Part Two

Graphics Development with Java 2-D and the Abstract Window Toolkit
There are numerous ways to deliver the content of your games to others. As you already know, Java applications can be run within any environment that supports Java. Such environments include not only Solaris or Windows, but handheld and “smart” devices as well. Heck, you can even program a blender to function as a game! However, the most popular way of running games is from an applet, either running within a Web browser or Sun’s appletviewer utility. We’ll begin Part Two with a look at the Java Applet class and how we can develop visual programs with very little code.

By now, you should have much of the knowledge required to start making games. You know the fundamentals of the Java language and how to use pre-defined Java classes to your advantage. In Part Two, we’ll begin our investigation into actual 2-D game programming. To do so, we’ll take a look at the Java Abstract Window Toolkit (AWT) and one of its main extensions, Java 2-D. Specifically, we’ll examine the following topics:

- **Applet Basics.** What exactly an applet is, as well as applet structure and use.
- **AWT Components.** Including using buttons, drop-boxes, text boxes, and layout managers to provide interaction with the user.
- **Event Handling.** Including how to catch events generated by the mouse and keyboard and respond to these events appropriately.
- **Java 2-D.** A two-chapter overview on modeling, rendering, and filling shapes, loading and drawing images from file, as well as manipulating and rendering text. I’ll also discuss topics such as geometry addition, collision-testing, and image enhancement operations.

So turn to Chapter 5, “Applet Basics,” and get ready to learn how to create an interactive visual application framework on which to build your Java games.
CHAPTER 5

Applet Basics
In this chapter we’ll look at the components that make up an applet, see how an applet differs from an application, and check out a few applet examples. Since a greater part of this book focuses on developing games within an applet, it is good to first know the basics of how applets work.

We’ll also look at how you can put some of the Java Abstract Window Toolkit (AWT) components to use in your applets. The Java AWT contains a set of classes and interfaces for drawing visual components within a window. I find it easiest to think of the AWT in terms of two separate categories: containers and components. A container represents an object capable of holding other objects. Components refer to the visible objects that the user can interact with. Buttons, radio buttons, and text labels are just a few examples of component objects.

The root of the AWT lies within the Component class, located in the java.awt package. The Component class is an abstract class intended to be extended to create specific controls. All AWT classes, even the container classes, extend the Component class. Figure 5.1 illustrates the class hierarchy under the Component class. Note this illustration does not show all of the AWT classes; it only shows those that extend the Component class.

Before we look at these classes, let’s see what exactly an applet is and then set up and run a simple example.

**What Is a Java Applet?**

A Java applet is simply a piece of software that is embedded within another application. In other words, applets are not stand-alone applications; they are dependent on other software to run. An applet context refers to the software that runs an applet. This can be anything from a Web browser to Sun’s appletviewer utility.

**Figure 5.1**

*Extensions of the Component class*
Applets are great because virtually anyone with a computer and an Internet connection already has the software required to run them. Typically, when a user loads a Web page containing an applet, the applet content is transmitted across the network and loaded into the user’s Web browser. An applet’s content can be anything from text to graphics to sound and music files.

Another great thing about applets is that they do not have to be downloaded again each time a user wishes to access them. Applets can be permanently downloaded for offline viewing, either within a Web browser or within the appletviewer utility.

Applets are a convenient way to deliver game content to your users. Applets are by nature contained within a window and can be easily programmed to respond to user events such as mouse clicks and keystrokes. In languages such as C++, you know it can take a while to get just the window up and running. Applets already contain the skeleton graphical user interface (GUI) for your games; all you need to do is fill it with exciting gaming content.

**Applets Versus Applications**

A Java applet is *not* the same as a Java application. Until now, most of our examples have been written as applications. Applications are written to run independently of any software except the Java Virtual Machine. The applications you’ve seen thus far were console programs, which were run on the command line.

However, there are other ways to deliver application content, such as through the Java Swing classes. Packages such as Swing produce a dedicated window and are not run through applet contexts. However, I find Swing programs to be too bulky to use for games. I prefer to use Swing for “productivity” software rather than gaming software. Additionally, it is not as convenient for the end user to load and run Swing programs as it is applets. That is why I usually choose to use applets for my games. If you prefer application programming to applet programming, be my guest. Most of the concepts we’ll cover can be applied to both program paradigms.

