

public class button_test extends Applet
{
   int y;
   Button b;

   public button_test()
   {
      y = 10;
      setLayout(new BorderLayout());

      b = new Button("Apply");
      add("North", b);
   }
   public void paint(Graphics g)
   {
      for (int x = 0 ; x < size().height ; x+=10)
      {
            g.drawString("Hello World!", y, x);
      }
   }
   public boolean action(Event ev, Object arg)
   {
      if (ev.target instanceof Button)
      {
          y+=10;
          repaint();
          return true;
      }
      return false;
   }
}
```

The "action" method in this applet taps into the event stream of the applet. As each event arrives it examines it to see if it comes from a Button object. If so it increments y and forces the applet to repaint itself. When you run the program you will find that the column of Hello Worlds moves over by 10 pixels each time you push the button.

If your applet contained two buttons, one to the north and one to the south, and you wanted to respond differently to them, the above code would have a problem because *any* event that comes from *any* button triggers the action. You respond to individual buttons by adding a line of code like the following:

```
   public boolean action(Event ev, Object arg)
   {
      if (ev.target instanceof Button)
      {
          if("Apply".equals(ev.arg))
          {
             y+=10;
             repaint();
             return true;
          }
      }
      return false;
   }
```

The ev.arg attribute carries the label from the button that gets clicked, and by comparing it to the button's label you can tell which button the user pressed. To see the program in action, click here.

Try to add other buttons to the BorderLayout on the south, east and west sides and have them do different things to the drawing that appears in the center.

Panels

From the previous exercise you may have noticed two things:

You can only have one control in each of the five positions on the BorderLayout

If you place buttons on the outside edges of the BorderLayout, the drawing is not a separate "entity". That is, the "Hello World" string is painted onto the applet's canvas and the buttons cover over parts of it. You would instead like the drawing to occur in its own canvas that is appropriately centered.

To solve both of these problems you can use a "Panel" (or in the case of the drawing area you could use a "Canvas"). In particular, a Panel is good for collecting together sets of controls. It arranges them in a row. The row will "wrap" if space limitations on the panel make it necessary.

```
// simple_UI.java

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

class panel_graph extends Panel
{
   int x = 10;

   public void paint(Graphics g)
   {
      for (int y = 0 ; y < size().height ; y+=15)
      {
            g.drawString("Hello World!", x, y);
      }
   }
   public void changeX()
   {
      x += 10;
      repaint();
   }
}

class panel_UI extends Panel
{
   Choice c;
   panel_graph pg;

   public panel_UI(panel_graph pgIn)
   {
      add(new Button("Apply"));
      add(new Label("Test Colors", Label.RIGHT));

      add(c = new Choice());
      c.addItem("Red");
      c.addItem("Green");
      c.addItem("Blue");
      pg = pgIn;
   }
   public boolean action(Event ev, Object arg)
   {
      if (ev.target instanceof Button)
      {
            pg.changeX();
            return true;
      }
      return false;
   }
}

public class simple_UI extends Applet
{
   panel_graph pg;
   panel_UI pui;

   public simple_UI()
   {
      setLayout(new BorderLayout());

      pg = new panel_graph();
      add("Center", pg);

      Panel p = new panel_UI(pg);
      add("North", p);
   }
}
```

In this program there are two new classes that both extend the Panel class. In the case of panel_graph, the panel acts like a drawing area into which the class draws "Hello World". This class has a paint method identical to those we have seen in the past, along with a changeX method that another class can call to move "Hello World" over.

The panel_UI class also extends a panel. This class's constructor adds three controls to the panel: a button, a label and a selection list (which when you run the program looks nice, but currently does nothing). The panel also recognizes button clicks and in response calls the changeX method in the panel_graph class.

The applet class simply creates the two panels and adds them to its layout. The panels do the rest. It would be possible, as an alternative design, to have the applet respond to the button events and call changeX in panel_graph from there. You can decide which approach makes you more comfortable.

Compile and run this program. You will find the panel to the north contains the three user interface controls, while the rest of the window is filled with the graphics panel. If you click on the button the Hello World text will move over. To see this program in action, click here.

To complete this program, we need to be able to change the color in the panel_graph based on the color selected by the user. To do this, change panel_graph so that it looks like this:

