Java™ Applets

Rex Jaeschke

Java Applets

© 1997–1998, 2009 Rex Jaeschke. All rights reserved.

Edition: 3.0 (matches JDK1.6/Java 2)

The training materials associated with this book are available for license. Interested parties should contact the author at the address below.

Please address comments, corrections, and questions to the author:

Rex Jaeschke
2051 Swans Neck Way
Reston, VA 20191-4023
+1 (703) 860-0091
+1 (703) 860-3008 (fax)
www.RexJaeschke.com
rex@RexJaeschke.com

Preface

Whereas a *Java application* can run stand-alone, a *Java applet* can run only under the control of a Java-compatible web *browser* (such as Internet Explorer) or under Sun's Applet Viewer.[1]

In this book, we'll learn how to implement and run applets. This book is not hardware, browser, or operating system-specific.

Assuming you meet the prerequisites, by the end of this book you should have a good understanding of the following topics:

- How an applet differs from an application.
- How to pass parameters from HTML to an applet.
- The display area and the `paint` method.
- The `init`, `start`, `stop`, and `destroy` methods.
- How applets can take advantage of threads.

Reader Assumptions

This text follows on from its companion *Programming in Java*. You are expected to be comfortable with the following concepts and the syntax required to express them in Java:

- Basic Language Elements
- Looping and Testing
- Methods
- References, Strings, and Arrays
- Classes
- Inheritance
- Exception handling
- Input and Output
- Packages
- Interfaces

To understand the use of threading in applets, you are expected to be conversant with threading in applications, which is covered in the companion book *Programming in Advanced Java*.

Some knowledge of HTML would be useful but is not necessary.

[1] Throughout this book, the term *browser* is meant to include the Applet Viewer as well. When there are differences between the two, these will be noted.

Limitations

This text does *not* cover the use of the Java Abstract Windows Toolkit (AWT); that topic is addressed in a companion book *Programming with Java AWT*.

Source Code, Exercises, and Solutions

The programs shown in the text are made available electronically in a directory tree named Source.

Each chapter contains exercises, some of which have the character * following their number. For each exercise so marked, a solution is provided on disk in a directory tree named Labs.[1] Exercises that are not so marked have no general solution and require experimentation or research in an implementation's documentation. Numerous exercises contain a statement of the form "(See directory *xx*.)". This indicates the corresponding solution or test file in the Labs subdirectory.

You are strongly encouraged to solve all exercises in one section before continuing to the next. Also, invent your own exercises as you go and be inquisitive; don't be afraid to experiment. Try to understand why the compiler gives you each error or why a program fails at run time.

The Java Development Kit

Sun's initial production release of Java was the Java Development Kit (JDK) version 1.0. Versions 1.1 through 1.6 contained numerous bug fixes, and language and library enhancements. The latest version can be downloaded from Sun's website (www.sun.com).

While the language has remained very stable, along the way, several features were added along with numerous new packages and classes and new methods to existing classes. Also, some existing method names have been changed. In these latter cases, the old names continue to be acceptable, but are flagged by the compiler as *deprecated*, meaning that support for them might well be removed in future versions. If your compiler issues such a warning, consult Sun's on-line documentation to find the recommended replacement.

From an internationalization (I18N) viewpoint, one of the most significant additions made by V1.1 was the completion of support for dealing with non-US, non-English environments, including those involving very large alphabets and non-Latin writing systems.

Rex Jaeschke, September 2009

[1] The solutions are only available to licensees of these materials when they are used in formal training scenarios.