**The Structure and Life Cycle of an Applet**

Now that you know what an applet is, you should know how an applet actually works. The Java Applet class contains the methods needed to initialize and run an applet. Your job is to use inheritance to override the Applet class and fill in these
methods as needed. When an applet is loaded into its applet context, certain applet methods are called in a specific order. At the scope of the Applet class, these methods don’t actually do anything. However, they are not declared as abstract either, so you don’t have to override them. Nonetheless, they are provided as empty skeleton methods for your applets to follow.

According to Figure 5.2, after an applet has been loaded into its context its init method is called. Within the init method you should provide the code to initialize your applet. This includes initializing your game objects as well as loading any images or sound your applet may use.

The next method to be called is the start method. It simply informs the applet that it is ready to begin execution. You can use the start method to do things such as begin animation sequences and threads.

After the start method has completed, the paint method is called. This is where the visible applet content is rendered to the window. It passes to it the applet’s Graphics context, which is used to present graphic material to the window. Don’t worry too much about the Graphics class right now; we’ll look at it in greater depth in Chapters 6 and 7.

What happens next is up to you. This is where you’ll want to add interactivity between the user and the applet. We’ll look at this more in the following chapters. When the user is finished with the applet and either goes to another Web page or closes the browser altogether, the stop and destroy methods are called. The stop method stops the execution of the applet. So if you started an animation within the start method, this is the time to end it. The final method called within an applet is the destroy method. It is here that you should terminate live objects, such as threads, created during the life of the applet.
Hey, What Happened to main?

In our discussion on applets thus far, there has been no mention of the main method. Actually, Java applets do not contain a main method. Applets are started automatically by their applet context; therefore, there is really no need for a main method. At the API level, all you need to concern yourself with is the fact that the init method is called first, then start, and so on. Later on, you'll see how you can implement the main method to allow your programs to behave as either an applet or an application.

A Sample Applet

Here’s the part you’ve been waiting for—a solid code example of how an applet works. The following program, ManyShapes, extends the Applet class and defines only one method: the paint method. The paint method draws 10,000 shapes to the window. Within each iteration of the loop, a random number chooses the shape to be drawn.

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

public class ManyShapes extends Applet
{
    // this method overrides the paint method from the Applet class
    public void paint(
        Graphics g // the Graphics context to draw with
    )
    {
        // create a new number generator
        Random r = new Random();

        // draw 10000 shapes
        for(int i = 0; i < 10000; i++)
        {
            // generate random values for our shape
            int x = r.nextInt()%300;
            int y = r.nextInt()%300;
            int width = r.nextInt()%300;
            int height = r.nextInt()%300;

            // set a random color
```
g.setColor(new Color(r.nextInt()));

// generate a positive number between 0 and 4
int n = Math.abs(r.nextInt() % 5);

// draw a shape based on the value of n
switch (n) {
    case 0:
        g.draw3DRect(x, y, width, height, true);
        break;
    case 1:
        g.drawRect(x, y, width, height);
        break;
    case 2:
        g.drawOval(x, y, width, height);
        break;
    case 3:
        g.fillRect(x, y, width, height);
        break;
    case 4:
        g.fillOval(x, y, width, height);
        break;

    // this shouldn't happen; but if it does, print a message
    default:
        System.out.println("Invalid case: " + n);
        break;
} // switch
} // for
} // paint

} // ManyShapes

Again, you only had to define a single method for your program. Don’t define methods such as the init method because you have no class members to initialize. Try running the code above if you haven’t already. If you need a refresher on compiling and running applets, see the next section.
Running Java Applets

Here’s a refresher on how to compile and run your newly created applet from the command line. If you haven’t already, type the code listing above into a file named ManyShapes.java. Now you must create an .html file containing the applet tag that loads your applet. The following .html code loads the ManyShapes applet with a width and height of 300 pixels. I also added a horizontal rule (<hr>) tag before and after to applet to set it off from the rest of the file. Include it if you wish.