```
class panel_graph extends Panel
{
   int x = 10;
   Color c = Color.red;

   public void paint(Graphics g)
   {
      g.setColor(c);
      for (int y = 0 ; y < size().height ; y+=15)
      {
            g.drawString("Hello World!", x, y);
      }
   }
   public void changeX()
   {
      x += 10;
      repaint();
   }
   public void changeColor(Color cin)
   {
      c = cin;
      repaint();
   }
}
```

Then change the panel_UI event handler so it looks like this:

```
   public boolean action(Event ev, Object arg)
   {
      if (ev.target instanceof Button)
      {
         if ("Red".equals(c.getSelectedItem()))
```

```
            pg.changeColor(Color.red);
         if ("Green".equals(c.getSelectedItem()))
            pg.changeColor(Color.green);
         if ("Blue".equals(c.getSelectedItem()))
            pg.changeColor(Color.blue);
         return true;
      }
      return false;
   }
```

Compile and run the new program. Click here to see it in action. What you will find is that you can select a color in the color list and then click the Apply button to change the color in the graphics panel. The event handler extracts the user's choice using the getSelectedItem method of the Choice control, and then it calls changeColor in panel_graph to change the color. Note that the Color class has members that you could use to eliminate the If chain seen in the code above.

Applet Parameters

There are many cases where you would like to customize an applet by passing parameters into it when it starts. Fortunately this is easy to do. On the HTML side, you can pass as many parameters as you like when you invoke the applet. For example, you could change the HTML file so that it looks like this:

```
<title>Hello World Demo</title>
<hr>
<applet code=hello.class width="300" height="120">
<param name=rows value="4">
</applet>
<hr>
```

The parameter can have any name that you like. On the applet side, you retrieve the parameter's value with the getParameter method of the Applet class:

```
String r = getParameter("rows");
```

If the parameter name does not exist, then getParameter returns null, which you can test as follows:

```
int rs;
if (r == null)
{
   rs = 10; // default value
}
else
{
   rs = Integer.parseInt(r);
}
```

Images and Animation

Java is multi-threaded. Java also makes it easy to work with bitmaps (also known as *images*). You can combine these two facilities to create animation. The following code demonstrates a very simple animation. It slides a bitmap downward.

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

public class logo extends Applet implements Runnable
{
    Image img;
    Thread thd = null;
    int i;
    int imgWidth = 359;
    int imgHeight = 121;

    public void run()
    {
        img = getImage(getCodeBase(), "ITI_logo.jpg");
        if (img != null)
        {
            i=imgHeight;
            repaint();
            while (true)
            {
                try {Thread.sleep(1000);} catch (InterruptedException e){}
                i=0;
                while (i<imgHeight)
                {
                    repaint();
                    try {Thread.sleep(50);} catch (InterruptedException e){}
                    i+=4;
                }
            }
        }
    }
    public void update(Graphics g)
    {
        if (img != null)
        {
            g.clipRect(0, 0, imgWidth, i);
            g.drawImage(img, 0, i - imgHeight, null);
        }
    }
    public void start()
    {
        if (thd == null)
        {
            thd = new Thread(this);
            thd.start();
        }
    }
    public void stop()
    {
        thd = null;
    }
}
```

To try out this code, click here.

To understand this code, start by reading about the thread class in the java documentation. The run method represents a new thread. In this separate thread, the code loads an image and then animates it by repainting it at a new position every 50 milliseconds in the innermost while loop. Then the code waits for one second and repeats the animation.

Each time the code wants to paint the bitmap at a new position, it calls repaint. This function calls the overridden update method. The update method is identical to the paint method, except that the paint method clears the window before drawing and the update method does not (try renaming the overridden update method with the name paint and you will clearly see the difference). Here the update method simply draws the image at the position specified by i. As an experiment try to comment out the clipRect line in the update method. You may notice that the animation is a bit jumpier.

You can see that if you wanted to animate a set of 15 frames, all that you would have to do is load all 15 frames into an array of images and paint each one in sequence in the update method. It is very easy.

Developed under:
JDK 1.3

Introduction to Java

by Marshall Brain

Back to Java tutorial.

Introduction to Java

by Marshall Brain

Back to Java tutorial.

Introduction to Java

by Marshall Brain

Back to Java tutorial.

Introduction to Java

by Marshall Brain

Back to Java tutorial.

Introduction to Java

by Marshall Brain

Back to Java tutorial.
