P.O. Box 342-01000 Thika

Email: info@mku.ac.ke

Web: www.mku.ac.ke

COURSE CODE: BIT 2205

COURSE TITLE: OBJECT ORIENTED PROGRAMMING II

Instructional Manual for BBIT – Distance Learning

Prepared by Paul M Kathale, mutindaz@yahoo.com

BIT 2205: OBJECT ORIENTED PROGRAMMING II

COURSE OUTLINE

1. Overview of OOP Technology (Objects and Classes)
   - Creating classes
   - Class declaration
   - Implementing methods
   - Creating Objects and using Objects
   - Method overloading
   - Method overriding
   - Inheritance
   - Constructors

2. *Java Applet Basics*
   - How Applets and Applications Are Different
   - Creating Applets
   - Major Applet Activities
   - A Simple Applet
   - Including an Applet on a Web Page
   - The <APPLET> Tag
   - More About the <APPLET> Tag
   - ALIGNMENTS
   - HSPACE and VSPACE
   - CODE and CODEBASE

3. Creating User Interface Component
   - AWT Component Overview
   - Adding Component to Applets
   - AWT Components
   - Label
   - Button
   - Choice
   - List
   - Text Field
   - Text Area
   - Layout Managers
     - Flow layout
     - Grid layout
     - Boarder layout
     - Card layout
     - Grid layout
4. Creating User Interface – Swing Components
   - A swing Component Overview
   - swing Components
   - JLabel

- JButton
- JChoice
- JList
- JTextField
- JTextArea

5. Drawing Graphics:
   - Shapes: Lines,Circles,Rectangles, Polygons and Colors.
   - Swing components
     - Jcomponent classes
     - Event Listener object
     - WindowAdapter classes
     - JMenuBar class
     - The popupMenu class
     - Dialogs
     -
6. Retrieving and Using Images
   - Getting Images
   - Drawing images

7. Event Handling
   - Event Listener Object
   - Event Object
   - Window Adapter classes
   - Extending Window Adapter
   - Action Events
   - Registering the Listener
   - Exception-handler parameters

NB: Implementation language: Java

CATs and Assignments – 30%

Final Examination - 70%

Java Applets

- Java applications are standalone Java programs that can be run by using just the Java interpreter, for example, from a command line.
- Java applets run from inside a World Wide Web browser. They are embedded on Web page as image is done.
- A reference to an applet embedded in a Web page is done using a special HTML tag.
- When a reader, using a Java-enabled browser, loads a Web page with an applet in it, the browser downloads that applet from a Web server and executes it on the local system (the one the browser is running on).
- The Java interpreter is built into the browser and runs the compiled Java class file from there.

Java Applet Security Restrictions

There is the set of restrictions placed on how applets can operate in the name of security. Given the fact that Java applets can be downloaded from any site on the World Wide Web and run on a client's system, Java-enabled browsers and tools limit what can be done to prevent a rogue applet from causing system damage or security breaches. Without these restrictions in place, Java applets could be written to contain viruses or Trojan horses (programs that seem friendly but do some sort of damage to the system), or be used to compromise the security of the system that runs them. The restrictions on applets include the following:

- Applets **can't read or write to the reader's file system**, which means they cannot delete files or test to see what programs you have installed on the hard drive.
- Applets **can't communicate with any network server** other than the one that had originally stored the applet, to prevent the applet from attacking another system from the reader's system.
- Applets **can't run any programs on the reader's system**. For UNIX systems, this includes forking a process.
- Applets **can't load programs native to the local platform**, including shared libraries such as DLLs.

Creating Applets

- To create an applet, you create a subclass of the class **`Applet.`**
- The `Applet` class, part of the **`java.applet`** package, provides much of the behavior your applet needs to work inside a Java-enabled browser.

  Therefore need to import the package as import **java.applet. Applet;**

  Then extend the class as follows

  `public class MyApplet extends Applet{`

Or

```
public class MyApplet extends java.applet.Applet {
```

- Applets also take strong advantage of Java's **Abstract Windowing Toolkit (awt),** which provides behavior for creating graphical user interface (GUI)-based applets and applications:
  - drawing to the screen; creating windows,
  - menu bars,
  - buttons,
  - check boxes, and
  - other UI elements;
- AWT is also used for managing user input such as mouse clicks and keypresses. The awt classes are part of the `java.awt` package.

New Term
Java's Abstract Windowing Toolkit (awt) provides classes and behavior for creating GUI-based applications in Java. Applets make use of many of the capabilities in the awt.

Need to import it.

```
import java.awt.Graphics;
import java.awt.Font;
import java.awt.Color;
```

or

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

Example : first applet code

```
import java.applet.Applet;
import java.awt.Graphics;

public class MyApplet extends Applet
{
public void paint(Graphics g)
{
g.drawString("This is my first applet", 60,40);
}
}
```

NB: Applets do not use main () method. This is because the applets are loaded when a web page is started.

Creating Web page to attach applet code.

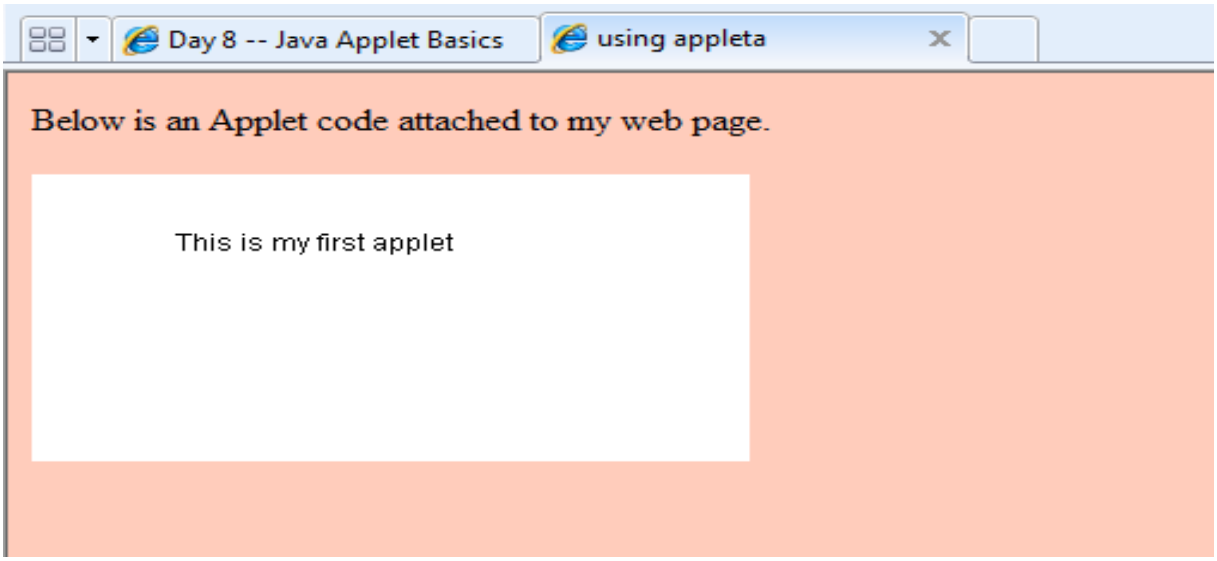
```
<html>
<head>
<title> using applets</title>
</head>
<body>
<p> Below is an Applet code attached to my web page.</p>
<applet code="MyApplet.class" width=300 eight=150></applet>
</body>
</html>
```

Compiling the applet source code

```
Javac MyApplet.java
```

To run the applet on web page:

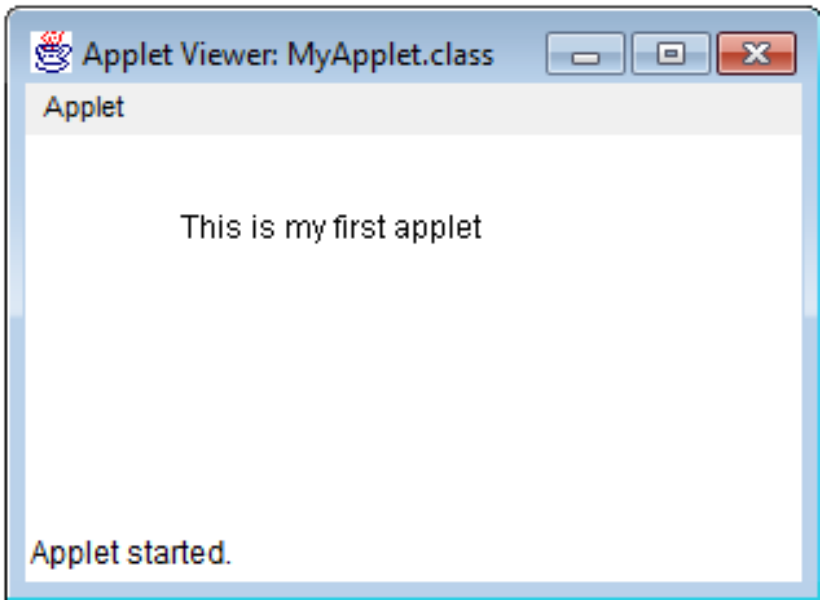
- Start the web page on which the applet is attached



To run applet alone – use the appletviewer tool on command prompt.

If using graphical interface e.g Textpad or Jcreater, then use Tools > Run Java applet.

```
Appletviewer myApplet.html
```



Explaining Applet code;

```
Public void paint(Graphics g)
{
}
```

- Painting- this is how applet draws something on the screen – text, line or image.
- Paint Takes argument – of type graphics instance.
- Object graphics holds graphics state – color, font.

```
g.drawString("This is my first applet", 60,40);
```

provides – (x,y) coordinates. It specifies the position to start printing the applet text on the screen.
60 – Pixels from left
40 – Pixels from top.

Understanding <APPLET> tag

```
<applet code="MyApplet.class" width=300 eight=150></applet>
```

code="MyApplet.class – specify the class file generated after compilation. It is the class file which is attached on the web page for execution.

width=300 eight=150 – specifies the dimension of the size of the applet to occupy on the web page

Using code base

It is used when the class files and the web pages to attach the applet are not in the same folder.

Codebase – specifies the sub folder containing the class files.

```
<applet code="MyApplet.class" codebase=”myclassfiles” width=300
eight=150></applet>
```

Major Applet Activities

- To create a basic Java application, your class has to have one method, `main()`, with a specific signature. Then, when your application runs, `main()` is found and executed, and from `main()` you can set up the behavior that your program needs to run. Applets are similar but more complicated-and, in fact, applets don't need a `main()` method at all. Applets have many different activities that correspond to various major events in the life cycle of the applet-for example, initialization, painting, and mouse events. Each activity has a corresponding method, so when an event occurs, the browser or other Java-enabled tool calls those specific methods.
- The default implementations of these activity methods do nothing; to provide behavior for an event you must override the appropriate method in your applet's subclass. You don't have to override all of them, of course; different applet behavior requires different methods to be overridden.
- The five most important methods in an applet's execution: initialization, starting, stopping, destroying, and painting.

Initialization

- Initialization occurs when the applet is first loaded (or reloaded), similarly to the `main()` method in applications. The initialization of an applet might include reading and parsing any parameters to the applet, creating any helper objects it needs, setting up an initial state, or loading images or fonts. To provide behavior for the initialization of your applet, override the `init()` method in your applet class:

```
public void init() {
    ...
}
```


Starting

- After an applet is initialized, it is started. Starting is different from initialization because it can happen many different times during an applet's lifetime, whereas initialization happens only once. Starting can also occur if the applet was previously stopped. For example, an applet is stopped if the reader follows a link to a different page, and it is started again when the reader returns to this page. To provide startup behavior for your applet, override the `start()` method:

```
public void start() {
    ...
}
```

- Functionality that you put in the `start()` method might include creating and starting up a thread to control the applet, sending the appropriate messages to helper objects, or in some way telling the applet to begin running

Stopping

- Stopping and starting go hand in hand. Stopping occurs when the reader leaves the page that contains a currently running applet, or you can stop the applet yourself by calling `stop()`. By default, when the reader leaves a page, any threads the applet had started will continue running. By overriding `stop()`, you can suspend execution of these threads and then restart them if the applet is viewed again:

```
public void stop() {
    ...
}
```

Destroying

- Destroying sounds more violent than it is. Destroying enables the applet to clean up after itself just before it is freed or the browser exits-for example, to stop and remove any running threads, close any open network connections, or release any other running objects. Generally, you won't want to override `destroy()` unless you have specific resources that need to be released-for example, threads that the applet has created. To provide clean-up behavior for your applet, override the `destroy()` method:

```
public void destroy() {
    ...
}
```

Technical Note

How is `destroy()` different from `finalize()`. First, `destroy()` applies only to applets. `finalize()` is a more general-purpose way for a single object of any type to clean up after itself.

Painting

- Painting is how an applet actually draws something on the screen, be it text, a line, a colored background, or an image. Painting can occur many thousands of times during an applet's life cycle (for example, after the applet is initialized, if the browser is placed behind another window on the screen and then brought forward again, if the browser window is moved to a different position on the screen, or perhaps repeatedly, in the case of animation).
- You override the `paint()` method if your applet needs to have an actual appearance on the screen (that is, most of the time). The `paint()` method looks like this:

```
public void paint(Graphics g) {
    ...
}
```

- Note that unlike the other major methods in this section, `paint()` takes an argument, an instance of the class `Graphics`. This object is created and passed to `paint` by the browser, so you don't have to worry about it. However, you will have to make sure that the `Graphics` class (part of the `java.awt` package) gets imported into your applet code, usually through an `import` statement at the top of your Java file:

```
import java.awt.Graphics;
```

A Simple Applet : applet to create a string text and apply font face, bold and size.

Listing : The Hello Again applet.

```
 1:  import java.awt.Graphics;
 2:  import java.awt.Font;
 3:  import java.awt.Color;
 4:
 5:  public class HelloAgainApplet extends java.applet.Applet {
 6:
 7:     Font f = new Font("TimesRoman", Font.BOLD, 36);
 8:
 9:     public void paint(Graphics g) {
10:        g.setFont(f);
11:        g.setColor(Color.red);
12:        g.drawString("Hello again!", 5, 40);
13:     }
14: }
```

Passing Parameters to Applets

With Java applications, you pass parameters to your `main()` routine by using arguments on the command line, or, for Macintoshes, in the Java Runner's dialog box. You can then parse those arguments inside the body of your class, and the application acts accordingly, based on the arguments it is given.

Applets, however, don't have a command line. How do you pass in different arguments to an applet? Applets can get different input from the HTML file that contains the `<APPLET>` tag through the use of applet parameters. To set up and handle parameters in an applet, you need two things:

- A special parameter tag in the HTML file
- Code in your applet to parse those parameters

Applet parameters come in two parts: a parameter name, which is simply a name you pick, and a value, which is the actual value of that particular parameter. So, for example, you can indicate the color of text in an applet by using a parameter with the name `color` and the value `red`. You can determine an animation's speed using a parameter with the name `speed` and the value `5`.

In the HTML file that contains the embedded applet, you indicate each parameter using the `<PARAM>` tag, which has two attributes for the name and the value, called (surprisingly enough) `NAME` and `VALUE`. The `<PARAM>` tag goes inside the opening and closing `<APPLET>` tags:

```
<APPLET CODE="MyApplet.class" WIDTH=100 HEIGHT=100>
<PARAM NAME=font VALUE="TimesRoman">
<PARAM NAME=size VALUE="36">
A Java applet appears here.</APPLET>
```

This particular example defines two parameters to the `MyApplet` applet: one whose name is `font` and whose value is `TimesRoman`, and one whose name is `size` and whose value is `36`.

Parameters are passed to your applet when it is loaded. In the `init()` method for your applet, you can then get hold of those parameters by using the `getParameter()` method. `getParameter()` takes one argument-a string representing the name of the parameter you're looking for-and returns a string containing the corresponding value of that parameter. (Like arguments in Java applications, all the parameter values are strings.) To get the value of the `font` parameter from the HTML file, you might have a line such as this in your `init()` method:

```
String theFontName = getParameter("font");
```

Note
The names of the parameters as specified in `<PARAM>` and the names of the parameters in `getParameter()` must match identically, including having the same case. In other words, `<PARAM NAME="name">` is different from `<PARAM NAME="Name">`. If your parameters are not being properly passed to your applet, make sure the parameter cases match.

Note that if a parameter you expect has not been specified in the HTML file, `getParameter()` returns `null`. Most often, you will want to test for a `null` parameter in your Java code and supply a reasonable default:

```
if (theFontName == null)
    theFontName = "Courier"
```

Keep in mind that `getParameter()` returns strings-if you want a parameter to be some other object or type, you have to convert it yourself. To parse the `size` parameter from that same HTML file and assign it to an integer variable called `theSize`, you might use the following lines:

```
int theSize;
String s = getParameter("size");
if (s == null)
    theSize = 12;
else theSize = Integer.parseInt(s);
```

Get it? Not yet? Let's create an example of an applet that uses this technique. You'll modify the Hello Again applet so that it says hello to a specific name, for example, `"Hello Bill"` or `"Hello Alice"`. The name is passed into the applet through an HTML parameter.