```html
<html>
<head>
<title>ManyShapes</title>
</head>

<body>
<hr>
<applet code=ManyShapes.java width=300 height=300></applet>
<hr>

</body>
</html>
```

You can now save the .html file to any name you want; for a lack of creativity, I named mine ManyShapes.html.

Now you can compile your source code. As always, use the javac utility as such:

```
javac ManyShapes.java
```

Fix any errors that may occur, and then recompile. Once you get a clean compile, you can simply open the .html file using your favorite (Java-enabled) Web browser, or use the appletviewer utility:

```
appletviewer ManyShapes.html
```

Congratulations! Your first Java applet. A sample run of the ManyShapes applet is shown in Figure 5.3.
Common AWT Components

A handy way to get input from your users is with some of the commonly found AWT components, such as buttons and text fields. In the next few sections, we’ll look at several of these components and how they can help your applets take shape.

Adding AWT components to your applets is fairly simple. First of all, if you anticipate using a visual AWT component in your programs, you must first implement an `EventListener` interface in the class definitions that are interested in receiving messages from your component. The `EventListener` class is located within the `java.util` package, but its subinterfaces that you’ll typically use are located in the `java.awt.event` package.

C++ programs can equate implementing the `EventListener` interface to adding a callback function to receive system messages. Typically, it will be your applet class that will receive component messages, hence it will implement an `EventListener`.

After implementing the proper `EventListener` interfaces you can define which AWT controls your program needs. Each control will usually reside as a private (or protected) class member within your applet. After allocating memory to your AWT controls you must register them with the applet so that their events will be caught. Figure 5.4 illustrates how you can attach a button to your applet and catch any action events that occur.

To solidify this in your mind, I offer the steps needed to add AWT components to your applets:

- Implement your applet class as a subinterface of `EventListener`.
- Define the methods required by implemented `EventListener` interfaces; you can leave these methods empty for now.
Add private or protected AWT class members to your class.

Initialize, register, and add your newly created AWT components to the container.

Fill in the methods defined by your EventListener interfaces.

In the following sections, we’ll look at more concrete examples of using AWT components within your applets.

**Buttons**

Buttons are a quick and easy way to allow your users to make choices inside your applets. Figure 5.4 showed us a simple model of button usage. Let’s jump right into a more complex example that will actually allow a button to generate an action. The following program, ButtonTest, contains a single Button object. When the button is pressed, the window’s background color will change. The program cycles through four different background colors.

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

public class ButtonTest extends Applet implements ActionListener
{
    // a single, lonely Button
    private Button button;

    // background colors used by the applet
    private final Color bgColors[] = new Color[] { Color.RED, Color.BLUE, Color.GREEN, Color.YELLOW
```
private int currentColor;

// this method overrides the init method from the Applet class
public void init()
{
    // create a new Button object, register it with the applet, then add
    // it to the applet
    button = new Button("Press me!");
    button.addActionListener(this);
    add(button);

    // initialize the index for background coloring
    currentColor = -1;
    changeWindowColor();
}

// this method overrides the paint method from the Applet class
public void paint(Graphics g)
{
    // set the window’s background color based on the current index
    setBackground(bgColors[currentColor]);

    // set the foreground (text) of the button to the background color
    button.setForeground(bgColors[currentColor]);
}

// increments the current background color index
private void changeWindowColor()
{
    currentColor++;
    if(currentColor == bgColors.length)
    {
        currentColor = 0;
    }
}
// implementation of the actionPerformed method from the ActionListener interface
public void actionPerformed(ActionEvent e) {
    // if button fired the event, change the window's background color
    if(button == e.getSource()) {
        changeWindowColor();
        repaint();
    }
}

// ButtonTest

As you can see, the Applet class implements the ActionListener interface in order to register itself as an object capable of receiving button events. ActionListener defines a single method, actionPerformed, which takes an ActionEvent object as its parameter. When an event occurs (such as clicking and releasing a button), an ActionEvent object that describes the event is built and sent to the actionPerformed method. This is where you can define the actual action that occurs.

You should also note the use of the getSource method from the ActionEvent class. It returns the Object that triggered the event. This is helpful when you have multiple components that register with the same ActionListener. Notice that for the above example, you don’t necessarily need to check the source because the applet contains only a single Button object.

Examine each method in the ButtonTest applet from top to bottom to understand exactly what’s going on. You might also want to type it in and try it for yourself. Your friends and family will enjoy hours of fun cycling through the four different colored backgrounds. Check out a screenshot of the ButtonTest applet in Figure 5.5.
Radio Buttons

There will be times when you’ll want the user to select among several values, or you’ll want to keep track of “on” and “off” values for a specific attribute. This is where radio buttons (you might call them check boxes) come in handy. Radio buttons differ from regular buttons in that interaction with radio buttons is not caught by the ActionListener interface. Instead, your applet must implement the ItemListener interface. Much like the ActionListener interface, the ItemListener interface defines a single method, in this case the itemStateChanged method, which takes an ItemEvent as its parameter.