1. Getting Started

The following simple applet (see Ap01.java) draws the text "Welcome" on the screen starting at some coordinate position. We say "draws", since an applet really runs in graphics mode; subsequently, text-mode methods like `System.out`'s `print` and `println` are not used in applets:[1]

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

public class Ap01 extends Applet
{
    public void paint(Graphics g)
    {
          g.drawString("Welcome", 10, 20);
    }
}
```

The first thing we notice is the presence of two `import` declarations: the first is necessary for every applet while the second is necessary if our applet is to use the Abstract Windows Toolkit (AWT). As all useful applets involve drawing, and the AWT contains the drawing tools, we really need its package as well.

Every applet must contain a class that extends `Applet`, support for which comes from the package `java.applet`. Let us call the extending class the *primary class*. Because class `Applet` provides all of the program start-up and screen management support, an applet inherits a large amount of support from its superclasses. As a result, an applet need not have a `main` method and can be quite simple. Note that the primary class must be public so it can be found by a browser.

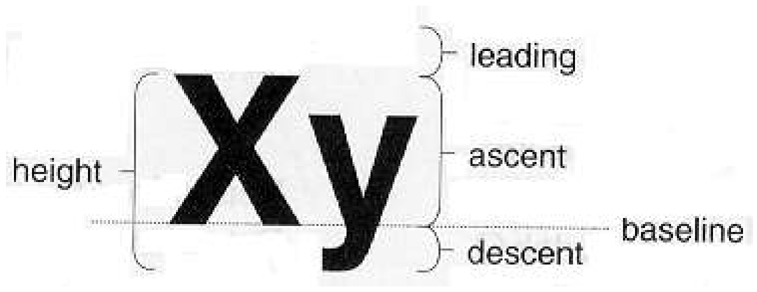
To do useful work, the primary class must override at least one of its superclasses' methods. The simplest and most common one to be overridden is `Container.paint` This method is called directly by the browser, and indirectly by the applet, at certain times to redraw the display area. `paint` must have the signature as shown. The argument passed into this method is known as a *graphics context* and it represents the rectangular drawing area displayed by the browser. `Graphics` is the abstract base class for all graphics contexts that allow an application to draw onto components or onto off-screen images. By default, all drawing or writing is done in the current color, using the current paint mode, and in the current font. Therefore, to draw anything on our applet's display area, we must specify that area when we invoke the drawing method. For example, `g.drawString` draws a string on the display area designated by `g`.

The display area is two dimensional and its origin is located at the top-left corner; that is, location (10,20) is 10 pixels in from the left edge and 20 pixels down from the top edge. `drawString` draws its string first argument

[1] Some browsers do write text output to a special place.

from left-to-right starting at the location given by its second and third arguments, which represent the baseline of the first character.

The following picture shows the various attributes a font can have:



A browser interprets text and commands coded using HyperText Markup Language (HTML). One such command is `<applet>` and that is used to run an applet. The following HTML file (see Ap01.html) can be used to execute the .class file that results when Ap01.java is compiled:

```
<! Ap01.html - the html file for applet Ap01.java >

<html>
<head>
<title>Ap01</title>
</head>
<body>
<hr>
<applet>
   code=Ap01.class
   width=100
   height=50
>
</applet
<hr>
</body>
</html>
```

HTML commands, or *tags* as they are called, are delimited with <…>, and, for the most part, they come in pairs. For example, `<head>` } and `</head>`start and end, respectively, the applet header, and `<title>` and `</title>` start and end the applet's title. `<!` and `>` delimit a comment. Tags can span multiple lines and they can be written in upper-, lower-, or mixed-case. (As HTML is a whole subject on its own, we will discuss only those aspects of it that are relevant to applets. For more details on HTML, refer to Annex A .)

An HTML file is divided into two sections: the head and the body. In the head, we define the title that the browser displays on its title line. The body contains any text and text-formatting tags that we want to be displayed or interpreted by the browser, along with the `<applet>` tags needed to invoke any applets associated with that HTML document.

The tag `<hr>` draws a horizontal rule the full width of the browser display. (In all our examples, we'll use rules like these to delimit the horizontal edges of our applets' display areas.)

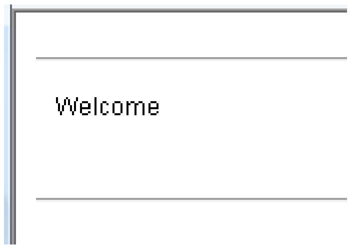
The general format of the `<applet>` tag is as follows:

```
<applet [ options ] >
```

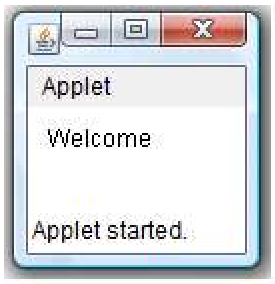
In our example above, we have three options: `code` specifies the .class file at which the applet should start executing (the browser looks for this file in the same directory as the HTML document that contains the tag); and `width` and `height` specify the size in pixels of the applet's display area. So, given the values 100 and 50, the on-screen coordinate range for our applet is (0,0) through (99,49).

When the browser finds the class file, it loads it (from across a network, if necessary) onto the computer on which the browser is running. The browser then creates an instance of the primary class, in this case, Ap01.

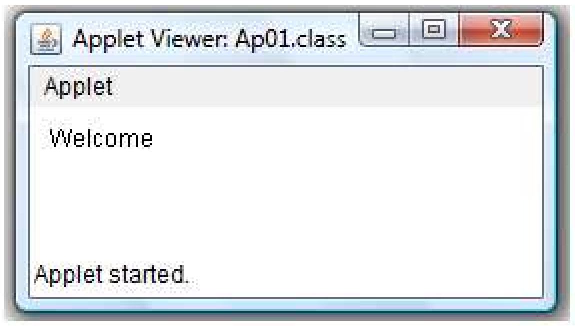
Here's what the output looks like when we browse this HTML file using Internet Explorer:


```
Welcome
```

And here's the output when using Applet Viewer:



```
Welcome
Applet started.
```



```
Welcome
Applet started.
```

Unlike the browsers, Applet Viewer allows the size of its display specified by the HTML file to be overridden. As we can see in the right-hand picture above, the width and height have been increased. We'll exploit this feature of Applet Viewer in future examples.

3. Display Area and Drawing Attributes

From within an applet, we can find out the size of our display area; we can also change fonts and colors (see Ap03). Here are the relevant parts of the HTML (see Ap03.html):

```
<applet
    code=Ap03.class
    width=200
    height=100
>
</applet>
```

and here's the Java source:

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

public class Ap03 extends Applet
{
    public void paint(Graphics g)
    {
          int width = getSize().width;
          int height = getSize().height;

          g.drawRect(0, 0, width - 1, height - 1);
          g.setColor(Color.blue);
          g.setFont(new Font("Serif", Font.BOLD | Font.ITALIC, height/3));
          g.drawString("Welcome", width/10, height/2);
    }
}
```

`Component` is a superclass of `Applet`, and `Component` contains a method called `getSize`,[1] which returns the size of the current display area as a `Dimension`. Class `Dimension` encapsulates the width and height of a component in a single object; it contains two public `int` fields: `height` and `width`.

`Graphics:drawRect` draws a 1-pixel thick rectangle using the current color. Its first two arguments represent the coordinates of the top-left corner of that rectangle while the other two arguments represent the width and height, respectively. (Two related methods, `Graphics.fillRect` and `Graphics.clearRect` fill the specified rectangle with the current color or background color, respectively. The draw version draws an extra pixel along the bottom and right edges.)

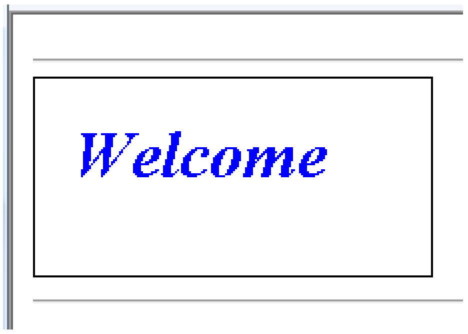
[1] `getSize` was a JDK1.1 invention. In JDK1.0 it was called `size`.