Let's start by copying the original `HelloAgainApplet` class and calling it `MoreHelloAgain` (see Listing below).

Listing :The More Hello Again applet.

```
 1:import java.awt.Graphics;
 2:import java.awt.Font;
 3:import java.awt.Color;
 4:
 5:public class MoreHelloApplet extends java.applet.Applet {
 6:
 7:    Font f = new Font("TimesRoman", Font.BOLD, 36);
 8:
```

```
 9:    public void paint(Graphics g) {
10:        g.setFont(f);
11:        g.setColor(Color.red);
12:        g.drawString("Hello Again!", 5, 40);
13:    }
14:}
```

The first thing you need to add to this class is a place to hold the name of the person you're saying hello to. Because you'll need that name throughout the applet, let's add an instance variable for the name, just after the variable for the font in line 7:

```
String name;
```

To set a value for the name, you have to get that parameter from the HTML file. The best place to handle parameters to an applet is inside an `init()` method. The `init()` method is defined similarly to `paint()` (`public`, with no arguments, and a return type of `void`). Make sure when you test for a parameter that you test for a value of `null`. The default, in this case, if a name isn't indicated, is to say hello to `"Laura"`. Add the `init()` method in between your instance variable definitions and the definition for `paint()`, just before line 9:

```
public void init() {
    name = getParameter("name");
    if (name == null)
        name = "Laura";
}
```

Now that you have the name from the HTML parameters, you'll need to modify it so that it's a complete string-that is, to tack the word `Hello` with a space onto the beginning, and an exclamation point onto the end. You could do this in the `paint()` method just before printing the string to the screen, but that would mean creating a new string every time the applet is painted. It would be much more efficient to do it just once, right after getting the name itself, in the `init()` method. Add this line to the `init()` method just before the last brace:

```
name = "Hello " + name + "!";
```

And now, all that's left is to modify the `paint()` method to use the new name parameter. The original `drawString()` method looked like this:

```
g.drawString("Hello Again!", 5, 40);
```

To draw the new string you have stored in the `name` instance variable, all you need to do is substitute that variable for the literal string:

```
g.drawString(name, 5, 40);
```

Listing 8.4 shows the final result of the `MoreHelloApplet` class. Compile it so that you have a class file ready.

Listing 8.4. The `MoreHelloApplet` class.

```
 1:  import java.awt.Graphics;
 2:  import java.awt.Font;
 3:  import java.awt.Color;
 4:
 5:  public class MoreHelloApplet extends java.applet.Applet {
 6:
 7:     Font f = new Font("TimesRoman", Font.BOLD, 36);
 8:     String name;
 9:
10:     public void init() {
11:         name = getParameter("name");
12:         if (name == null)
13:             name = "Laura";
14:
15:         name = "Hello " + name + "!";
16:     }
17:
18:     public void paint(Graphics g) {
19:         g.setFont(f);
20:         g.setColor(Color.red);
21:         g.drawString(name, 5, 40);
22:     }
23: }
```

Now let's create the HTML file that contains this applet. Listing 8.5 shows a new Web page for the `MoreHelloApplet` applet.

Listing 8.5. The HTML file for the `MoreHelloApplet` applet.

```
 1:  <HTML>
 2:  <HEAD>
 3:  <TITLE>Hello!</TITLE>
 4:  </HEAD>
 5:  <BODY>
 6:  <P>
 7:  <APPLET CODE="MoreHelloApplet.class" WIDTH=200 HEIGHT=50>
 8:  <PARAM NAME=name VALUE="Bonzo">
 9:  Hello to whoever you are!
10: </APPLET>
11: </BODY>
12: </HTML>
```

Analysis
Note the `<APPLET>` tag, which points to the class file for the applet and has the appropriate width and height (`200` and `50`). Just below it (line 8) is the `<PARAM>` tag, which you use to pass in the value for the name. Here, the `NAME` parameter is simply name, and the `VALUE` is the string `"Bonzo"`.

Let's try a second example. Remember that in the code for `MoreHelloApplet`, if no name is specified in a parameter, the default is the name `Laura`. Listing 8.6 creates an HTML file with no parameter tag for `name`.

Listing : Another HTML file for the `MoreHelloApplet` applet.

```
 1: <HTML>
 2: <HEAD>
 3: <TITLE>Hello!</TITLE>
 4: </HEAD>
 5: <BODY>
 6: <P>
 7: <APPLET CODE="MoreHelloApplet.class" WIDTH=200 HEIGHT=50>
 8: Hello to whoever you are!
 9: </APPLET>
10: </BODY>
11: </HTML>
```

Here, because no name was supplied, the applet uses the default, and the result is what you might expect.

Drawing and Filling Objects

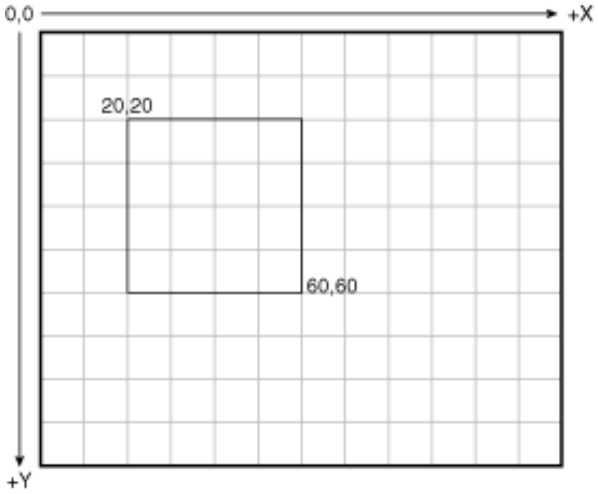
The `Graphics` Class

- With the basic graphics capabilities built into Java's class libraries, you can draw *lines, shapes, characters, and images* to the screen inside your applet.
- Most of the graphics operations in Java are methods defined in the `Graphics` class.
- You don't have to create an instance of `Graphics` in order to draw something in your applet; in your applet's `paint()` method (which you learned about yesterday), you are given a `Graphics` object.
- By drawing on that object, you draw onto your applet and the results appear onscreen.
- The `Graphics` class is part of the `java.awt` package, so if your applet does any painting (as it usually will), make sure you import that class at the beginning of your Java file:

```
import java.awt.Graphics;

public class MyClass extends java.applet.Applet {
...
}
```

The Graphics Coordinate System

- To draw an object on the screen, you call one of the drawing methods available in the `Graphics` class.
- All the drawing methods have arguments representing endpoints, corners, or starting locations of the object as values in the applet's coordinate system-for example, a line starts at the point `10,10` and ends at the point `20,20`.
- Java's coordinate system has the origin (`0,0`) in the top-left corner. Positive `x` values are to the right and positive `y` values are down. All pixel values are integers; there are no partial or fractional pixels. Figure 9.1 shows how you might draw a simple square by using this coordinate system.
- Java's coordinate system is different from that of many painting and layout programs, which have their `x` and `y` in the bottom left. If you're not used to working with this upside-down graphics system, it may take some practice to get familiar with it.



Drawing and Filling

The `Graphics` class provides a set of simple built-in graphics primitives for drawing, including lines, rectangles, polygons, ovals, and arcs

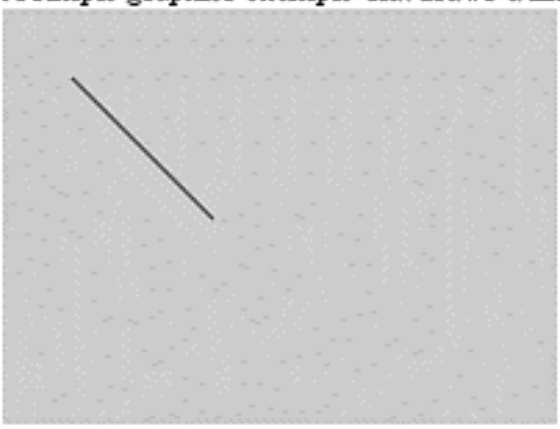
Lines

- To draw straight lines, use the **`drawLine()`** method. `drawLine()` takes four arguments: the `x` and `y` coordinates of the starting point and the `x` and `y` coordinates of the ending point. So, for example, the following `MyLine` class draws a line from the point `25,25` to the point `75,75`.
- Note that the `drawLine()` method is defined in the `Graphics` class (as are all the other graphics methods you'll learn about today). Here we're using that method for the current graphics context stored in the variable `g`:

```
import java.awt.Graphics;

public class MyLine extends java.applet.Applet {
    public void paint(Graphics g)
{
        g.drawLine(25,25,75,75);
    }
}
```

Figure below shows how the simple `MyLine` class looks in a Java-enabled browser such as Netscape.

A simple graphics example that draws a line:



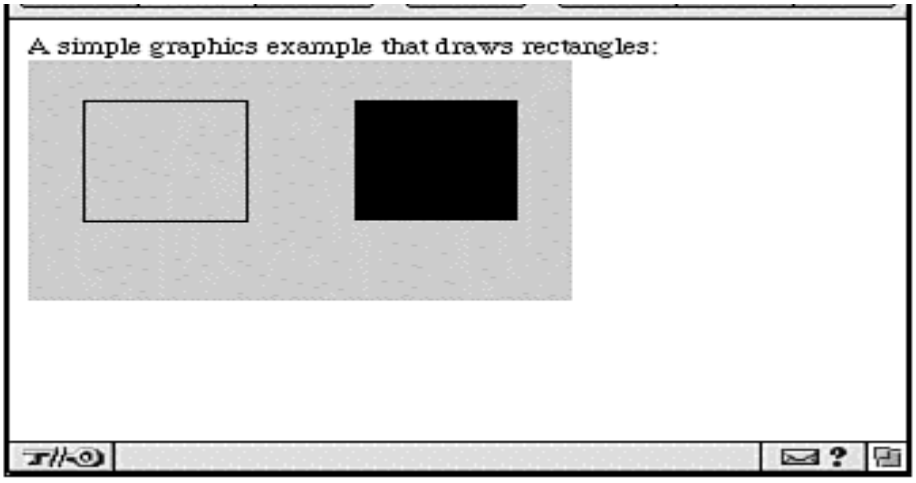
Rectangles

- The Java graphics primitives provide not just one, but three kinds of rectangles:
    - Plain rectangles
    - Rounded rectangles, which are rectangles with rounded corners
    - Three-dimensional rectangles, which are drawn with a shaded border
- For each of these rectangles, you have two methods to choose from: one that draws the rectangle in outline form and one that draws the rectangle filled with color.
- To draw a plain rectangle, use either the `drawRect()` or `fillRect()` methods. Both take ***four arguments:*** the `x` and `y` coordinates of the top-left corner of the rectangle, and the *width and height* of the rectangle to draw. For example, the following class (`MyRect`) draws two squares: The left one is an outline and the right one is filled (Figure 9.3 shows the result):

```
import java.awt.Graphics;

public class MyRect extends java.applet.Applet {
    public void paint(Graphics g) {
        g.drawRect(20,20,60,60);
        g.fillRect(120,20,60,60);
    }
}
```

- Rounded rectangles are, as you might expect, rectangles with rounded corners. The `drawRoundRect()` and `fillRoundRect()` methods to draw rounded rectangles are similar to regular rectangles except that rounded rectangles have two extra arguments for the *width and height of the angle of the corners*.

- Those two arguments determine how far along the edges of the rectangle the arc for the corner will start; the first for the angle along the horizontal plane, the second for the vertical.
- Larger values for the angle width and height make the overall rectangle more rounded; values equal to the width and height of the rectangle itself produce a circle. Figure 9.4 shows some examples of rounded corners.
- The following is a `paint()` method inside a class called `MyRRect` that draws two rounded rectangles: one as an outline with a rounded corner `10` pixels square; the other, filled, with a rounded corner `20` pixels square.




```
import java.awt.Graphics;

public class MyRRect extends java.applet.Applet {
    public void paint(Graphics g) {
        g.drawRoundRect(20,20,60,60,10,10);
        g.fillRoundRect(120,20,60,60,20,20);
    }
}
```

- Finally, there are three-dimensional rectangles. These rectangles aren't really 3D; instead, they have a slight shadow effect that makes them appear either raised or indented from the surface of the applet.
- Three-dimensional rectangles have four arguments for the `x` and `y` of the start position and the width and height of the rectangle.
- The fifth argument is a boolean indicating whether the 3D effect is to raise the rectangle (`true`) or indent it (`false`). As with the other rectangles, there are also different methods for drawing and filling: `draw3DRect()` and `fill3DRect()`.
- The following is a class called `My3DRect`, which produces two 3D squares-the left one raised, the right one indented.

```
import java.awt.Graphics;

public class My3DRect extends java.applet.Applet {
    public void paint(Graphics g) {
        g.draw3DRect(20,20,60,60,true);
        g.draw3DRect(120,20,60,60,false);
    }
}
```

Polygons

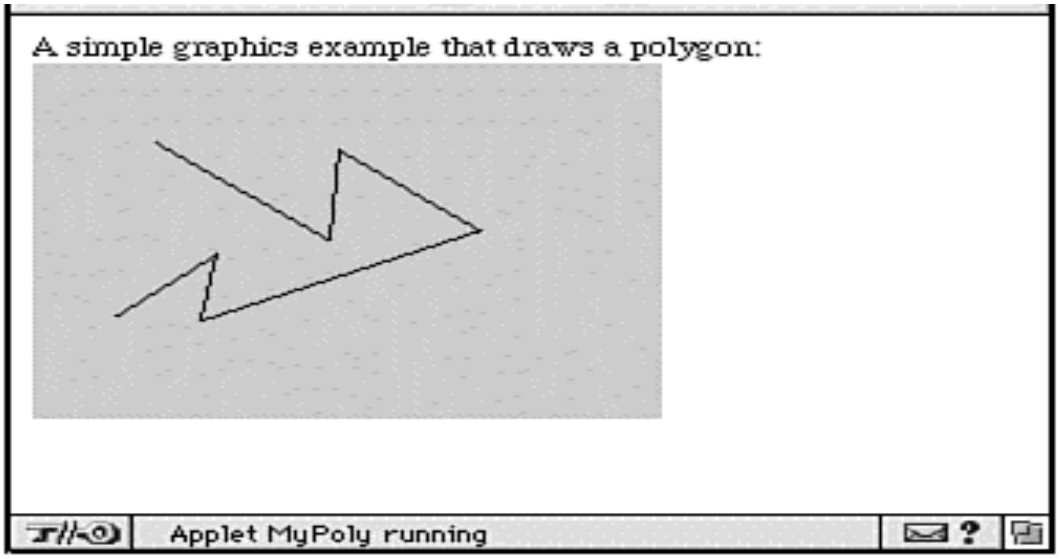
- Polygons are shapes with an unlimited number of sides. To draw a polygon, you need a set of `x` and `y` coordinates. The polygon is then drawn as a set of straight lines from the first point to the second, the second to the third, and so on.
- As with rectangles, you can draw an outline or a filled polygon (using the `drawPolygon()` and `fillPolygon()` methods, respectively). You also have a choice of how you want to indicate the list of coordinates-either as arrays of `x` and `y` coordinates or as an instance of the `Polygon` class.
- Using the first way of drawing polygons, the `drawPolygon()` and `fillPolygon()` methods take three arguments:
    - An array of integers representing `x` coordinates
    - An array of integers representing `y` coordinates
    - An integer for the total number of points

The `x` and `y` arrays should, of course, have the same number of elements.

Here's an example of drawing a polygon's outline using this method.

import java.awt.Graphics;

```
public class MyPoly extends java.applet.Applet {
    public void paint(Graphics g) {
        int exes[] = { 39,94,97,142,53,58,26 };
        int whys[] = { 33,74,36,70,108,80,106 };
        int pts = exes.length;
        g.drawPolygon(exes,whys,pts);
    }
}
```



- Note that Java does not automatically close the polygon; if you want to complete the shape, you have to include the starting point of the polygon at the end of the array. Drawing a filled polygon, however, joins the starting and ending points.
- The second way of calling `drawPolygon()` and `fillPolygon()` is to use a `Polygon` object to store the individual points of the polygon. The `Polygon` class is useful if you intend to add points to the polygon or if you're building the polygon on-the-fly. Using the `Polygon` class, you can treat the polygon as an object rather than having to deal with individual arrays.