Radio buttons can be added to your applets singularly or within button groups. To allow the user to toggle the sound on or off, you might use a single radio button. To create such a labeled radio button initially set to “on,” type the following:

```
Checkbox sound = new Checkbox("Enable Sound", true);
```

To access the state of a radio button, you can use the getState method, such as the following:

```
boolean soundEnabled = sound.getState();
```

Another way to use radio buttons is within button groups. Button groups preserve the idea of a single selection within the group. When a member of the group is toggled “on,” all other buttons within the group are automatically toggled “off.” Use this when your applet requires only one of a group of items to be active. The Java CheckboxGroup class is used to hold a group of radio buttons.

The following program, CheckboxTest, uses a CheckboxGroup to hold a number of selections. Each time a different item is selected, a message is printed to the console (see Figure 5.6).

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

public class CheckboxTest extends Applet implements ItemListener {
    // a CheckboxGroup to hold a number of items
    private CheckboxGroup cbg;

    // selections used by the applet
    private final String selections[] = {
```

private Checkbox createCheckbox(String label, // label for Checkbox CheckboxGroup group, // group Checkbox belongs to boolean enabled       // true to set this Checkbox "on"
)
{
    Checkbox cb = new Checkbox(label, group, enabled);
    cb.addItemListener(this);
    return cb;
}

// this method overrides the init method from the Applet class
public void init()
{
    cbg = new CheckboxGroup();
    for(int i = 0; i < selections.length; i++)
    {
        add(createCheckbox(selections[i], cbg, false));
    }
}

// implementation of the itemStateChanged method from the ItemListener
// interface
public void itemStateChanged(ItemEvent e)
{
    // print out a message about the selection
    System.out.println("Yes, I certainly agree, " +
                      cbg.getSelectedCheckbox().getLabel() +
                      " is very delicious!");
}

} // CheckboxTest

Note the use of the ItemListener interface and the addItemListener method. I also created a createCheckbox method to aid in creating Checkbox objects. Evil, indeed.
Another component you might find useful is the easy-to-use `Choice` class. Microsoft Foundation Classes (MFC) programmers will probably know these objects as *combo boxes*. Either way, a `choice` simply provides the user with a drop-down box of items to select from. Items maintained within the list are kept as `String` objects.

Much like the `Checkbox` class, `Choice` objects register themselves to `ItemListener` classes that wish to receive events about them. `ItemListener` classes must implement the `itemStateChanged` method. Again, this method takes an `ItemEvent` as its parameter.

Let’s jump right into an example. The following applet listing, `ChoiceTest`, creates a `Choice` object with the very scientific and technical names of several parts of our skeletal system. When the state of the `Choice` box is altered, an informative message about our skeletal system is printed to the console.

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

public class ChoiceTest extends Applet implements ItemListener
{
    // technical names of some skeletal bones
    public final String[] BONES = {
        "foot", "leg", "knee", "hip", "rib", "shoulder", "neck"};

    // a drop-down box containing the above String array
    private Choice choice;

    public void init()
    {
        setBackground(new Color(125, 0, 225));

        // create our Choice and register it as an item listener
        choice = new Choice();
        for(int i = 0; i < BONES.length; i++)
        {
            choice.addItem(BONES[i], BONES[i]);
        }

        // register as an item listener
        choice.addItemListener(this);
    }

    public void itemStateChanged(ItemEvent e)
    {
        // do something
    }
}
```

Figure 5.6
Testing the CheckboxTest applet
{
    // add a String to describe each choice
    choice.add(BONES[i]);
}
choice.addItemListener(this);
add(choice);
}

// called when the state of a registered listener is changed
public void itemStateChanged(ItemEvent e)
{
    // generate a different index than the one currently selected
    int index;
    do
    {
        index = (int)(Math.random()*BONES.length);
    } while(index == choice.getSelectedIndex());

    // print out an important fact about the human anatomy
    System.out.println("The " + choice.getSelectedItem() +
        " bone is connected to the " + BONES[index] +
        " bone...");
}

}    // ChoiceTest

Figure 5.7 shows us the ChoiceTest applet in action.

Not only is the ChoiceTest applet a great example of how to use Java Choice objects, but it is rather informative as well.

Now let's switch directions and try something a little more practical; let's develop an applet that will allow the user to play and loop different audio files.