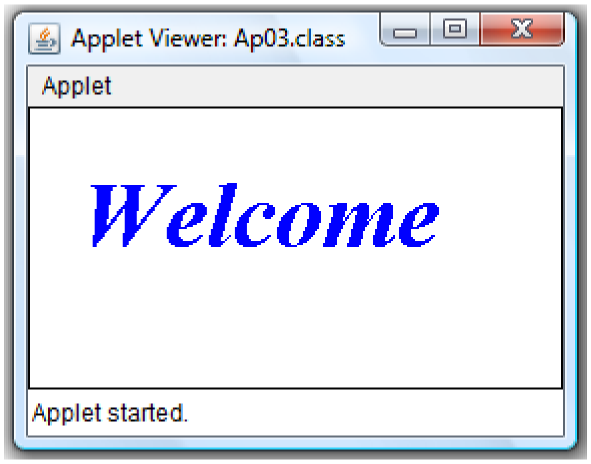
`Graphics.setColor` allows us to set the foreground color for this graphics context. All subsequent graphics operations using this graphics context will use this color. The class `Color` contains a number of predefined `final static Color` fields that represent the 13 colors including blue. We can also construct any color by mixing red, green, and blue in the desired mix, by varying the amount of each color from 0 to 255.

`Component.setFont` sets the current font by passing in a Font. (Its sibling method `Component.getFont` returns the current font in an object of type `Font`.) We construct a Font by specifying its name, style, and size. This example uses Serif. There are three predefined styles: `BOLD`, `ITALIC`, and `PLAIN`, and the size is measured in points. The style fields are integers and can be ORed together as shown above. The available fonts[1] are: `Dialog`, `DialogInput`, `Monospaced`, `Serif`, `SansSerif`, and `Symbol`.

Finally, we draw the string "Welcome" at a location that is a function of the display area's size.

Here's the output from Internet Explorer and Applet Viewer:





As Applet Viewer allows us to change the display area's size, we can test our applet to make sure it is not dependent on absolute sizes and/or coordinates. (By the way, we can change our display area's size explicitly from within an applet by calling `Applet:resize`.)

Exercise 3-1: In Ap03, what happens if the HTML display width is set to 100 instead of 200? Try it using both a browser and Applet Viewer.

Exercise 3-2*: In Ap03, the color is hard-coded. How can we specify the color via a parameter? The method `setColor` is declared as follows:

void setColor(Color c)

where `Color` is an object type. As each parameter is passed into an applet from the browser or Applet Viewer as a string, what we need to do is to take the string passed in and somehow create a corresponding Color. The valid values for `c` are: Color.black, Color.darkGray, Color.lightGray,

[1] The names `Monospaced`, `Serif`, and `SansSerif` were introduced in JDK1.1, at which time their predecessors `Courier`, `TimesRoman`, and `Helvetica`, were deprecated.

Color.pink, Color.yellow, Color.blue, Color.gray, Color.magenta, Color.red, Color.cyan, Color.green, Color.orange, and Color.white.

Add parameter support for the display color. (See directory Lbap02.)

The font style might be easier to handle since the three possibilities, `Font.BOLD`, `Font.ITALIC`, and `Font.PLAIN`, are `int`s that can be ORed together. As such, we could pass in a parameter having the corresponding `int` value. However, does the `Font` class require that all implementations use the same value for these fields? If not, how can we be sure our code will work with implementations that use different values?

Exercise 3-3: How would we go about decoding two parameters that represented Boolean and floating-point values, respectively?

Exercise 3-4*: Create a version of Ap03 that uses the method Font.decode to create a Font rather than using Font.Font. For example:

```
Font.decode("Monospaced-bolditalic-16")      // as specified
Font.decode("Monospaced-bolditalic")         // defaults to 12-point
Font.decode("Monospaced")                    // defaults to 12-point plain
Font.decode("Monospaced–16")                 // defaults to plain
```

(See directory Lbap03.)

Exercise 3-5*: Enhance the previous solution by centering horizontally and vertically the display text in the display area. (Hint: use class `FontMetrics` and obtain an object of this type for any given font by calling Graphics:getFontMetrics passing that method the corresponding Font. See directory Lbap04.)

Exercise 3-6*: Enhance the previous solution by using the largest possible font while ensuring that the display string is completely visible in the display area. (See directory Lbap05.)

Exercise 3-7*: Using GraphicsEnvironment.getAvailableFontFamilyNames, find the names of all of the fonts supported on your system. Display each font name on its own line in 24-point Serif, then set to the corresponding font and display a variety of characters on the same line. (Hint: to obtain a GraphicsEnvironment[1] object, call GraphicsEnvironment.getLocalGraphicsEnvironment. See directory Lbap06.)