To create a polygon object, you can either first create an empty polygon:

```
Polygon poly = new Polygon();
```

or create a polygon from a set of points using integer arrays, as in the previous example:

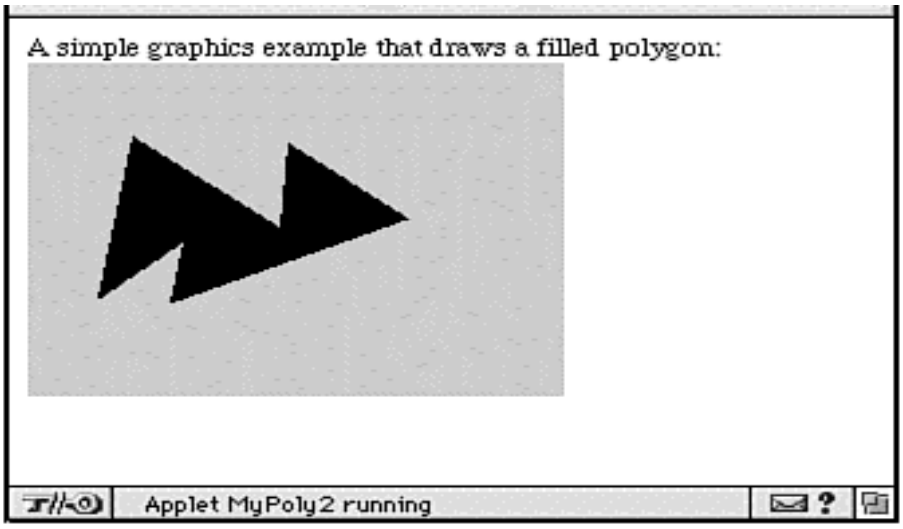
```
int exes[] = { 39,94,97,142,53,58,26 };
int whys[] = { 33,74,36,70,108,80,106 };
int pts = exes.length;
Polygon poly = new Polygon(exes,whys,pts);
```

Once you have a polygon object, you can add points to the polygon as you need to:

```
poly.addPoint(20,35);
```

Then, to draw the polygon, just use the polygon object as an argument to `drawPolygon()` or `fillPolygon()`. Here's that previous example, rewritten this time with a `Polygon` object. You'll also fill this polygon rather than just drawing its outline.

```
import java.awt.Graphics;
public class MyPoly2 extends java.applet.Applet {
    public void paint(Graphics g) {
        int exes[] = { 39,94,97,142,53,58,26 };
        int whys[] = { 33,74,36,70,108,80,106 };
        int pts = exes.length;
        Polygon poly = new Polygon(exes,whys,pts);
        g.fillPolygon(poly);
    }
}
```



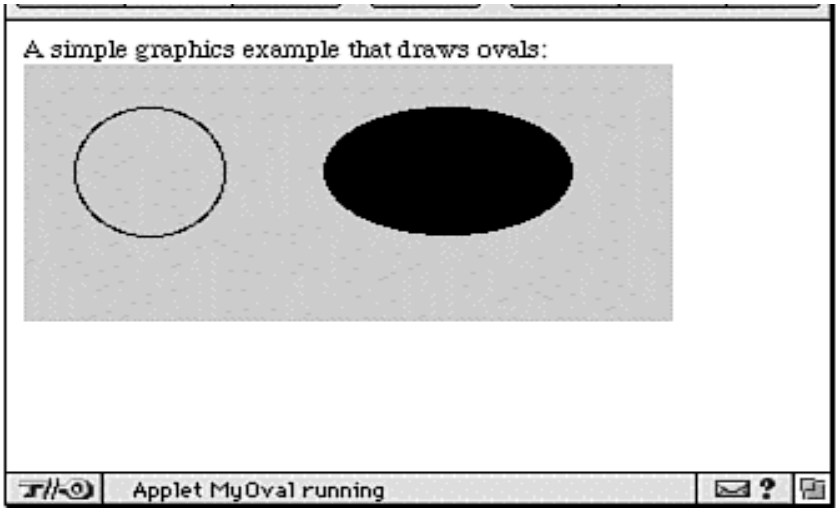
Ovals

You use ovals to draw ellipses or circles. Ovals are just like rectangles with overly rounded corners. You draw them using four arguments: the `x` and `y` of the top corner, and the width and height of the oval itself. Note that because you're drawing an oval, the starting point is some distance to the left and up from the actual outline of the oval itself. Again, if you think of it as a rectangle, it's easier to place.

As with the other drawing operations, the `drawOval()` method draws an outline of an oval, and the `fillOval()` method draws a filled oval.

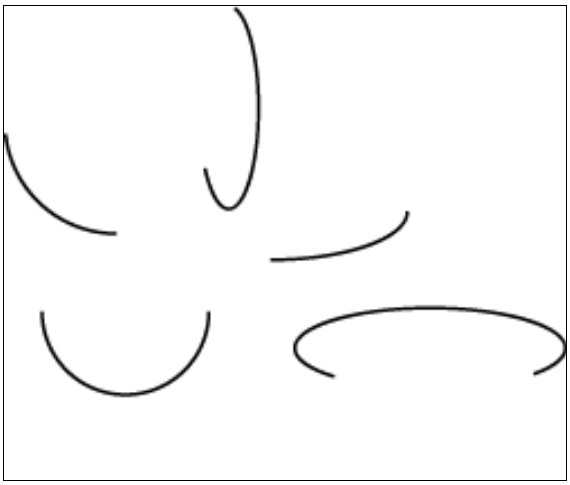
The following example draws two ovals-a circle and an ellipse (Figure below shows how these two ovals appear onscreen):

```
import java.awt.Graphics;

public class MyOval extends java.applet.Applet {
    public void paint(Graphics g) {
        g.drawOval(20,20,60,60);
        g.fillOval(120,20,100,60);
    }
}
```



Arcs

- Of all the shapes you can construct using methods in the `Graphics` class, arcs are the most complex to construct, which is why I saved them for last.
- An arc is a part of an oval; in fact, the easiest way to think of an arc is as a section of a complete oval. Figure shows some arcs.



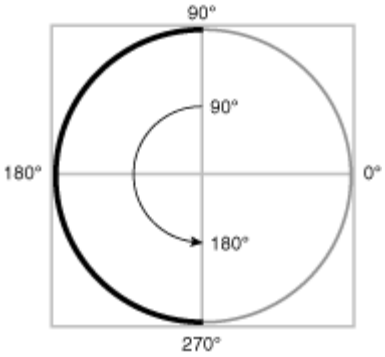
- The ***drawArc()*** method takes six arguments: the starting corner, the width and height, the angle at which to start the arc, and the degrees to draw it before stopping.
- Once again, there is a `drawArc` method to draw the arc's outline and the `fillArc()` method to fill the arc. Filled arcs are drawn as if they were sections of a pie; instead of joining the two endpoints, both endpoints are joined to the center of the circle.
- The important thing to understand about arcs is that you're actually formulating the arc as an oval and then drawing only some of that.
- The starting corner and width and height are not the starting point and width and height of the actual arc as drawn on the screen; they're the width and height of the full ellipse of which the arc is a part. Those first points determine the size and shape of the arc; the last two arguments (for the degrees) determine the starting and ending points.

Let's start with a simple arc, a C shape on a circle, as shown in Figure.

To construct the method to draw this arc, the first thing you do is think of it as a complete circle. Then you find the `x` and `y` coordinates and the width and height of that circle. Those four values are the first four arguments to the `drawArc()` or `fillArc()` methods. Figure 9.12 shows how to get those values from the arc.

To get the last two arguments, think in degrees around the circle, going counterclockwise. Zero degrees is at 3 o'clock, 90 degrees is at 12 o'clock, 180 at 9 o'clock, and 270 at 6 o'clock. The start of the arc is the degree value of the start of the arc. In this example, the starting point is the top of the C at 90 degrees; 90 is the fifth argument.

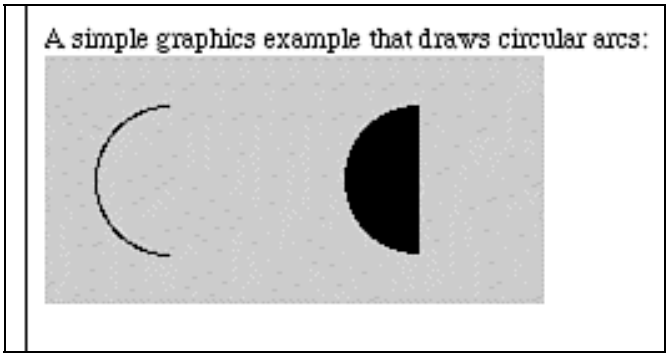
The sixth and last argument is another degree value indicating how far around the circle to sweep and the direction to go in (it's not the ending degree angle, as you might think). In this case, because you're going halfway around the circle, you're sweeping 180 degrees-and 180 is therefore the last argument in the arc. The important part is that you're sweeping 180 degrees counterclockwise, which is in the positive direction in Java. If you are drawing a backwards C, you sweep 180 degrees in the negative direction, and the last argument is `-180`. See Figure for the final illustration of how this works.



Note
It doesn't matter which side of the arc you start with. Because the shape of the arc has already been determined by the complete oval it's a section of, starting at either endpoint will work.

Here's the code for this example; you'll draw an outline of the C and a filled C to its right, as shown in Figure.

```
import java.awt.Graphics;

public class MyOval extends java.applet.Applet {
    public void paint(Graphics g) {
        g.drawArc(20,20,60,60,90,180);
        g.fillArc(120,20,60,60,90,180);
    }
}
```



Circles are an easy way to visualize arcs on circles; arcs on ellipses are slightly more difficult.

Like the arc on the circle, this arc is a piece of a complete oval, in this case, an elliptical oval. By completing the oval that this arc is a part of, you can get the starting points and the width and height arguments for the `drawArc()` or `fillArc()` method

Then all you need is to figure out the starting angle and the angle to sweep. This arc doesn't start on a nice boundary such as 90 or 180 degrees, so you'll need some trial and error. This arc starts somewhere around 25 degrees, and then sweeps clockwise about 130 degrees .

With all portions of the arc in place, you can write the code. Here's the Java code for this arc, both drawn and filled (note in the filled case how filled arcs are drawn as if they were pie sections):

```
import java.awt.Graphics;

public class MyOval extends java.applet.Applet {
    public void paint(Graphics g) {
        g.drawArc(10,20,150,50,25,-130);
        g.fillArc(10,80,150,50,25,-130);
    }
}
```

To summarize, here are the steps to take to construct arcs in Java:

1. Think of the arc as a slice of a complete oval.
2. Construct the full oval with the starting point and the width and height (it often helps to draw the full oval on the screen to get an idea of the right positioning).
3. Determine the starting angle for the beginning of the arc.
4. Determine how far to sweep the arc and in which direction (counterclockwise indicates positive values, clockwise indicates negative).

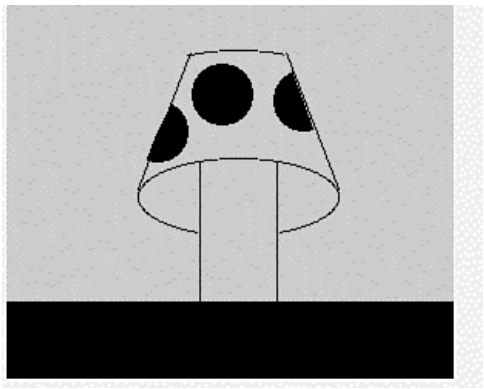
A Simple Graphics Example

Here's an example of an applet that uses many of the built-in graphics primitives to draw a rudimentary shape. In this case, it's a lamp with a spotted shade (or a sort of cubist mushroom, depending on your point of view). Listing 9.1 has the complete code for the lamp; Figure below shows the resulting applet.

Listing : The `Lamp` class.

```
1: import java.awt.*;
2:
3: public class Lamp extends java.applet.Applet {
```

```
 4:
 5:    public void paint(Graphics g) {
 6:        // the lamp platform
 7:        g.fillRect(0,250,290,290);
 8:
 9:        // the base of the lamp
10:        g.drawLine(125,250,125,160);
11:        g.drawLine(175,250,175,160);
12:
13:        // the lamp shade, top and bottom edges
14:         g.drawArc(85,157,130,50,-65,312);
15:         g.drawArc(85,87,130,50,62,58);
16:
17:         // lamp shade, sides
18:         g.drawLine(85,177,119,89);
19:         g.drawLine(215,177,181,89);
20:
21:         // dots on the shade
22:         g.fillArc(78,120,40,40,63,-174);
23:         g.fillOval(120,96,40,40);
24:         g.fillArc(173,100,40,40,110,180);
25:    }
26: }
```



Text and Fonts

- Using the `Graphics` class, you can also print text on the screen, in conjunction with the `Font` class (and, sometimes, the `FontMetrics` class). The `Font` class represents a given font-its name, style, and point size-and `FontMetrics` gives you information about that font (for example, the actual height or width of a given character) so that you can precisely lay out text in your applet.

- Note that the text here is drawn to the screen once and intended to stay there. You'll learn about entering text from the keyboard later this week.

Creating Font Objects

- To draw text to the screen, first you need to create an instance of the `Font` class. Font objects represent an individual font-that is, its name, style (bold, italic), and point size. Font names are strings representing the family of the font, for example, `"TimesRoman"`, `"Courier"`, or `"Helvetica"`. Font styles are constants defined by the `Font` class; you can get to them using class variables-for example, `Font.PLAIN`, `Font.BOLD`, or `Font.ITALIC`. Finally, the point size is the size of the font, as defined by the font itself; the point size may or may not be the height of the characters.
- To create an individual font object, use these three arguments to the `Font` class's `new` constructor:

```
Font f = new Font("TimesRoman", Font.BOLD, 24);
```

- This example creates a font object for the `TimesRoman BOLD` font, in `24` points. Note that like most Java classes, you have to import the `java.awt.Font` class before you can use it.

Tip
Font styles are actually integer constants that can be added to create combined styles; for example, `Font.BOLD + Font.ITALIC` produces a font that is both bold and italic.

- The fonts you have available to you in your applets depend on which fonts are installed on the system where the applet is running. If you pick a font for your applet and that font isn't available on the current system, Java will substitute a default font (usually Courier). You can get an array of the names of the current fonts available in the system using this bit of code:

```
String[] fontslist = this.getToolkit().getFontList();
```

- From this list, you can then often intelligently decide which fonts you want to use in your applet. For best results, however, it's a good idea to stick with standard fonts such as `"TimesRoman"`, `"Helvetica"`, and `"Courier"`.

Drawing Characters and Strings

- With a font object in hand, you can draw text on the screen using the methods `drawChars()` and `drawString()`. First, though, you need to set the current font to your font object using the `setFont()` method.
- The current font is part of the graphics state that is kept track of by the `Graphics` object on which you're drawing. Each time you draw a character or a string to the screen, Java draws that text in the current font. To change the font of the text, therefore, first change the current font. The following `paint()` method creates a new font, sets the current font to that font, and draws the string `"This is a big font."`, at the point `10,100`:

```
public void paint(Graphics g) {
    Font f = new Font("TimesRoman", Font.PLAIN, 72);
    g.setFont(f);
    g.drawString("This is a big font.", 10, 100);
}
```

.

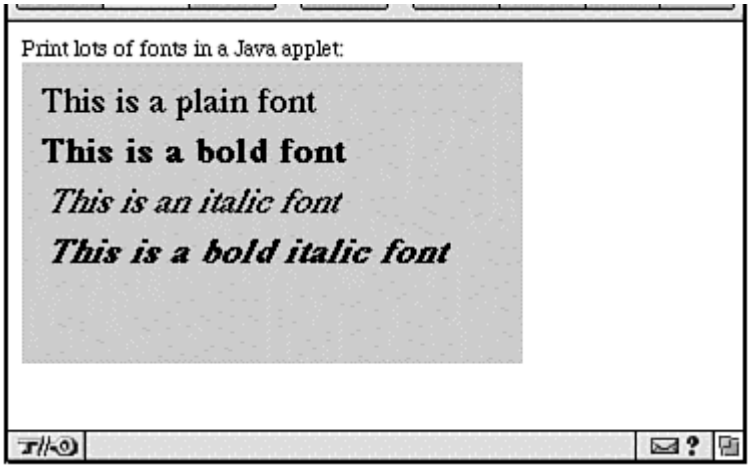
- The latter two arguments to `drawString()` determine the point where the string will start. The `x` value is the start of the leftmost edge of the text; `y` is the baseline for the entire string.
- Similar to `drawString()` is the `drawChars()` method that, instead of taking a string as an argument, takes an array of characters. `drawChars()` has five arguments: the array of characters, an integer representing the first character in the array to draw, another integer for the last character in the array to draw (all characters between the first and last are drawn), and the `x` and `y` for the starting point. Most of the time, `drawString()` is more useful than `drawChars()`.