Figure 5.7
The ChoiceTest applet
Playing and Looping Audio Files

I thought this would be a good time to jump topics and look at something just about every game needs: sound. I remember when I was in second grade and I had to play *The Legend of Zelda* on mute late at night on school nights. It wasn’t nearly as fun.

In the ancient days when MS-DOS games were all the rage, developers had to write separate drivers to make games work on different sound cards. As more and more different types of sound cards became available, the harder it was for game developers to keep up. Luckily for us, today there are easier ways to play audio files, such as with Microsoft’s DirectSound package. But even with DirectSound it can take a while to get a good sound engine up and running. Heck, DirectSound doesn’t even have a standard way to load audio files. What a pain!

Although playing sound within a Java applet is a rather trivial task, I’d first like to make a few points about playing sounds in Java. First of all, I suggest using the Sun .au format for all of your sound files. It’s a good format that is guaranteed to work across platforms. Although other formats might work well within Windows, they might not be valid formats on other systems such as Linux or the Mac.

In this section, we’ll focus on how to load and play audio files from within a Java applet. The packages related to java.sound contain a lot more as far as manipulating and sampling audio files; however, since some of it requires manual installations and security access that may be inconvenient for the user to adjust, using it might not be the best idea for games. But feel free to refer to these packages anyway if you’re still curious about the deeper features of sound manipulation under Java.

For now, let’s focus on the most direct way to load audio files within an applet. The Applet class provides several ways to load and play audio files. There are two methods for direct audio playback, both named play; one takes a URL (Uniform Resource Locator) object along with a String object, and the other just takes a URL. These methods create a temporary AudioClip object and immediately stream its data to audio output. No errors are generated if the audio file cannot be found or is otherwise unavailable.

The play method wastes memory and can potentially slow down applets since the audio data is read and played on the fly. A better way to handle audio for games is to cache the audio data ahead of time. This way if a sound is requested for playback over and over, it can be read from a single source that is already loaded into memory. For this, Java provides the AudioClip interface for playing, looping, and stopping audio playback. Although AudioClip is an interface, all versions of Java for varying platforms define their own internal class implementations that define this
interface. Since you're programming generically, all you need to worry about is the AudioClip interface at this scope.

To create an AudioClip object, use the Applet getAudioClip method. Like the play method, this method takes either a URL object or a URL object along with a String object. I prefer using the version that takes two parameters when my audio content is stored at the same location as my applet bytecode. For instance, the following will load a file named bang.au:

```
AudioClip ac = getAudioClip(getCodeBase(), "bang.au");
```

The getCodeBase method returns the URL from where the applet code is located. So whether you decide to run the applet remotely or locally, the above method call will work properly.

A third way to load an audio clip is with the Applet newAudioClip method. It comes in a single flavor that takes a URL object that points directly to the audio file. Since this is a static method, it is useful for when you want to load an AudioFile but you do not have a reference to a valid Applet object.

The following code listing, AudioChoiceTest, loads in several AudioClip objects from file and uses a Choice to allow the user to select a clip. There are also buttons that allow playback, looped playback, and termination of playback for the clips. As always, this applet is available on the CD-ROM for those who don’t feel like typing it in.

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

// allows the user to choose from several audio clips to play
public class AudioChoiceTest extends Applet implements ActionListener
{
    // audio names for this program
    public final String[] AUDIO =
        { "ping", "pop", "return", "salvation", "shuffle", "squish" };

    // a drop-down box containing the above String array
    private Choice choice;

    // the actual audio clip data
    private AudioClip[] clips;