Here's an example of some output from one system:

[1] Prior to the addition of this class to the standard library, class `Toolkit`'s methods `getDefaultToolkit` and `getFontList` were used instead; however, they have been deprecated.

8. Applets with Multiple Threads

While an applet can display static information or information that changes only as the result of some user action, very often, applets contain multiple, and sometimes asynchronous, threads executing in parallel. The example below shows how by having a second thread draw an X at some random spot on the display area every 100 milliseconds while the primary thread can be made available for other activities such as user interaction. (See Ap06):

```
<html>
<head>
<title>Ap06</title>
</head>
<body>
<hr>
<applet
    code=Ap06.class
    width=250
    height=100
>
    <param name=sleepTime value=100>
</applet>
<hr>
</body>
</html>
```

And here's the source code (see Ap06):

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

public class Ap06 extends Applet implements Runnable
{
    private Thread moverThread = null;
    private int count = 0;
    private int parmSleep = 500;
```

The simplest way to have a program create a thread is to derive its primary class from `Thread`. However, this is not possible with an applet, as an applet must already be derived from `Applet`,and a class can have only one direct superclass. Therefore, we must have the primary class implement the interface `Runnable`.

```
    public void init()
    {
          String param = getParameter("sleepTime");
          if (param != null) {
                parmSleep = Integer.parseInt(param);
          }
    }

    public void paint(Graphics g)
    {
          g.drawRect(0, 0, getSize().width - 1, getSize().height - 1);
          g.drawString("" + ++count, 5, 15);

          int x = (int)(Math.random() * getSize().width);
          int y = (int)(Math.random() * getSize().height);

          g.drawString("X", x, y);
    }
```

As we can see, `paint` simply draws a rectangle around the whole drawing surface, increments and displays the display count, computes the random display position, and then displays the X at that location.

```
    public void start()
    {
          if (moverThread == null) {
                moverThread = new Thread(this);
                moverThread.start();
          }
    }
```

Each time this applet is started, we spawn off another thread and start it running.

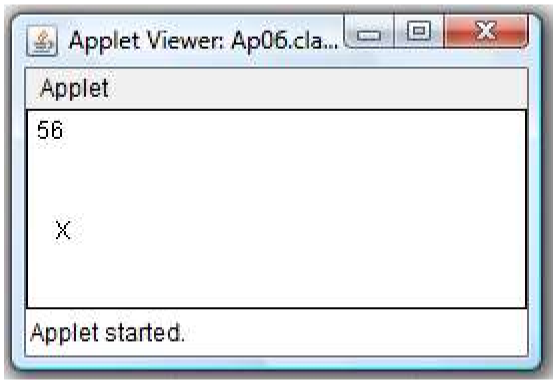
```
    public void stop()
    {
          if (moverThread != null) {
                moverThread.stop();
                moverThread == null;
          }
    }
```

Each time this applet is stopped, we kill the thread we started earlier.

```
    public void run()
    {
          while (true) {
                repaint();

                try {
                      Thread.currentThread().sleep(parmSleep);
                }
                catch (InterruptedException e) {
                      stop();
                }
          }
    }
}
```

The new thread begins execution at the method `run`, as specified by the interface `Runnable`. The thread loops indefinitely, calling `paint` indirectly via `repaint` every 100 milliseconds. Here's a snapshot of the output produced with Applet Viewer:



Exercise 8-1: If you have Applet Viewer, use it to run Ap06 and change the display area's width and/or height, to see if the whole display area is used each time `paint` is called.

Exercise 8-2*: Make a version of Ap06.java that displays the image stored in file Germany.gif instead of the letter X. (Hint: See `Applet.getImage` and `Graphics.drawImage`. See directory Lbap07.)

As we have seen, an HTML file can have multiple applets running on the same document page. The HTML below does just this; in fact, it runs two copies of Ap06.class, each having a different display area size and time delay. As we have multiple copies of an applet on the same page, the browser loads the class file once and creates two instances of the primary class. (See Ap07):