Listing below shows an applet that draws several lines of text in different fonts; Figure 9.20 shows the result.

Listing : Many different fonts.

```
 1: import java.awt.Font;
 2: import java.awt.Graphics;
 3:
 4: public class ManyFonts extends java.applet.Applet {
 5:
 6:    public void paint(Graphics g) {
 7:        Font f = new Font("TimesRoman", Font.PLAIN, 18);
 8:        Font fb = new Font("TimesRoman", Font.BOLD, 18);
 9:        Font fi = new Font("TimesRoman", Font.ITALIC, 18);
10:        Font fbi = new Font("TimesRoman", Font.BOLD + Font.ITALIC, 18);
11:
```

```
12:         g.setFont(f);
13:         g.drawString("This is a plain font", 10, 25);
14:         g.setFont(fb);
15:         g.drawString("This is a bold font", 10, 50);
16:         g.setFont(fi);
17:         g.drawString("This is an italic font", 10, 75);
18:         g.setFont(fbi);
19:         g.drawString("This is a bold italic font", 10, 100);
20:     }
21:
22: }
```



Finding Out Information About a Font

Sometimes you may want to make decisions in your Java program based on the qualities of the current font-for example, its point size and the total height of its characters. You can find out some basic information about fonts and font objects by using simple methods on `Graphics` and on the `Font` objects. Table 9.1 shows some of these methods.

Table ; Font methods.

Method Name	*In Object*	*Action*
`getFont()`	`Graphics`	Returns the current font object as previously set by `setFont()`
`getName()`	`Font`	Returns the name of the font as a string
`getSize()`	`Font`	Returns the current font size (an integer)
`getStyle()`	`Font`	Returns the current style of the font (styles are integer constants: `0` is plain, `1` is bold, `2` is italic, `3` is bold italic)

`isPlain()`	`Font`	Returns `true` or `false` if the font's style is plain
`isBold()`	`Font`	Returns `true` or `false` if the font's style is bold
`isItalic()`	`Font`	Returns `true` or `false` if the font's style is italic

For more detailed information about the qualities of the current font (for example, the length or height of given characters), you need to work with font metrics. The `FontMetrics` class describes information specific to a given font: the leading between lines, the height and width of each character, and so on. To work with these sorts of values, you create a `FontMetrics` object based on the current font by using the applet method `getFontMetrics()`:

```
Font f = new Font("TimesRoman", Font.BOLD, 36);
FontMetrics fmetrics = getFontMetrics(f);
g.setfont(f);
```

Table 9.2 shows some of the things you can find out using font metrics. All these methods should be called on a `FontMetrics` object.

Table 9.2. Font metrics methods.

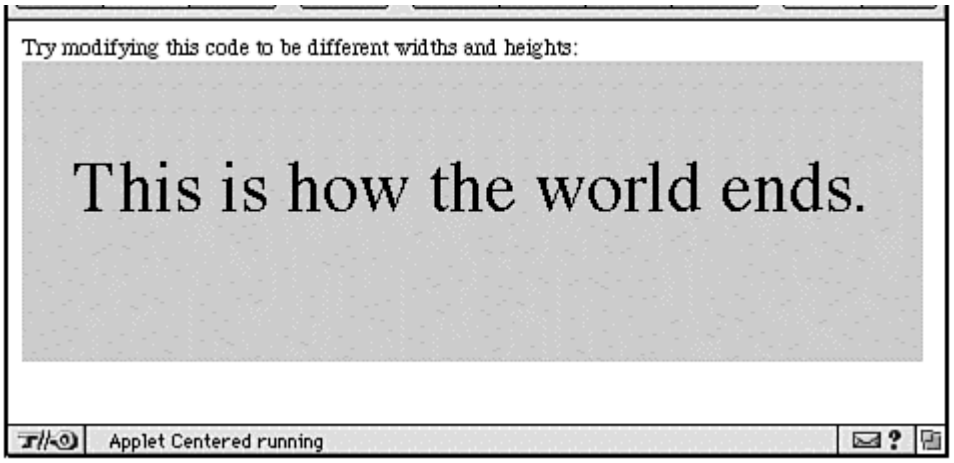
Method Name	*Action*
`stringWidth(string)`	Given a string, returns the full width of that string, in pixels
`charWidth(char)`	Given a character, returns the width of that character
`getAscent()`	Returns the ascent of the font, that is, the distance between the font's baseline and the top of the characters
`getDescent()`	Returns the descent of the font-that is, the distance between the font's baseline and the bottoms of the characters (for characters such as p and q that drop below the baseline)
`getLeading()`	Returns the leading for the font, that is, the spacing between the descent of one line and the ascent of another line
`getHeight()`	Returns the total height of the font, which is the sum of the ascent, descent, and leading value

As an example of the sorts of information you can use with font metrics, Listing 9.3 shows the Java code for an applet that automatically centers a string horizontally and vertically inside an applet. The centering position is different depending on the font and font size; by using font metrics to find out the actual size of a string, you can draw the string in the appropriate place.

Figure shows the result (which is less interesting than if you actually compile and experiment with various applet and font sizes).

Listing : Centering a string.

```
 1: import java.awt.Font;
 2: import java.awt.Graphics;
 3: import java.awt.FontMetrics;
 4:
 5: public class Centered extends java.applet.Applet {
 6:
 7:    public void paint(Graphics g) {
 8:        Font f = new Font("TimesRoman", Font.PLAIN, 36);
 9:        FontMetrics fm = getFontMetrics(f);
10:        g.setFont(f);
11:
12:        String s = "This is how the world ends.";
13:        int xstart = (size().width - fm.stringWidth(s)) / 2;
14:        int ystart = size().height / 2;
15:
16:        g.drawString(s, xstart, ystart);
17:    }
18:}
```



Analysis
Note the `size()` method in lines 13 and 14, which returns the width and height of the overall applet area as a `Dimension` object. You can then get to the individual width and height using the `width` and `height` instance variables of that `Dimension`, here by chaining the method call and the

variable name. Getting the current applet size in this way is a better idea than hard coding the size of the applet into your code; this code works equally well with an applet of any size.

Note also that the line of text, as shown in Figure isn't precisely vertically centered in the applet bounding box. This example centers the baseline of the text inside the applet; using the `getAscent()` and `getDescent()` methods from the `FontMetrics` class (to get the number of pixels from the baseline to the top of the characters and the number of pixels from the baseline to the bottom of the characters), you can figure out exactly the middle of the line of text.

Color

- Drawing black lines and text on a gray background is all very nice, but being able to use different colors is much nicer. Java provides methods and behaviors for dealing with color in general through the `Color` class, and also provides methods for setting the current foreground and background colors so that you can draw with the colors you created.
- Java's abstract color model uses 24-bit color, wherein a color is represented as a combination of red, green, and blue values. Each component of the color can have a number between `0` and `255`. `0,0,0` is black, `255,255,255` is white, and Java can represent millions of colors between as well.
- Java's abstract color model maps onto the color model of the platform Java is running on, which usually has only 256 or fewer colors from which to choose. If a requested color in a color object is not available for display, the resulting color may be mapped to another or dithered, depending on how the browser viewing the color implemented it, and depending on the platform on which you're running. In other words, although Java gives the capability of managing millions of colors, very few may actually be available to you in real life.

Using Color Objects

To draw an object in a particular color, you must create an instance of the `Color` class to represent that color. The `Color` class defines a set of standard color objects, stored in class variables, to quickly get a color object for some of the more popular colors. For example, `Color.red` returns a `Color` object representing red (RGB values of `255, 0`, and `0`), `Color.white` returns a white color (RGB values of `255`, `255`, and `255`), and so on. Table below shows the standard colors defined by variables in the `Color` class.

Table Standard colors.

Color Name	*RGB Value*
`Color.white`	`255,255,255`
`Color.black`	`0,0,0`
`Color.lightGray`	`192,192,192`
`Color.gray`	`128,128,128`
`Color.darkGray`	`64,64,64`
`Color.red`	`255,0,0`
`Color.green`	`0,255,0`
`Color.blue`	`0,0,255`
`Color.yellow`	`255,255,0`
`Color.magenta`	`255,0,255`
`Color.cyan`	`0,255,255`
`Color.pink`	`255,175,175`
`Color.orange`	`255,200,0`

If the color you want to draw in is not one of the standard `Color` objects, fear not. You can create a color object for any combination of red, green, and blue, as long as you have the values of the color you want. Just create a new color object:

```
Color c = new Color(140,140,140);
```

This line of Java code creates a color object representing a dark gray. You can use any combination of red, green, and blue values to construct a color object.

Alternatively, you can create a color object using three floats from `0.0` to `1.0`:

```
Color c = new Color(0.55,0.55,0.55);
```

Testing and Setting the Current Colors

To draw an object or text using a color object, you have to set the current color to be that color object, just as you have to set the current font to the font in which you want to draw. Use the `setColor()` method (a method for `Graphics` objects) to do this:

```
g.setColor(Color.green);
```

After you set the current color, all drawing operations will occur in that color.

In addition to setting the current color for the graphics context, you can also set the background and foreground colors for the applet itself by using the `setBackground()` and `setForeground()` methods. Both of these methods are defined in the `java.awt.Component` class, which `Applet`-and therefore your classes-automatically inherits.

The `setBackground()` method sets the background color of the applet, which is usually a light gray (to match the default background of the browser). It takes a single argument, a `Color` object:

```
setBackground(Color.white);
```

The `setForeground()` method also takes a single color as an argument, and it affects everything that has been drawn on the applet, regardless of the color in which it has been drawn. You can use `setForeground()` to change the color of everything in the applet at once, rather than having to redraw everything:

```
setForeground(Color.black);
```

In addition to the `setColor()`, `setForeground()`, and `setBackground()` methods, there are corresponding `get` methods that enable you to retrieve the current graphics color, background, or foreground. Those methods are `getColor()` (defined in `Graphics` objects), `getForeground()` (defined in `Applet`), and `getBackground()` (also in `Applet`). You can use these methods to choose colors based on existing colors in the applet:

```
setForeground(g.getColor());
```

A Simple Color Example

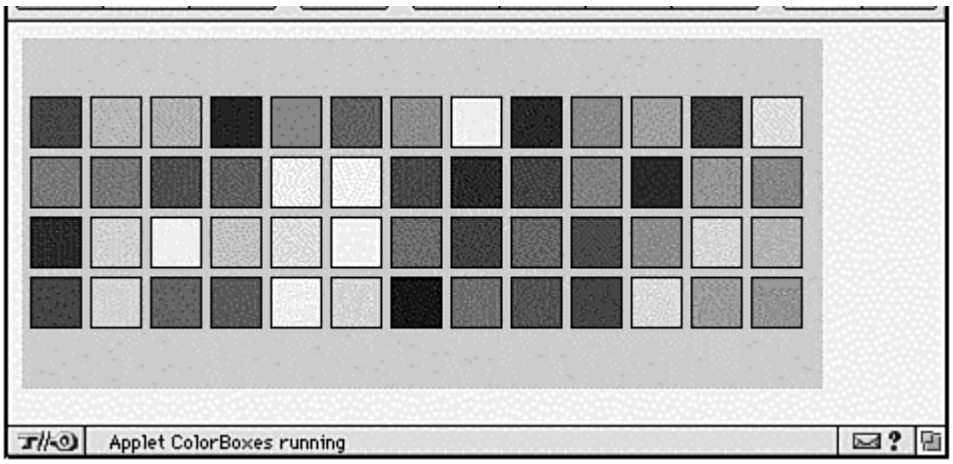
Listing below shows the code for an applet that fills the applet's drawing area with square boxes, each of which has a randomly chosen color in it. It's written so that it can handle any size of applet and automatically fill the area with the right number of boxes.

Listing Random color boxes.

```
 1:  import java.awt.Graphics;
 2:  import java.awt.Color;
 3:
 4:  public class ColorBoxes extends java.applet.Applet {
 5:
 6:      public void paint(Graphics g) {
 7:          int rval, gval, bval;
 8:
 9:          for (int j = 30; j < (size().height -25); j += 30)
10:             for (int i = 5; i < (size().width -25); i += 30) {
11:                 rval = (int)Math.floor(Math.random() * 256);
12:                 gval = (int)Math.floor(Math.random() * 256);
13:                 bval = (int)Math.floor(Math.random() * 256);
14:
15:                 g.setColor(new Color(rval,gval,bval));
16:                 g.fillRect(i, j, 25, 25);
17:                 g.setColor(Color.black);
18:                 g.drawRect(i-1, j-1, 25, 25);
19:             }
20:     }
21: }
```

Analysis
The two `for` loops are the heart of this example; the first one draws the rows, and the second draws the individual boxes within each row. When a box is drawn, the random color is calculated first, and then the box is drawn. A black outline is drawn around each box, because some of them tend to blend into the background of the applet.

Because this `paint` method generates new colors each time the applet is painted, you can regenerate the colors by moving the window around or by covering the applet's window with another one (or by reloading the page). Figure below shows the final applet (although given that this picture is black and white, you can't get the full effect of the multicolored squares).


Applet ColorBoxes running

Creating User Interfaces with the awt

An awt Overview

The awt provides the following:

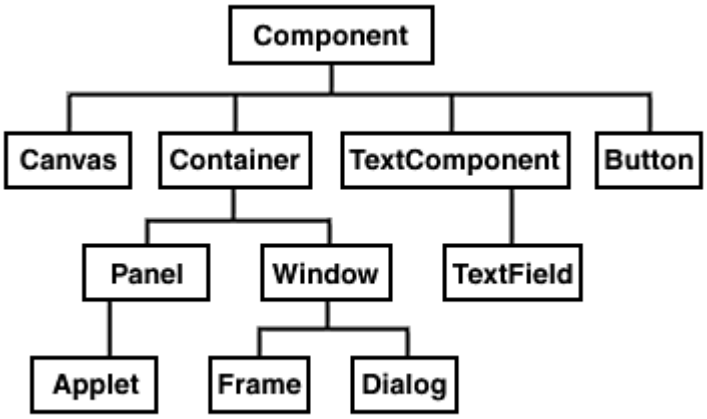
- A full set of user interface (UI) widgets and other components, including windows, menus, buttons, check boxes, text fields, scrollbars, and scrolling lists
- Support for UI containers, which can contain other embedded containers or UI widgets
- An event system for managing system and user events among parts of the awt
- Mechanisms for laying out components in a way that enables platform-independent UI design

- The basic idea behind the awt is that a graphical Java program is a set of nested components, starting from the outermost window all the way down to the smallest UI component.
- Components can include things you can actually see on the screen, such as windows, menu bars, buttons, and text fields, and they can also include containers, which in turn can contain other components
- This nesting of components within containers within other components creates a hierarchy of components, from the smallest check box inside an applet to the overall window on the screen.
- The hierarchy of components determines the arrangement of items on the screen and inside other items, the order in which they are painted, and how events are passed from one component to another.
- These are the major components you can work with in the awt:

  - *Containers.* Containers are generic awt components that can contain other components, including other containers. The most common form of container is the *panel*, which represents a container that can be displayed onscreen. Applets are a form of panel (in fact, the `Applet` class is a subclass of the `Panel` class).
  - *Canvases.* A canvas is a simple drawing surface. Although you can draw on panels (as you've been doing all along), canvases are good for painting images or performing other graphics operations.
  - *UI components.* These can include buttons, lists, simple pop-up menus, check boxes, test fields, and other typical elements of a user interface.
  - *Window construction components.* These include windows, frames, menu bars, and dialog boxes. They are listed separately from the other UI components because you'll use these less often-

particularly in applets. In applets, the browser provides the main window and menu bar, so you don't have to use these. Your applet may create a new window, however, or you may want to write your own Java application that uses these components. (You'll learn about these tomorrow.)

- The classes inside the `java.awt` package are written and organized to mirror the abstract structure of containers, components, and individual UI components.
- Figure below shows some of the class hierarchy that makes up the main classes in the awt.



- The root of most of the awt components is the class `Component`, which provides basic display and event-handling features.
- The classes `Container`, `Canvas`, `TextComponent`, and many of the other UI components inherit from `Component`.
- Inheriting from the `Container` class are objects that can contain other awt components-the `Panel` and `Window` classes, in particular.
- Note that the `java.applet.Applet` class, even though it lives in its own package, inherits from `Panel`, so your applets are an integral part of the hierarchy of components in the awt system.

- A graphical user interface-based application that you write by using the awt can be as complex as you like, with dozens of nested containers and components inside each other.
- The awt was designed so that each component can play its part in the overall awt system without needing to duplicate or keep track of the behavior of other parts in the system.
- **Layout Managers**: In addition to the components themselves, the awt also includes a set of layout managers. Layout managers determine how the various components are arranged when they are

displayed onscreen, and their various sizes relative to each other. Because Java applets and applications that use the awt can run on different systems with different displays, different fonts, and different resolutions, you cannot just stick a particular component at a particular spot on the window. Layout managers help you create UI layouts that are dynamically arranged and can be displayed anywhere the applet or application might be run.