    // control buttons to play or stop sounds
```
private Button playClip;
private Button loopClip;
private Button stopClip;
private Button stopAllClips;

// tracks which clips are currently being played
private boolean[] clipsPlaying;

public void init()
{
    setBackground(new Color(48, 255, 0));

    // create the drop-down box and AudioClip objects
    choice = new Choice();
    clips = new AudioClip[AUDIO.length];
    clipsPlaying = new boolean[AUDIO.length];
    for(int i = 0; i < AUDIO.length; i++)
    {
        // add a String to describe each choice
        choice.add(AUDIO[i]);
        // add pathname and extension to the audio clip name
        clips[i] = getAudioClip(getCodeBase(), "audio/"+AUDIO[i]+".au");

        // a value of false means that the clip is not playing
        clipsPlaying[i] = false;
    }
    add(choice);

    // create the buttons to play or stop audio clips
    playClip = new Button("Play clip");
    playClip.addActionListener(this);
    add(playClip);

    loopClip = new Button("Loop clip");
    loopClip.addActionListener(this);
    add(loopClip);

    stopClip = new Button("Stop clip");
    stopClip.addActionListener(this);
    add(stopClip);
add(stopClip);

stopAllClips = new Button("Stop all clips");
stopAllClips.addActionListener(this);
add(stopAllClips);

// gray-out the stop buttons if there is nothing to stop
stopClip.setEnabled(false);
stopAllClips.setEnabled(false);
}

// stops all playing audio clips
public void stop()
{
    for(int i = 0; i < AUDIO.length; i++)
    {
        if(clipsPlaying[i])
        {
            clips[i].stop();
        }
    }
}

// allows the user to play, loop, or stop the audio clips
public void actionPerformed(ActionEvent e)
{
    int clipIndex = choice.getSelectedIndex();
    AudioClip clip = clips[clipIndex];

    // play the selected clip
    if(e.getSource() == playClip)
    {
        clip.play();
        stopClip.setEnabled(true);
        stopAllClips.setEnabled(true);
        clipsPlaying[clipIndex] = true;
    }

    // loop the selected clip
else if(e.getSource() == loopClip) {
    clip.loop();
    stopClip.setEnabled(true);
    stopAllClips.setEnabled(true);
    clipsPlaying[clipIndex] = true;
}

// stop the selected clip
else if(e.getSource() == stopClip) {
    clip.stop();
    stopClip.setEnabled(false);
    stopAllClips.setEnabled(false);

    clipsPlaying[clipIndex] = false;

    // enable stop buttons if at least one clip is playing
    for(int i = 0; i < AUDIO.length; i++) {
        if(clipsPlaying[i]) {
            stopClip.setEnabled(true);
            stopAllClips.setEnabled(true);
            break;
        }
    }
}

// stop all playing clips
else if(e.getSource() == stopAllClips) {
    for(int i = 0; i < AUDIO.length; i++) {
        if(clipsPlaying[i]) {
            clips[i].stop();
            clipsPlaying[i] = false;
Figure 5.8 gives you an inside look at the AudioChoiceTest applet.

I threw in a liberal amount of inline documentation, so the above code should be pretty self-explanatory. Try looping several sounds all at once. Java allows several sounds to be played simultaneously and produces a composite of all simultaneously playing streams. This enables you to produce some nice effects with minimal effort.

One final word of caution: The AudioChoiceTest is bound to annoy both friends and foes alike. So make sure your speakers are on all the way and try this one out late at night when everyone is asleep. Try adding in your own annoying sounds and setting them for loop playback next time you leave the house.

I hope you have a good feel for playing sounds within an applet. Now we’ll move on and talk more about the Java AWT, starting with text fields.

**Text Fields**

Text fields are a very easy way to read string input from your users. When creating a text field, you will typically want to specify the maximum number of characters the...
field can contain. You can also specify optional default text within the field. The following illustrates two ways to create text fields and add them to your applet:

```java
TextField tf1 = new TextField("I am TextField tf1!", 30);
add(tf1);
TextField tf2 = new TextField(22);
add(tf2);
```

Another useful method in the TextField class is the getText method. It returns a String object containing the field’s text. The following prints out the contents of tf1 to the console:

```java
System.out.println(tf1.getText());
```

You do not need any listeners to retrieve text from a text field. The next section shows an example of how to use a TextField object in your applets.

**Labels**

A Java label is simply a single line of static text painted to the screen. Labels are great for adding titles to components or to request user input. The following creates a simple label, as well as a text field and an “OK” button to retrieve the user’s name. You can see the output for this code in Figure 5.9.