The Basic User Interface Components

- The simplest form of awt component is the basic UI component.

Basic UI components: ***labels, buttons, check boxes, choice menus, and text fields***.

Procedure for creating the component

- First create the component and
- Then add it to the panel that holds it, at which point it is displayed on the screen.
- To add a component to a panel (such as your applet, for example), use the

`add()` method:

```
public void init() {
    Button b = new Button("OK");
    add(b);
}
```

OR

Create the component and add to the panel at the same time;

```
add(new Button("OK"));
```

Labels

- The simplest form of UI component is the label, which is, effectively, a text string that you can use to label other UI components.
- Labels are not editable; they just label other components on the screen.

A *label* is *an uneditable text string that acts as a description for other awt components.*

To create a label, use one of the following constructors:

- **`Label()`** creates an empty label, with its text aligned left.
- **`Label(String)`** creates a label with the given text string, also aligned left.
- **`Label(String, int)`** creates a label with the given text string and the given alignment. The available alignment numbers are stored in class variables in `Label`, making them easier to remember: `Label.RIGHT`, `Label.LEFT`, and `Label.CENTER`.

```
import java.awt.*;

public class LabelTest extends java.applet.Applet {

  public void init() {
    setFont(new Font ("Helvetica", Font.BOLD, 14));
    setLayout(new GridLayout(3,1));
    add(new Label("aligned left", Label.LEFT));
    add(new Label("aligned center", Label.CENTER));
    add(new Label("aligned right", Label.RIGHT));
  }
}
```

Buttons

- Buttons are simple UI components that trigger some action in your interface when they are pressed.
- For example, a calculator applet might have buttons for each number and operator, or a dialog box might have buttons for OK and Cancel.
- A *button* is a UI component that, when "pressed" (selected) with the mouse, triggers some action.

To create a button, use one of the following constructors:

- **`Button()`** creates an empty button with no label.
- **`Button(String)`** creates a button with the given string as a label.

```
public class ButtonTest extends java.applet.Applet {

  public void init() {
    add(new Button("Rewind"));
    add(new Button("Play"));
    add(new Button("Fast Forward"));
    add(new Button("Stop"));
  }
}
```

Check Boxes

- *Check boxes* are user-interface components that have two states: on and off (or checked and unchecked, selected and unselected, true and false, and so on).
- Unlike buttons, check boxes usually don't trigger direct actions in a UI, but instead are used to indicate optional features of some other action.

Check boxes can be used in two ways:

- Nonexclusive: Given a series of check boxes, any of them can be selected.
- Exclusive: Given a series, only one check box can be selected at a time.

The latter kind of check boxes are called radio buttons or check box groups, and are described in the next section.

Check boxes are UI components that can be selected or deselected (checked or unchecked) to provide options. Nonexclusive check boxes can be checked or unchecked independently of other check boxes. Exclusive check boxes, sometimes called *radio buttons*, exist in groups; only one in the group can be checked at one time.

Nonexclusive check boxes can be created by using the `Checkbox` class. You can create a check box using one of the following constructors:

- **`Checkbox()`** creates an empty check box, unselected.
- **`Checkbox(String)`** creates a check box with the given string as a label.

- **`Checkbox(String, null, boolean)`** creates a check box that is either selected or deselected based on whether the boolean argument is `true` or `false`, respectively. (The `null` is used as a placeholder for a group argument. Only radio buttons have groups, as you'll learn in the next section.)

Figure 13.5 shows a few simple check boxes (only `Underwear` is selected) generated using the following code:

```
import java.awt.*;

public class CheckboxTest extends java.applet.Applet {

  public void init() {
    setLayout(new FlowLayout(FlowLayout.LEFT));
    add(new Checkbox("Shoes"));
    add(new Checkbox("Socks"));
    add(new Checkbox("Pants"));
    add(new Checkbox("Underwear", null, true));
    add(new Checkbox("Shirt"));
  }

}
```

Radio Buttons

- Radio buttons have the same appearance as check boxes, but only one in a series can be selected at a time.
- To create a series of radio buttons, first create an instance of **`CheckboxGroup`**:

```
CheckboxGroup cbg = new CheckboxGroup();
```

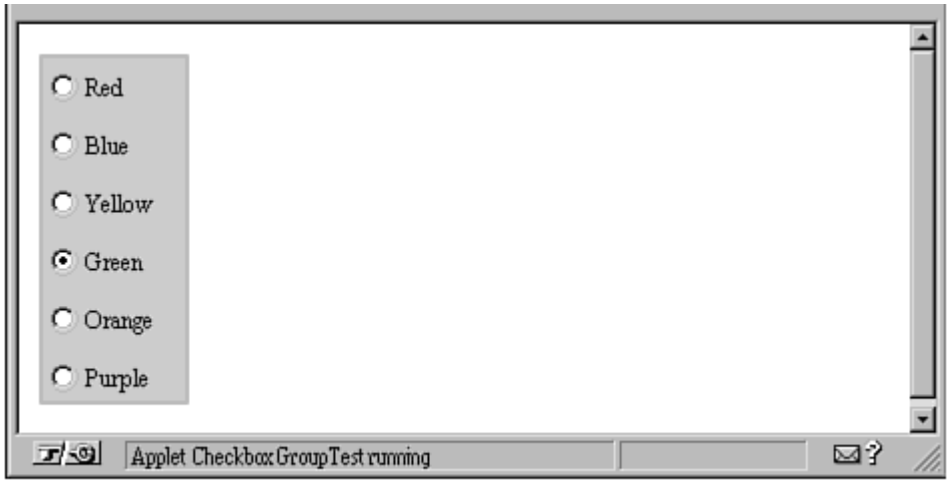
- Then create and add the individual check boxes using the constructor with **three arguments**
  - the first is the label,
  - second is the group, and
  - the third is whether that check box is selected).

- Note that because radio buttons, by definition, have only one in the group selected at a time, the last `true` to be added will be the one selected by default:

```
add(new Checkbox("Yes", cbg, true);
add(new Checkbox("No", cbg, false);
```

Here's a simple example (the results of which are shown in Figure 13.6):

```
import java.awt.*;

public class CheckboxGroupTest extends java.applet.Applet {

  public void init() {
    setLayout(new FlowLayout(FlowLayout.LEFT));
    CheckboxGroup cbg = new CheckboxGroup();

    add(new Checkbox("Red", cbg, false));
    add(new Checkbox("Blue", cbg, false));
    add(new Checkbox("Yellow", cbg, false));
    add(new Checkbox("Green", cbg, true));
    add(new Checkbox("Orange", cbg, false));
    add(new Checkbox("Purple", cbg, false));
  }

}
```



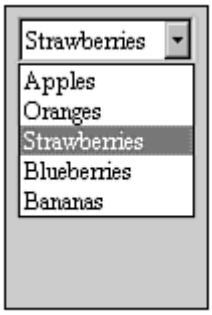
Choice Menus

- Choice menus are pop-up (or pull-down) menus from which you can select an item.
- The menu then displays that choice on the screen. The function of a choice menu is the same across platforms, but its actual appearance may vary from platform to platform.
- Note that choice menus can have ***only one item selected at a time***.
- If you want to be able to choose multiple items from the menu, use a scrolling list instead .
- *Choice menus* are pop-up menus of items from which you can choose one item.

- To create a choice menu,
  - create an ***instance of the*** `Choice` class and then
  - use the ***`addItem()` method*** to add individual items to it in the order in which they should appear.
  - Finally, ***add the entire choice menu to the panel in the usual way***.

```
import java.awt.*;

public class ChoiceTest extends java.applet.Applet {

  public void init() {
    Choice c = new Choice();

    c.addItem("Apples");
    c.addItem("Oranges");
    c.addItem("Strawberries");
    c.addItem("Blueberries");
    c.addItem("Bananas");

    add(c);
  }
}
```

- Even after your choice menu has been added to a panel, you can continue to add items to that menu with the `addItem()` method



Text Fields

- Text fields allow you to enter and edit text.
- Text fields are generally only a single line and do not have scrollbars;
- Text areas, are better for larger amounts of text.
- Text fields are different from labels in that they can be edited; labels are good for just displaying text, text fields for getting text input from the user.

Text fields provide an area where you can enter and edit a single line of text.

To create a text field, use one of the following constructors:

- ***TextField()*** creates an empty `TextField` that is 0 characters wide (it will be resized by the current layout manager).
- ***TextField(int)*** creates an empty text field. The integer argument indicates the minimum number of characters to display.
- ***TextField(String)*** creates a text field initialized with the given string. The field will be automatically resized by the current layout manager.
- ***TextField(String, int)*** creates a text field some number of characters wide (the integer argument) containing the given string. If the string is longer than the width, you can select and drag portions of the text within the field, and the box will scroll left or right.

For example, the following line creates a text field 30 characters wide with the string `"Enter Your Name"` as its initial contents:

```
TextField tf = new TextField("Enter Your Name", 30);
add(tf);
```

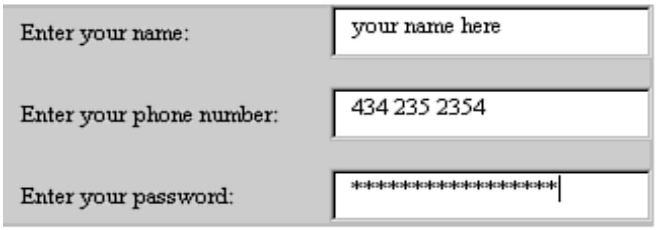
Tip
Text fields include only the editable field itself. You usually need to include a label with a text field to indicate what belongs in that text field.

You can also create a text field that obscures the characters typed into it-for example, for password fields. To do this, first create the text field itself; then use the `setEchoCharacter()` method to set the character that is echoed on the screen. Here is an example:

```
TextField tf = new TextField(30);
tf.setEchoCharacter('*');
```

Figure 13.8 shows three text boxes (and labels) that were created using the following code:

```
add(new Label("Enter your Name"));
add(new TextField("your name here", 45));
add(new Label("Enter your phone number"));
add(new TextField(12));
add(new Label("Enter your password"));
TextField t = new TextField(20);
t.setEchoCharacter('*');
add(t);
```



The text in the first field (`your name here`) was initialized in the code; I typed the text in the remaining two boxes just before taking the snapshot.

Text fields inherit from the class `TextComponent` and have a whole suite of methods, both inherited from that class and defined in their own class, that may be useful to you in your Java programs..

Panels and Layout

- awt panels can contain UI components or other panels. The question now is how those components are actually arranged and displayed onscreen.
- In other windowing systems, UI components are often arranged using hard-coded pixel measurements-put a text field at the position `10,30`, for example-the same way you used the graphics operations to paint squares and ovals on the screen.
- In the awt, your UI design may be displayed on many different window systems on many different screens and with many different kinds of fonts with different font metrics. Therefore, you need a more flexible method of arranging components on the screen so that a layout that looks nice on one platform isn't a jumbled, unusable mess on another.
- For just this purpose, Java has layout managers, insets, and hints that each component can provide to help dynamically lay out the screen.

Layout Managers: An Overview

- The actual appearance of the awt components on the screen is usually determined by two things:
    - how those components are added to the panel that holds them (either the order or through arguments to `add()`) and

- the layout manager that panel is currently using to lay out the screen. The layout manager determines how portions of the screen will be sectioned and how components within that panel will be placed.

- The *layout manager* determines how awt components are dynamically arranged on the screen.
- Each panel on the screen can have its own layout manager. By nesting panels within panels, and using the appropriate layout manager for each one, you can often arrange your UI to group and arrange components in a way that is functionally useful and that looks good on a variety of platforms and windowing systems.

The awt provides five basic layout managers:

- `FlowLayout`,
- `GridLayout`,
- `BorderLayout`,
- `CardLayout`, and
- `GridBagLayout`.

- To create a layout manager for a given panel, create an instance of that layout manager and then use the `setLayout()` method for that panel.
- This example sets the layout manager of the entire enclosing applet panel:

```
public void init() {
    setLayout(new FlowLayout());
}
```

- Setting the default layout manager, like creating user-interface components, is best done during the applet's initialization, which is why it's included here.
- After the layout manager is set, you can start adding components to the panel.
- The order in which components are added or the arguments you use to add those components is often significant, depending on which layout manager is currently active.

The following sections describe the five basic Java awt layout managers.

The `FlowLayout` Class

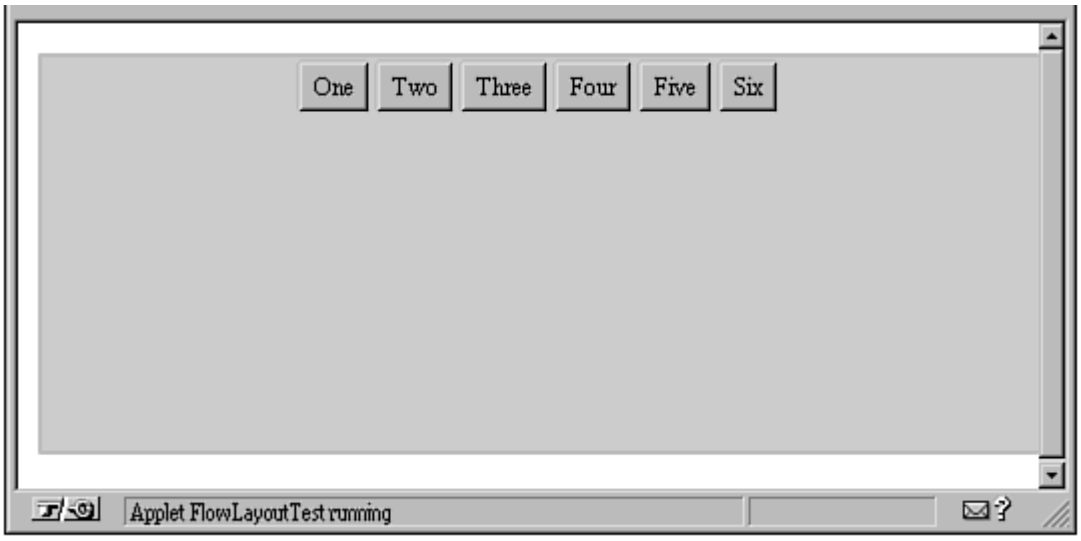
- The `FlowLayout` class is the most basic of layouts.

- Using flow layout, components are added to the panel one at a time, row by row. If a component doesn't fit onto a row, it's wrapped onto the next row.
- The flow layout also has an alignment, which determines the alignment of each row. By default, each row is centered.
- *Flow layout* arranges components from left to right in rows. The rows are aligned left, right, or centered.
- To create a basic flow layout with a centered alignment, use the following line of code in your panel's initialization (because this is the default pane layout, you don't need to include this line if that is your intent):

```
setLayout(new FlowLayout());
```

With the layout set, the order in which you add elements to the layout determines their position. The following code creates a simple row of six buttons in a centered flow layout.

```
import java.awt.*;

public class FlowLayoutTest extends java.applet.Applet {

  public void init() {
    setLayout(new FlowLayout());
    add(new Button("One"));
    add(new Button("Two"));
    add(new Button("Three"));
    add(new Button("Four"));
    add(new Button("Five"));
    add(new Button("Six"));
  }
}
```



To create a flow layout with an alignment other than centered, add the `FlowLayout.RIGHT` or `FlowLayout.LEFT` class variable as an argument:

```
setLayout(new FlowLayout(FlowLayout.LEFT));
```

You can also set horizontal and vertical gap values by using flow layouts. The *gap* is the number of pixels between components in a panel; by default, the horizontal and vertical gap values are three pixels, which can be very close indeed. Horizontal gap spreads out components to the left and to the right; vertical gap spreads them to the top and bottom of each component. Add integer arguments to the flow layout constructor to increase the gap.

Figure shows the result of adding a gap of 30 points in the horizontal and 10 in the vertical directions, like this:

```
setLayout(new FlowLayout(FlowLayout.LEFT, 30, 10));
```

Grid Layouts

- *Grid layouts* offer more control over the placement of components inside a panel.
- Using a grid layout, you portion off the display area of the panel into rows and columns.
- Each component you then add to the panel is placed in a *cell* of the grid, starting from the top row and progressing through each row from left to right (here's where the order of calls to the `add()` method are very relevant to how the screen is laid out).
- To create a grid layout, indicate the number of rows and columns you want the grid to have when you create a new instance of the `GridLayout` class. Here's a grid layout with three rows and two columns.

```
import java.awt.*;

public class GridLayoutTest extends java.applet.Applet {

  public void init() {
    setLayout(new GridLayout(3,2);
    add(new Button("One"));
    add(new Button("Two"));
    add(new Button("Three"));
    add(new Button("Four"));
    add(new Button("Five"));
    add(new Button("Six"));
  }
}
```

Grid layouts can also have a horizontal and vertical gap between components. To create gaps, add those pixel values:

```
setLayout(new GridLayout(3, 3, 10, 30));
```

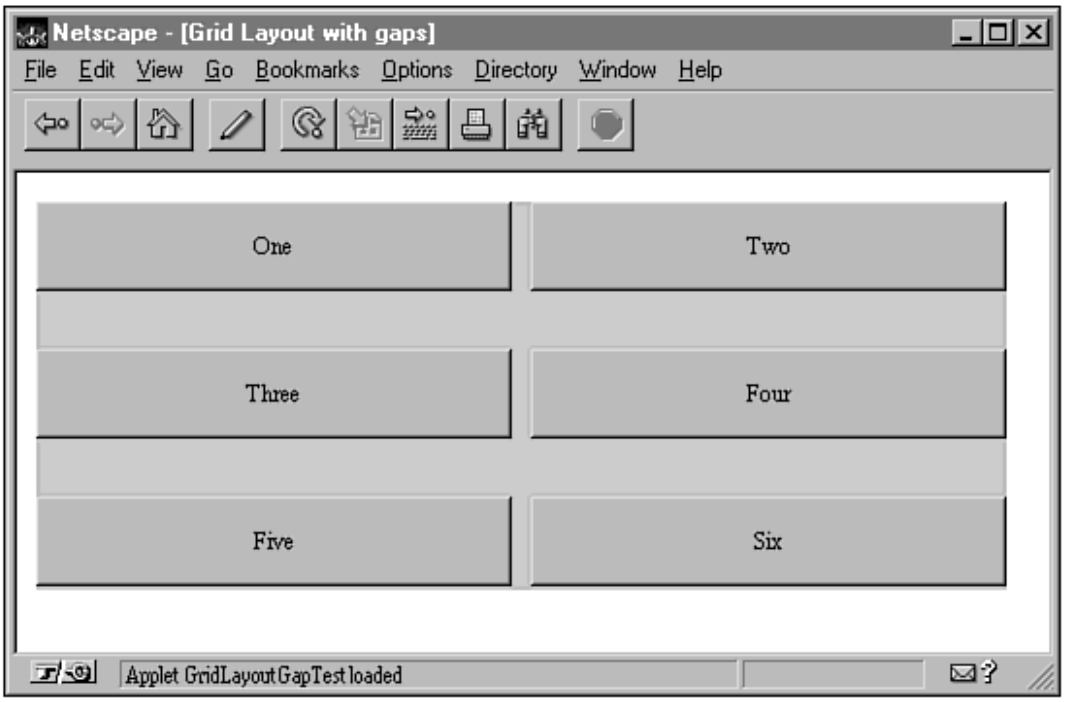


Figure shows a grid layout with a 10-pixel horizontal gap and a 30-pixel vertical gap.

Border Layouts

- *Border layouts* behave differently from flow and grid layouts.
- When you add a component to a panel that uses a border layout, you indicate its placement as a geographic direction: north, south, east, west, or center.

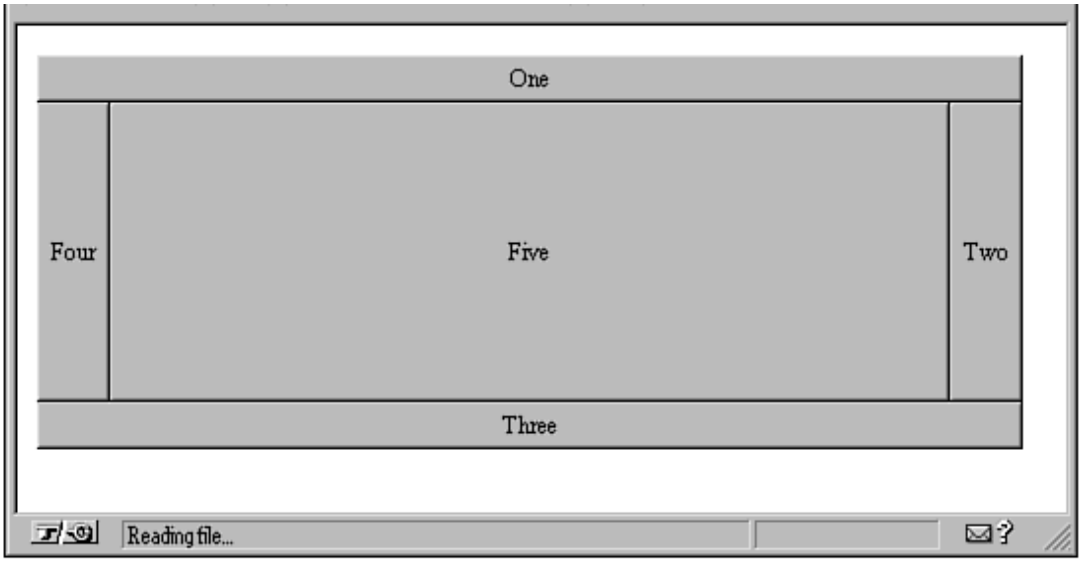
- The components around all the edges are laid out with as much size as they need; the component in the center, if any, gets any space left over.
- To use a border layout, you create it as you do the other layouts; then you add the individual components with a special `add()` method that has two arguments.
- The first argument is a string indicating the position of the component within the layout, and the second is the component to add:

```
add("North", new TextField("Title", 50));
```

You can also use this form of `add()` for the other layout managers; the string argument will just be ignored if it's not needed.

Here's the code to generate the border layout shown in Figure:

```
import java.awt.*;

public class BorderLayoutTest extends java.applet.Applet {

  public void init() {
    setLayout(new BorderLayout());
    add("North", new Button("One"));
    add("East", new Button("Two"));
    add("South", new Button("Three"));
    add("West", new Button("Four"));
    add("Center", new Button("Five"));
    add(new Button("Six"));
  }
}
```



Border layouts can also have horizontal and vertical gaps. Note that the north and south components extend all the way to the edge of the panel, so the gap will result in less vertical space for the east, right, and center components. To add gaps to a border layout, include those pixel values in the constructor as with the other layout managers:

```
setLayout(new BorderLayout(10, 10));
```

Card Layouts

- *Card layouts* behave much differently from the other layouts.
- When you add components to one of the other layout managers, all those components appear on the screen at once. Card layouts are used to produce slide shows of components, one at a time. If you've ever used the HyperCard program on the Macintosh, or seen dialog boxes on windows with several different tabbed pages, you've worked with the same basic idea.
- When you create a card layout, the components you add to the outer panel will be other container components-usually other panels. You can then use different layouts for those individual cards so that each screen has its own look.
- *Cards,* in a card layout, are different panels added one at a time and displayed one at a time. If you think of a card file, you'll get the idea; only one card can be displayed at once, but you can switch between cards.
- When you add each card to the panel, you can give it a name. Then, to flip between the container cards, you can use methods defined in the `CardLayout` class to move to a named card, move forward or back, or move to the first card or to the last card. Typically you'll have a set of buttons that call these methods to make navigating the card layout easier.

Here's a simple snippet of code that creates a card layout containing three cards:

```
setLayout(new CardLayout());
//add the cards
Panel one = new Panel()
add("first", one);
Panel two = new Panel()
add("second", two);
Panel three = new Panel()
add("third", three);

// move around
show(this, "second"); //go to the card named "second"
show(this, "third");   //go to the card named "third"
previous(this);       //go back to the second card
first(this);          // got to the first card
```

Grid Bag Layouts

I've saved grid bag layouts for last because although they are the most powerful way of managing awt layout, they are also extremely complicated.

Using one of the other four layout managers, it can sometimes be difficult to get the exact layout you want without doing a lot of nesting of panels within panels. Grid bags provide a more general-purpose solution. Like grid layouts, *grid bag layouts* allow you to arrange your components in a grid-like layout. However, grid bag layouts also allow you to control the span of individual cells in the grid, the proportions between the rows and columns, and the arrangement of components inside cells in the grid.

To create a grid bag layout, you actually use two classes: `GridBagLayout`, which provides the overall layout manager, and `GridBagConstraints`, which defines the properties of each component in the grid-its placement, dimensions, alignment, and so on. It's the relationship between the grid bag, the constraints, and each component that defines the overall layout.

In its most general form, creating a grid bag layout involves the following steps:

- Creating a `GridBagLayout` object and defining it as the current layout manager, as you would any other layout manager
- Creating a new instance of `GridBagConstraints`
- Setting up the constraints for a component
- Telling the layout manager about the component and its constraints
- Adding the component to the panel

Here's some simple code that sets up the layout and then creates constraints for a single button (don't worry about the various values for the constraints; I'll cover these later on in this section):

```
// set up layout
GridBagLayout gridbag = new GridBagLayout();
GridBagConstraints constraints = new GridBagConstraints();
setLayout(gridbag);

// define constraints for the button
Button b = new Button("Save");
constraints.gridx = 0;
constraints.gridy = 0;
constraints.gridwidth = 1;
constraints.gridheight = 1;
constraints.weightx = 30;
constraints.weighty = 30;
constraints.fill = GridBagConstraints.NONE;
```

```
constraints.anchor = GridBagConstraints.CENTER;

// attach constraints to layout, add button
gridbag.setConstraints(b, constraints);
add(b);
```

Retrieving and Using Images

- Basic image handling in Java is easy. The `Image` class in the `java.awt` package provides abstract methods to represent common image behavior, and special methods defined in `Applet` and `Graphics` give you everything you need to load and display images in your applet as easily as drawing a rectangle.

Getting Images

- To display an image in your applet, you first must load that image over the Net into your Java program. Images are stored as separate files from your Java class files, so you have to tell Java where to find them.
- The `Applet` class provides a method called `getImage()`, which loads an image and automatically creates an instance of the `Image` class for you. To use it, all you have to do is import the `java.awt.Image` class into your Java program, and then give `getImage` the URL of the image you want to load. There are two ways of doing the latter step:
    - The `getImage()` method with a single argument (an object of type `URL`) retrieves the image at that URL.
    - The `getImage()` method with two arguments: the base URL (also a `URL` object) and a string representing the path or filename of the actual image (relative to the base).

The latter form, therefore, is usually the one to use. The `Applet` class also provides two methods that will help with the base URL argument to `getImage()`:

- The `getDocumentBase()` method returns a `URL` object representing the directory of the HTML file that contains this applet. So, for example, if the HTML file is located at `http://www.myserver.com/htmlfiles/javahtml/`, `getDocumentBase()` returns a URL pointing to that path.
- The `getCodeBase()` method returns a string representing the directory in which this applet is contained-which may or may not be the same directory as the HTML file, depending on whether the `CODEBASE` attribute in `<APPLET>` is set or not.

- Whether you use `getDocumentBase()` or `getCodebase()` depends on whether your images are relative to your HTML files or relative to your Java class files. Use whichever one applies better to your situation.
- Note that either of these methods is more flexible than hard-coding a URL or pathname into the `getImage()` method; using either `getDocumentBase()` or `getCodeBase()` enables you to move your HTML files and applets around and Java can still find your images
- Here are a few examples of `getImage`, to give you an idea of how to use it. This first call to `getImage()` retrieves the file at that specific URL (http://www.server.com/files/image.gif). If any part of that URL changes, you have to recompile your Java applet to take into account the new path:

```
Image img = getImage(
    new URL("http://www.server.com/files/image.gif"));
```

In the following form of `getImage`, the `image.gif` file is in the same directory as the HTML files that refer to this applet:

```
Image img = getImage(getDocumentBase(), "image.gif")
```

In this similar form, the file `image.gif` is in the same directory as the applet itself:

```
Image img = getImage(getCodeBase(), "image.gif")
```

If you have lots of image files, it's common to put them into their own subdirectory. This form of `getImage()` looks for the file `image.gif` in the directory `images`, which, in turn, is in the same directory as the Java applet:

```
Image img = getImage(getCodeBase(), "images/image.gif")
```

If `getImage()` can't find the file indicated, it returns `null`. `drawImage()` on a `null` image will simply draw nothing. Using a `null` image in other ways will probably cause an error.

Note
Currently, Java supports images in the GIF and JPEG formats. Other image formats may be available later; however, for now, your images should be in either GIF or JPEG.

Drawing Images

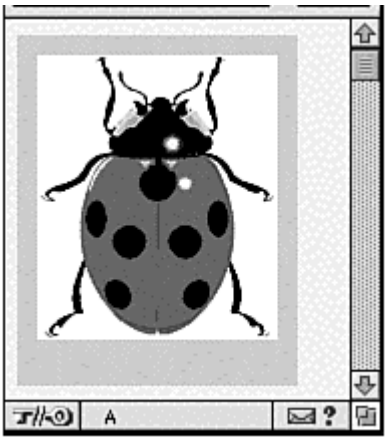
- All that stuff with `getImage()` does nothing except go off and retrieve an image and stuff it into an instance of the `Image` class. Now that you have an image, you have to do something with it.
- The most likely thing you're going to want to do with an image is display it as you would a rectangle or a text string. The `Graphics` class provides two methods to do just this, both called `drawImage()`.
- The first version of `drawImage()` takes four arguments: the image to display, the `x` and `y` positions of the top left corner, and `this`:

```
public void paint() {
    g.drawImage(img, 10, 10, this);
}
```

- This first form does what you would expect it to: It draws the image in its original dimensions with the top-left corner at the given `x` and `y` positions.
- Listing below shows the code for a very simple applet that loads an image called `ladybug.gif` and displays it. Figure 11.1 shows the obvious result.

Listing . The Ladybug applet.

```
 1:import java.awt.Graphics;
 2:import java.awt.Image;
 3:
 4:public class LadyBug extends java.applet.Applet {
 5:
 6:    Image bugimg;
 7:
 8:    public void init() {
 9:       bugimg = getImage(getCodeBase(),
10:          "images/ladybug.gif");
11:    }
12:
13:    public void paint(Graphics g) {
14:      g.drawImage(bugimg, 10, 10,this);
15:    }
16:}
```



- In this example the instance variable `bugimg` holds the ladybug image, which is loaded in the `init()` method. The `paint()` method then draws that image on the screen.
- The second form of `drawImage()` takes six arguments: the image to draw, the `x` and `y` coordinates of the top-left corner, a width and height of the image bounding box, and `this`.
- If the width and height arguments for the bounding box are smaller or larger than the actual image, the image is automatically scaled to fit.
- By using those extra arguments, you can squeeze and expand images into whatever space you need them to fit in (keep in mind, however, that there may be some image degradation from scaling it smaller or larger than its intended size).
- One helpful hint for scaling images is to find out the size of the actual image that you've loaded, so you can then scale it to a specific percentage and avoid distortion in either direction.
- Two methods defined for the `Image` class can give you that information:
    - `getWidth()` and
    - `getHeight()`.
- Both take a single argument, an instance of `ImageObserver`, which is used to track the loading of the image (more about this later). Most of the time, you can use just `this` as an argument to either `getWidth()` or `getHeight()`.

If you stored the ladybug image in a variable called `bugimg`, for example, this line returns the width of that image, in pixels:

```
theWidth = bugimg.getWidth(this);
```

Technical Note

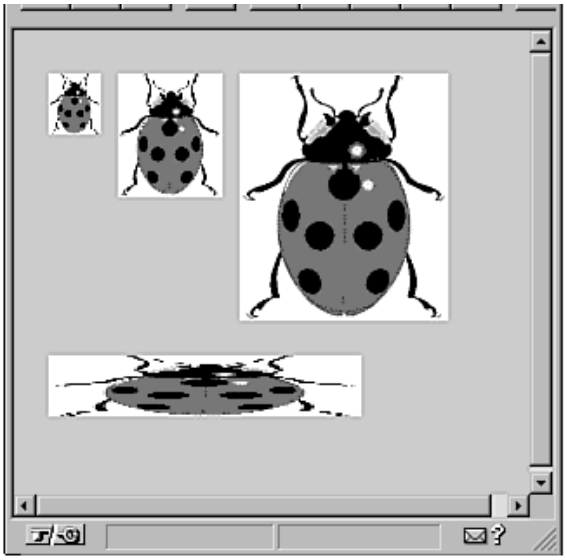
Here's another case where, if the image isn't loaded all the way, you may get different results. Calling `getWidth()` or `getHeight()` before the image has fully loaded will result in values of `-1` for each one. Tracking image loading with image observers can help you keep track of when this information appears.

Listing shows another use of the ladybug image, this time scaled several times to different sizes .

Listing: More ladybugs, scaled.

```
 1:  import java.awt.Graphics;
 2: import java.awt.Image;
 3:
 4: public class LadyBug2 extends java.applet.Applet {
 5:
 6:     Image bugimg;
 7:
 8:     public void init() {
 9:         bugimg = getImage(getCodeBase(),
10:             "images/ladybug.gif");
11:     }
12:
13:     public void paint(Graphics g) {
14:         int iwidth = bugimg.getWidth(this);
15:         int iheight = bugimg.getHeight(this);
16:         int xpos = 10;
17:
18:         // 25 %
19:        g.drawImage(bugimg, xpos, 10,
20:            iwidth / 4, iheight / 4, this);
21:
22:         // 50 %
23:         xpos += (iwidth / 4) + 10;
24:         g.drawImage(bugimg, xpos , 10,
25:              iwidth / 2, iheight / 2, this);
26:
27:         // 100%
```

```
28:          xpos += (iwidth / 2) + 10;
29:          g.drawImage(bugimg, xpos, 10, this);
30:
31:          // 150% x, 25% y
32:          g.drawImage(bugimg, 10, iheight + 30,
33:              (int)(iwidth * 1.5), iheight / 4, this);
34:      }
35: }
```



Event Handling in Java

- Java events are part of the Java awt (Abstract Windowing Toolkit) package.
- An event is the way that the awt communicates to you, as the programmer, and to other Java awt components that *something* has happened. That something can be input from the user (mouse movements or clicks, keypresses), changes in the system environment (a window opening or closing, the window being scrolled up or down), or a host of other things that might, in some way, affect the operation of the program.
- Some events are handled by the awt or by the environment your applet is running in (the browser) without you needing to do anything. `paint()` methods, for example, are generated and handled by the environment-all you have to do is tell the awt what you want painted when it gets to your part of the window.
- However, you may need to know about some events, such as a mouse click inside the boundaries of your applet. By writing your Java programs to handle these kinds of events, you can get input from the user and have your applet change its behavior based on that input.

This section discusses about managing simple events, including the following basics:

- Mouse clicks
- Mouse movements, including mouse dragging
- Keyboard actions

`handleEvent()` method, which is the basis for collecting, handling, and passing on events of all kinds from your applet to other components of the window or of your applet itself.

Mouse Clicks

- Mouse-click events occur when your user clicks the mouse somewhere in the body of your applet. You can intercept mouse clicks to do very simple things-for example, to toggle the sound on and off in your applet, to move to the next slide in a presentation, or to clear the screen and start over-or you can use mouse clicks in conjunction with mouse movements to perform more complex motions inside your applet.

Mouse Down and Mouse Up Events

- When you click the mouse once, the awt generates two events: a mouse down event when the mouse button is pressed and a mouse up event when the button is released.
- Why two individual events for a single mouse action? Because you may want to do different things for the "down" and the "up." For example, look at a pull-down menu.
- The mouse down extends the menu, and the mouse up selects an item (with mouse drags between-but you'll learn about that one later). If you have only one event for both actions (mouse up and mouse down), you cannot implement that sort of user interaction.
- Handling mouse events in your applet is easy-all you have to do is override the right method definition in your applet. That method will be called when that particular event occurs. Here's an example of the method signature for a mouse down event:

```
public boolean mouseDown(Event evt, int x, int y) {
...
}
```

The `mouseDown()` method (and the `mouseUp()` method as well) takes three parameters: the event itself and the x and y coordinates where the mouse down or mouse up event occurred.

- The `evt` argument is an instance of the class `Event`. All system events generate an instance of the `Event` class, which contains information about where and when the event took place, the kind of event it is, and other information that you might want to know about this event
- The x and the y coordinates of the event, as passed in through the `x` and `y` arguments to the `mouseDown()` method, are particularly nice to know because you can use them to determine precisely where the mouse click took place.
- So, for example, if the mouse down event were over a graphical button, you could activate that button.

For example, here's a simple method that prints out information about a mouse down when it occurs:

```
public boolean mouseDown(Event evt, int x, int y)
 {
    System.out.println("Mouse down at " + x + "," + y);
    return true;
```

By including this method in your applet, every time your user clicks the mouse inside your applet, this message will get printed. The awt system calls each of these methods when the actual event takes place.

- Note that this method, far, returns a boolean value instead of not returning anything (`void`). Having an event handler method return `true` or `false` determines whether a given component can intercept an event or whether it needs to pass it on to the enclosing component.
- The general rule is that if your method intercepts and does something with the event, it should return `true`.
- If for any reason the method doesn't do anything with that event, it should return `false` so that other components in the system can have a chance to see that event.
- In most of the examples in today's lesson, you'll be intercepting simple events, so most of the methods here will return `true`. Tomorrow you'll learn about nesting components and passing events up the component hierarchy.
- The second half of the mouse click is the `mouseUp()` method, which is called when the mouse button is released. To handle a mouse up event, add the `mouseUp()` method to your applet: `mouseUp()` looks just like `mouseDown()`:

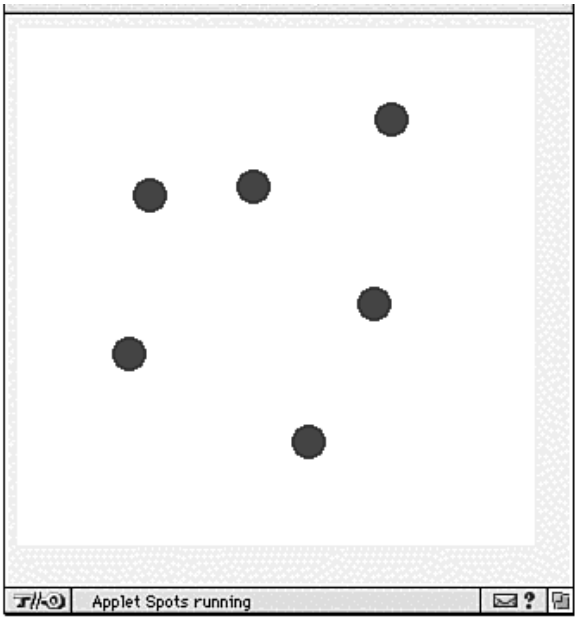
```
public boolean mouseUp(Event evt, int x, int y) {
    ....
}
```

An Example: Spots

In this section you'll create an example of an applet that uses mouse events-mouse down events in particular. The Spots applet starts with a blank screen and then sits and waits. When you click the mouse on that screen, a blue dot is drawn. You can place up to 10 dots on the screen. Figure below shows the Spots applet.

Let's start from the beginning and build this applet, starting from the initial class definition:

```
import java.awt.Graphics;
import java.awt.Color;
import java.awt.Event;

public class Spots extends java.applet.Applet {

    final int MAXSPOTS = 10;
    int xspots[] = new int[MAXSPOTS];
    int yspots[] = new int[MAXSPOTS];
    int currspots = 0;

}
```



- This class uses three other awt classes: `Graphics`, `Color`, and `Event`. That last class, `Event`, needs to be imported in any applets that use events. The class has four instance variables: a constant to determine the maximum number of spots that can be drawn, two arrays to store the x and y coordinates of the spots that have already been drawn, and an integer to keep track of the number of the current spot.
- Let's start by adding the `init()` method, which does only one thing: set the background color to `white`:

```
public void init() {
    setBackground(Color.white);
}
```

- We've set the background here in `init()` instead of in `paint()` as you have in past examples because you need to set the background only once. Because `paint()` is called repeatedly each time a new spot is added, setting the background in the `paint()` method unnecessarily slows down that method. Putting it here is a much better idea.

The main action of this applet occurs with the `mouseDown()` method, so let's add that one now:

```
public boolean mouseDown(Event evt, int x, int y) {
    if (currspots < MAXSPOTS) {
        addspot(x,y);
        return true;
    }
    else {
       System.out.println("Too many spots.");
       return false;
    }
}
```

- When the mouse click occurs, the `mouseDown()` method tests to see whether there are fewer than 10 spots. If so, it calls the `addspot()` method (which you'll write soon) and returns `true` (the mouse down event was intercepted and handled). If not, it just prints an error message and returns `false`.
- What does `addspot()` do? It adds the coordinates of the spot to the arrays that store the coordinates, increments the `currspots` variable, and then calls `repaint()`:

```
void addspot(int x, int y) {
    xspots[currspots] = x;
    yspots[currspots] = y;
    currspots++;
    repaint();
}
```

- You may be wondering why you have to keep track of all the past spots in addition to the current spot. It's because of `repaint()`: Each time you paint the screen, you have to paint all the old spots in addition to the newest spot. Otherwise, each time you painted a new spot, the older spots would get erased. Now, on to the `paint()` method:

```
public void paint(Graphics g) {
    g.setColor(Color.blue);
    for (int i = 0; i < currspots; i++) {
        g.fillOval(xspots[i] -10, yspots[i] - 10, 20, 20);
    }
}
```

- Inside `paint()`, you just loop through the spots you've stored in the `xspots` and `yspots` arrays, painting each one (actually, painting them a little to the right and upward so that the spot is painted around the mouse pointer rather than below and to the right).
- That's it! That's all you need to create an applet that handles mouse clicks. Everything else is handled for you. You have to add the appropriate behavior to `mouseDown()` or `mouseUp()` to intercept and handle that event. Listing shows the full text for the Spots applet.

Listing . The Spots applet.

```
 1: import java.awt.Graphics;
 2: import java.awt.Color;
 3: import java.awt.Event;
 4:
 5: public class Spots extends java.applet.Applet {
 6:
 7:     final int MAXSPOTS = 10;
 8:     int xspots[] = new int[MAXSPOTS];
 9:     int yspots[] = new int[MAXSPOTS];
10:     int currspots = 0;
```

```
11:
12:     public void init() {
13:         setBackground(Color.white);
14:     }
15:
16:     public boolean mouseDown(Event evt, int x, int y) {
17:         if (currspots < MAXSPOTS) {
18:             addspot(x,y);
19:             return true;
20:         }
21:         else {
22:            System.out.println("Too many spots.");
23:            return false;
24:         }
25:     }
26:
27:     void addspot(int x,int y) {
28:          xspots[currspots] = x;
29:          yspots[currspots] = y;
30:          currspots++;
31:          repaint();
32:     }
33:
34:     public void paint(Graphics g) {
35:         g.setColor(Color.blue);
36:         for (int i = 0; i < currspots; i++) {
37:               g.fillOval(xspots[i] - 10, yspots[i] - 10, 20, 20);
38:        }
39:     }
40: }
```

Double-Clicks

- What if the mouse event you're interested in is more than a single mouse click-what if you want to track double- or triple-clicks? The Java `Event` class provides a variable for tracking this information, called `clickCount`. `clickCount` is an integer representing the number of consecutive mouse clicks that have occurred (where "consecutive" is usually determined by the operating system or the mouse hardware). If you're interested in multiple mouse clicks in your applets, you can test this value in the body of your `mouseDown()` method, like this:

```
public boolean mouseDown(Event evt, int x, int y) {
    switch (evt.clickCount) {
      case 1:  // single-click
      case 2:  // double-click
      case 3:  // triple-click
      ....
    }
}
```

Mouse Movements

Every time the mouse is moved a single pixel in any direction, a mouse move event is generated. There are two mouse movement events: mouse drags, where the movement occurs with the mouse button pressed down, and plain mouse movements, where the mouse button isn't pressed.

To manage mouse movement events, use the `mouseDrag()` and `mouseMove()` methods.

Mouse Drag and Mouse Move Events

The `mouseDrag()` and `mouseMove()` methods, when included in your applet code, intercept and handle mouse movement events. Mouse move and move drag events are generated for every pixel change the mouse moves, so a mouse movement from one side of the applet to the other may generate hundreds of events. The `mouseMove()` method, for plain mouse pointer movements without the mouse button pressed, looks much like the mouse-click methods:

```
public boolean mouseMove(Event evt, int x, int y) {
    ...
}
```

The `mouseDrag()` method handles mouse movements made with the mouse button pressed down (a complete dragging movement consists of a mouse down event, a series of mouse drag events for each pixel the mouse is moved, and a mouse up when the button is released). The `mouseDrag()` method looks like this:

```
public boolean mouseDrag(Event evt, int x, int y) {
    ...
}
```

Note that for both the `mouseMove()` and `mouseDrag()` methods, the arguments for the x and y coordinates are the new location of the mouse, not its starting location.

Mouse Enter and Mouse Exit Events

Finally, there are the `mouseEnter()` and `mouseExit()` methods. These two methods are called when the mouse pointer enters or exits an applet or a portion of that applet. (In case you're wondering why you might need to know this, it's more useful on awt components that you might put inside an applet.

Both `mouseEnter()` and `mouseExit()` have signatures similar to the mouse click methods-three arguments: the event object and the x and y coordinates of the point where the mouse entered or exited the applet. These examples show the signatures for `mouseEnter()` and `mouseExit()`:

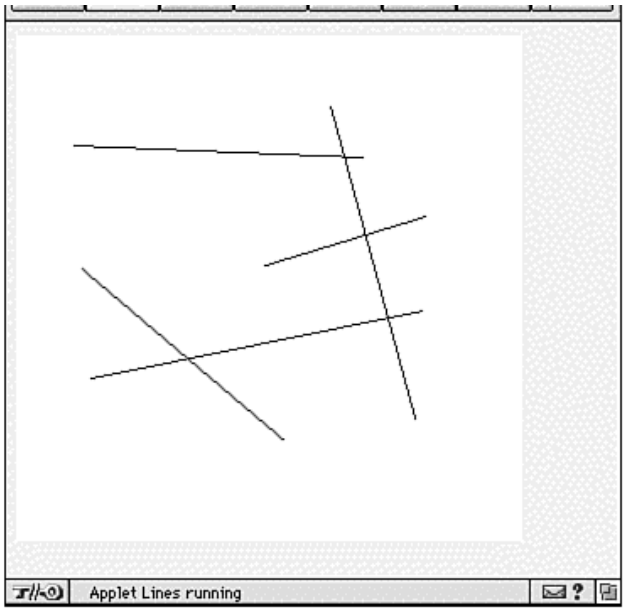
```
public boolean mouseEnter(Event evt, int x, int y) {
    ...
}

public boolean mouseExit(Event evt, int x, int y) {
    ...
}
```

An Example: Drawing Lines

Examples always help to make concepts more concrete. In this section you'll create an applet that enables you to draw straight lines on the screen by dragging from the startpoint to the endpoint. Figure 12.2 shows the applet at work.

As with the Spots applet (on which this applet is based), let's start with the basic definition and work our way through it, adding the appropriate methods to build the applet. Here's a simple class definition for the Lines applet, with a number of initial instance variables and a simple `init()` method:

```
import java.awt.Graphics;
import java.awt.Color;
import java.awt.Event;
import java.awt.Point;

public class Lines extends java.applet.Applet {

    final int MAXLINES = 10;
    Point starts[] = new Point[MAXLINES]; // starting points
    Point ends[] = new Point[MAXLINES];    // ending points
    Point anchor;    // start of current line
    Point currentpoint; // current end of line
    int currline = 0; // number of lines

    public void init() {
        setBackground(Color.white);
    }
}
```



- This applet adds a few more things than Spots. Unlike Spots, which keeps track of individual integer coordinates, this one keeps track of `Point` objects. Points represent an `x` and a `y` coordinate, encapsulated in a single object. To deal with points, you import the `Point` class and set up a bunch of instance variables that hold points:
    - The `starts` array holds points representing the starts of lines already drawn.
    - The `ends` array holds the endpoints of those same lines.
    - `anchor` holds the starting point of the line currently being drawn.
    - `currentpoint` holds the current endpoint of the line currently being drawn.
    - `currline` holds the current number of lines (to make sure you don't go over `MAXLINES`, and to keep track of which line in the array to access next).
- Finally, the `init()` method, as in the Spots applet, sets the background of the applet to `white`.
- The three main events this applet deals with are `mouseDown()`, to set the anchor point for the current line, `mouseDrag()`, to animate the current line as it's being drawn, and `mouseUp()`, to set the ending point for the new line. Given that you have instance variables to hold each of these values, it's merely a matter of plugging the right variables into the right methods. Here's `mouseDown()`, which sets the anchor point (but only if we haven't exceeded the maximum number of lines):

```
public boolean mouseDown(Event evt, int x, int y) {
   if (currline < MAXLINES) {
       anchor = new Point(x,y);
       return true;
  }
   else  {
      System.out.println("Too many lines.");
      return false;
```

```
    }
}
```

- While the mouse is being dragged to draw the line, the applet animates the line being drawn. As you drag the mouse around, the new line moves with it from the anchor point to the tip of the mouse.
- The `mouseDrag()` event contains the current point each time the mouse moves, so use that method to keep track of the current point (and to repaint for each movement so the line "animates"). Note that if we've exceeded the maximum number of lines, we won't want to do any of this. Here's the `mouseDrag()` method to do all those things:

```
public boolean mouseDrag(Event evt, int x, int y) {
    if (currline < MAXLINES) {
        currentpoint = new Point(x,y);
        repaint();
        return true;
     }
    else return false;
}
```

The new line doesn't get added to the arrays of old lines until the mouse button is released. Here's `mouseUp()`, which tests to make sure you haven't exceeded the maximum number of lines before calling the `addline()` method (described next):

```
public boolean mouseUp(Event evt, int x, int y) {
     if (currline < MAXLINES) {
         addline(x,y);
         return true;
     }
     else return false;
}
```

The `addline()` method is where the arrays of starting and ending points get updated and where the applet is repainted to take the new line into effect:

```
void addline(int x,int y) {
    starts[currline] = anchor;
    ends[currline] = new Point(x,y);
    currline++;
    currentpoint = null;
    anchor = null;
    repaint();
}
```

- Note that in this method you also set `currentpoint` and `anchor` to `null`. Why? Because the current line you were drawing is over. By setting these variables to `null`, you can test for that value in the `paint()` method to see whether you need to draw a current line.

Painting the applet means drawing all the old lines stored in the `starts` and `ends` arrays, as well as drawing the current line in progress (whose endpoints are in `anchor` and `currentpoint`, respectively). To show the animation of the current line, draw it in blue. Here's the `paint()` method for the Lines applet:

```
public void paint(Graphics g) {

    // Draw existing lines
    for (int i = 0; i < currline; i++) {
        g.drawLine(starts[i].x, starts[i].y,
            ends[i].x, ends[i].y);
    }

    // Draw current line
    g.setColor(Color.blue);
    if (currentpoint != null)
        g.drawLine(anchor.x, anchor.y,
            currentpoint.x, currentpoint.y);
}
```

In `paint()`, when you're drawing the current line, you test first to see whether `currentpoint` is `null`. If it is, the applet isn't in the middle of drawing a line, so there's no reason to try drawing a line that doesn't exist. By testing for `currentpoint` (and by setting `currentpoint` to `null` in the `addline()` method), you can paint only what you need.

That's it-just 60 lines of code and a few basic methods, and you have a very basic drawing application in your Web browser. Listing below shows the full text of the Lines applet so that you can put the pieces together.

Listing The Lines applet.

```
 1: import java.awt.Graphics;
 2: import java.awt.Color;
 3: import java.awt.Event;
 4: import java.awt.Point;
 5:
 6: public class Lines extends java.applet.Applet {
 7:
 8:     final int MAXLINES = 10;
 9:     Point starts[] = new Point[MAXLINES]; // starting points
10:     Point ends[] = new Point[MAXLINES];    // endingpoints
11:     Point anchor;    // start of current line
12:     Point currentpoint; // current end of line
13:     int currline = 0; // number of lines
14:
15:     public void init() {
16:         setBackground(Color.white);
17:     }
18:
19:     public boolean mouseDown(Event evt, int x, int y) {
20:        if (currline < MAXLINES) {
21:            anchor = new Point(x,y);
```

```
22:             return true;
23:        }
24:         else  {
25:            System.out.println("Too many lines.");
26:            return false;
27:         }
28:      }
29:
30:    public boolean mouseUp(Event evt, int x, int y) {
31:         if (currline < MAXLINES) {
32:             addline(x,y);
33:             return true;
34:         }
35:         else return false;
36:    }
37:
38:     public boolean mouseDrag(Event evt, int x, int y) {
39:        if (currline < MAXLINES) {
40:            currentpoint = new Point(x,y);
41:            repaint();
42:            return true;
43:         }
44:        else return false;
45:     }
46:
47:     void addline(int x,int y) {
48:         starts[currline] = anchor;
49:         ends[currline] = new Point(x,y);
50:         currline++;
51:         currentpoint = null;
52:         anchor = null;
53:         repaint();
54:     }
55:
56:     public void paint(Graphics g) {
57:
58:         // Draw existing lines
59:         for (int i = 0; i < currline; i++) {
50:             g.drawLine(starts[i].x, starts[i].y,
51:                  ends[i].x, ends[i].y);
52:         }
53:
54:         // draw current line
55:         g.setColor(Color.blue);
56:         if (currentpoint != null)
57:             g.drawLine(anchor.x,anchor.y,
58:             currentpoint.x,currentpoint.y);
59:     }
60:}
```

Keyboard Events

- A keyboard event is generated whenever a user presses a key on the keyboard. By using keyboard events, you can get hold of the values of the keys the user pressed to perform an action or merely to get character input from the users of your applet.

The `keyDown()` and `keyUp()` Methods

To capture a keyboard event, use the `keyDown()` method:

```
public boolean keyDown(Event evt, int key) {
    ...
}
```

- The keys generated by key down events (and passed into `keyDown()` as the `key` argument) are integers representing Unicode character values, which include alphanumeric characters, function keys, tabs, returns, and so on. To use them as characters (for example, to print them), you need to cast them to characters:

  ```
  currentchar = (char)key;
  ```

- Here's a simple example of a `keyDown()` method that does nothing but print the key you just typed in both its Unicode and character representation (it can be fun to see which key characters produce which values):

  ```
  public boolean keyDown(Event evt, int key) {
      System.out.println("ASCII value: " + key);
      System.out.println("Character: " + (char)key);
      return true;
  }
  ```

As with mouse clicks, each key down event also has a corresponding key up event. To intercept key up events, use the `keyUp()` method:

```
public booklean keyUp(Event evt, int key)  {
   ...
}
```

Default Keys

- The `Event` class provides a set of class variables that refer to several standard nonalphanumeric keys, such as the arrow and function keys. If your applet's interface uses these keys, you can provide more readable code by testing for these names in your `keyDown()` method rather than testing for their numeric values (and you're also more likely to be cross-platform if you use these variables).
- For example, to test whether the up arrow was pressed, you might use the following snippet of code:

  ```
  if (key == Event.UP) {
      ...
  }
  ```

- Because the values these class variables hold are integers, you also can use the `switch` statement to test for them.
- Table 12.1 shows the standard event class variables for various keys and the actual keys they represent.

Table : Standard keys defined by the `Event` class.

Class Variable	Represented Key
`Event.HOME`	The Home key
`Event.END`	The End key
`Event.PGUP`	The Page Up key
`Event.PGDN`	The Page Down key
`Event.UP`	The up arrow
`Event.DOWN`	The down arrow
`Event.LEFT`	The left arrow
`Event.RIGHT`	The right arrow
`Event.f1`	The f1 key
`Event.f2`	The f2 key
`Event.f3`	The f3 key
`Event.f4`	The f4 key
`Event.f5`	The f5 key
`Event.f6`	The f6 key
`Event.f7`	The f7 key
`Event.f8`	The f8 key
`Event.f9`	The f9 key
`Event.f10`	The f10 key
`Event.f11`	The f11 key
`Event.f12`	The f12 key

An Example: Entering, Displaying, and Moving Characters

- Let's look at an applet that demonstrates keyboard events. With this applet, you type a character, and that character is displayed in the center of the applet window. You then can move that character around on the screen with the arrow keys. Typing another character at any time changes the character as it's currently displayed. Figure below shows an example.

Note
To get this applet to work, you might have to click once with the mouse on it in order for the keys to show up. This is to make sure the applet has the keyboard focus (that is, that its actually listening when you type characters on the keyboard).

- This applet is actually less complicated than the previous applets you've used. This one has only three methods: `init()`, `keyDown()`, and `paint()`. The instance variables are also simpler because the only things you need to keep track of are the x and y positions of the current character and the values of that character itself. Here's the initial class definition:

```
import java.awt.Graphics;
import java.awt.Event;
import java.awt.Font;
import java.awt.Color;

public class Keys extends java.applet.Applet {

    char currkey;
    int currx;
    int curry;
}
```

- Let's start by adding an `init()` method. Here, `init()` is responsible for three things: setting the background color, setting the applet's font (here, 36-point Helvetica bold), and setting the beginning position for the character (the middle of the screen, minus a few points to nudge it up and to the right):

```
 public void init() {
     currx = (size().width / 2) - 8;
     curry = (size().height / 2) - 16;
     setBackground(Color.white);
     setFont(new Font("Helvetica", Font.BOLD, 36));
 }
```

- Because this applet's behavior is based on keyboard input, the `keyDown()` method is where most of the work of the applet takes place:

```
public boolean keyDown(Event evt, int key) {
   switch (key) {
         case Event.DOWN:
             curry += 5;
             break;
         case Event.UP:
             curry -= 5;
             break;
         case Event.LEFT:
             currx -= 5;
             break;
         case Event.RIGHT:
             currx += 5;
             break;
         default:
             currkey = (char)key;
         }
         repaint();
         return true;
}
```

- In the center of the `keyDown()` applet is a `switch` statement that tests for different key events. If the event is an arrow key, the appropriate change is made to the character's position. If the event is any other key, the character itself is changed (that's the default part of the `switch`). The method finishes up with a `repaint()` and returns `true`.
- The `paint()` method here is almost trivial; just display the current character at the current position. However, note that when the applet starts up, there's no initial character and nothing to draw, so you have to take that into account. The `currkey` variable is initialized to `0`, so you paint the applet only if `currkey` has an actual value:

```
public void paint(Graphics g) {
    if (currkey != 0) {
        g.drawString(String.valueOf(currkey), currx,curry);
    }
}
```

Listing shows the complete source code for the Keys applet.

Listing : The Keys applet.

```
 1: import java.awt.Graphics;
 2: import java.awt.Event;
 3: import java.awt.Font;
 4: import java.awt.Color;
 5:
```

```
 6: public class Keys extends java.applet.Applet {
 7:
 8:     char currkey;
 9:     int currx;
10:    int curry;
11:
12:     public void init() {
13:         currx = (size().width / 2) -8;  // default
14:         curry = (size().height / 2) -16;
15:
16:         setBackground(Color.white);
17:         setFont(new Font("Helvetica",Font.BOLD,36));
18:     }
19:
20:     public boolean keyDown(Event evt, int key) {
21:         switch (key) {
22:         case Event.DOWN:
23:             curry += 5;
24:             break;
25:         case Event.UP:
26:             curry -= 5;
27:             break;
28:         case Event.LEFT:
29:             currx -= 5;
30:             break;
31:         case Event.RIGHT:
32:             currx += 5;
33:             break;
34:         default:
35:             currkey = (char)key;
36:         }
37:
38:         repaint();
39:         return true;
40:     }
41:
42:     public void paint(Graphics g) {
43:         if (currkey != 0) {
44:             g.drawString(String.valueOf(currkey), currx,curry);
45:         }
46:     }
47: }
```

Testing for Modifier Keys and Multiple Mouse Buttons

- Shift, Control (Ctrl), and Meta are modifier keys. They don't generate key events themselves, but when you get an ordinary mouse or keyboard event, you can test to see whether those modifier keys were held down when the event occurred.
- Sometimes it may be obvious-shifted alphanumeric keys produce different key events than unshifted ones, for example. For other events, however-mouse events in particular-you may want to handle an event with a modifier key held down differently from a regular version of that event.

Note
The Meta key is commonly used on UNIX systems; it's usually mapped to Alt on pc keyboards and Command (apple) on Macintoshes.

- The `Event` class provides three methods for testing whether a modifier key is held down: `shiftDown()`, `metaDown()`, and `controlDown()`. All return boolean values based on whether that modifier key is indeed held down.
- You can use these three methods in any of the event- handling methods (mouse or keyboard) by calling them on the event object passed into that method:

```
public boolean mouseDown(Event evt, int x, int y ) {
if (evt.shiftDown())
      // handle shift-click
else // handle regular click
}
```

- One other significant use of these modifier key methods is to test for which mouse button generated a particular mouse event on systems with two or three mouse buttons.
- By default, mouse events (such as mouse down and mouse drag) are generated regardless of which mouse button is used.
- However, Java events internally map left and middle mouse actions to meta and Control (Ctrl) modifier keys, respectively, so testing for the key tests for the mouse button's action.
- By testing for modifier keys, you can find out which mouse button was used and execute different behavior for those buttons than you would for the left button. Use an `if` statement to test each case, like this:

```
public boolean mouseDown(Event evt, int x, int y ) {
if (evt.metaDown())
      // handle a right-click
else if (evt.controlDown())
      // handle a middle-click
else // handle a regular click
}
```

- Note that because this mapping from multiple mouse buttons to keyboard modifiers happens automatically, you don't have to do a lot of work to make sure your applets or applications work on different systems with different kinds of mouse devices.
- Because left-button or right-button mouse clicks map to modifier key events, you can use those actual modifier keys on systems with fewer mouse buttons to generate exactly the same results.
- So, for example, holding down the Ctrl key and clicking the mouse on Windows or holding the Control key on the Macintosh is the same as clicking the middle mouse button on a three-button mouse; holding down the Command (apple) key and clicking the mouse on the Mac is the same as clicking the right mouse button on a two- or three-button mouse.
- Consider, however, that the use of different mouse buttons or modifier keys may not be immediately obvious if your applet or application runs on a system with fewer buttons than you're used to working with.
- Consider restricting your interface to a single mouse button or to providing help or documentation to explain the use of your program in this case.

The awt Event Handler

- The default methods you've learned about today for handling basic events in applets are actually called by a generic event handler method called `handleEvent()`. The `handleEvent()` method is how the awt generically deals with events that occur between application components and events based on user input.
- In the default `handleEvent()` method, basic events are processed and the methods you learned about today are called. To handle events other than those mentioned here, you need to override `handleEvent()` in your own Java programs. The `handleEvent()` method looks like this:

```
public boolean handleEvent(Event evt) {
...
}
```

- To test for specific events, examine the `id` instance variable of the `Event` object that gets passed in to `handleEvent()`.
- The event ID is an integer, but fortunately the `Event` class defines a whole set of event IDs as class variables whose names you can test for in the body of `handleEvent()`.

- Because these class variables are integer constants, a `switch` statement works particularly well. For example, here's a simple `handleEvent()` method to print out debugging information about mouse events:

```
public boolean handleEvent(Event evt) {
    switch (evt.id) {
    case Event.MOUSE_DOWN:
        System.out.println("MouseDown: " +
                evt.x + "," + evt.y);
        return true;
    case Event.MOUSE_UP:
        System.out.println("MouseUp: " +
                evt.x + "," + evt.y);
        return true;
    case Event.MOUSE_MOVE:
        System.out.println("MouseMove: " +
                evt.x + "," + evt.y);
        return true;
    case Event.MOUSE_DRAG:
        System.out.println("MouseDrag: " +
                evt.x + "," + evt.y);
        return true;
    default:
        return false;
    }
}
```

You can test for the following keyboard events:

- `Event.KEY_PRESS` is generated when a key is pressed (the same as the `keyDown()` method).
- `Event.KEY_RELEASE` is generated when a key is released.
- `Event.KEY_ACTION` and `Event.KEY_ACTION_RELEASE` are generated when an action key (a function key, an arrow key, Page Up, Page Down, or Home) is pressed or released.

You can test for these mouse events:

- `Event.MOUSE_DOWN` is generated when the mouse button is pressed (the same as the `mouseDown()` method).
- `Event.MOUSE_UP` is generated when the mouse button is released (the same as the `mouseUp()` method).
- `Event.MOUSE_MOVE` is generated when the mouse is moved (the same as the `mouseMove()` method).
- `Event.MOUSE_DRAG` is generated when the mouse is moved with the button pressed (the same as the `mouseDrag()` method).
- `Event.MOUSE_ENTER` is generated when the mouse enters the applet (or a component of that applet). You can also use the `mouseEnter()` method.

- `Event.MOUSE_EXIT` is generated when the mouse exits the applet. You can also use the `mouseExit()` method.

In addition to these events, the `Event` class has a whole suite of methods for handling awt components. You'll learn more about these events tomorrow.

Note that if you override `handleEvent()` in your class, none of the default event-handling methods you learned about today will get called unless you explicitly call them in the body of `handleEvent()`, so be careful if you decide to do this. One way to get around this is to test for the event you're interested in, and if that event isn't it, call `super.handleEvent()` so that the superclass that defines `handleEvent()` can process things. Here's an example of how to do this:

```
public boolean handleEvent(Event evt) {
    if (evt.id == Event.MOUSE_DOWN) {
        // process the mouse down
        return true;
    } else {
        return super.handleEvent(evt);
    }
}
```

Also, note that like the individual methods for individual events, `handleEvent()` also returns a boolean. The value you return here is particularly important; if you pass handling of the event to another method, you must return `false`. If you handle the event in the body of this method, return `true`. If you pass the event up to a superclass, that method will return `true` or `false`; you don't have to yourself.