```java
Label label = new Label("Enter your name (15 chars max.): ");
add(label);
TextField field = new TextField(15);
add(field);
Button button = new Button("OK");
button.addActionListener(this);
add(button);
```

In addition to adding listeners to your objects, you can remove them as well. For instance, the Button class has a removeActionListener method that will remove the action listener from the object. Use this method if you ever want to disable components or you need to cease listener feedback. Methods that remove listeners may also be called when the applet’s stop method is invoked. However, since all child threads are killed once their parent thread is killed, removing listeners when your applet stops is not completely necessary.

![Figure 5.9](Prompting for the user's name)
Layout Management

Thus far, it might appear as though the components layout has been added haphazardly or randomly to the applet windows. Actually, there is a method to the madness. So how does an applet know how to place its components? The answer lies within the use of layout managers. LayoutManager is an interface implemented by classes that know how to place components within a container. There are several classes within the java.awt package that implement the LayoutManager interface. We’ll start by looking at the FlowLayout class.

The FlowLayout Class

The FlowLayout class is perhaps the simplest of all of the java.awt layout classes. It simply lays out components from left to right, in the order in which they were added to the applet. This is the default layout manager, so it is used automatically even if you do not specify one directly. All of the applet examples you’ve seen thus far have been using it implicitly, so you’ve seen how it works.

However, if you want to specify use of the FlowLayout class (or any other layout manager) directly, you would typically call the setLayout method within your container’s init method, such as the following:

```java
setLayout(new FlowLayout());
add(new Button("Layout"));
add(new Button("Managers"));
add(new Button("Rule!"));
```

Why would you ever want to explicitly specify a FlowLayout if it is the default? You can send parameters to the FlowLayout constructor to specify both the layout’s alignment as well as the horizontal and vertical gaps between components.

Other than the default constructor, there are two other FlowLayout constructor methods. The first takes an int value describing the alignment of the layout. There are five values that can be sent as the parameter: LEFT, CENTER, RIGHT, LEADING, and TRAILING. The following shows how to create a left-justified FlowLayout with horizontal and vertical gap size of 10:

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

If no parameters are sent to the constructor, the default alignment is FlowLayout.CENTER and the default gap size is set to five pixels. Figure 5.10 shows a right-justified FlowLayout with three options to put the hurt on your opponent.
The GridLayout Class

The GridLayout class represents a second type of layout manager. A GridLayout places components within a grid with a specified number of rows and columns. All components within the grid are of equal size.

Creating a GridLayout is similar to creating a FlowLayout, except that it has different constructor methods. The default constructor creates a layout containing a single component per column. The other two are listed below:

\[
\begin{align*}
\text{GridLayout} & \quad (\text{int rows, int cols}); \\
\text{GridLayout} & \quad (\text{int rows, int cols, int hgap, int vgap});
\end{align*}
\]

Use of these constructors should be self-explanatory. If you don’t specify the gaps their values will default to zero. If you specify either the rows or columns value as zero, or you add more than rows × columns components, the layout manager will adjust its values so that the components still fit within the grid.

The following listing, GridTest, creates a 3 x 3 arrangement of labels. There is a five-pixel horizontal and vertical gap between the labels. Each label background color is green so you can see the gaps.

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

public class GridTest extends Applet
{
    public void init()
    {
        // create a String and a StringTokenizer to parse the String
        String string = "My Head Is My Only House Unless It Rains";
        java.util.StringTokenizer st = new java.util.StringTokenizer(string);

        // create a 3 by 3 grid layout with a 5 pixel gap between components
        setLayout(new GridLayout(3, 3, 5, 5));

        // for each String token, create a label with a green background and
        // add it to the panel
        while(st.hasMoreTokens())
            // Figure 5.10
            // A right-justified
            // FlowLayout
```
{Label label = new Label(st.nextToken(), Label.CENTER);
label.setBackground(Color.green);
add(label);
}
// GridTest

Check out Figure 5.11 for output to the GridTest applet.

The BorderLayout Class

The BorderLayout class is yet another type of layout manager. This class arranges components according to the four ordinal directions (north, south, east, west), plus a fifth “center” location. Figure 5.12 shows how all five locations of a BorderLayout are established relative to one another.

The code to create a BorderLayout containing buttons similar to Figure 5.12 might look like the following:

setLayout(new BorderLayout());
add(new Button("North"), BorderLayout.NORTH);
add(new Button("South"), BorderLayout.SOUTH);
add(new Button("East"), BorderLayout.EAST);
add(new Button("West"), BorderLayout.WEST);
add(new Button("Center"), BorderLayout.CENTER);

Figure 5.11
The GridTest applet

Figure 5.12
Component arrangement according to the BorderLayout class
Other than the default constructor, there is also a constructor that takes the horizontal and the vertical gap size, respectively, to separate the components.

**The CardLayout Class**

The CardLayout class stacks components on top of one another like a deck of cards. The CardLayout class has two constructors: the default constructor as well as one that takes the horizontal and vertical gap sizes.

The following applet animates through 10 Button objects labeled “Card 1” through “Card 10.” The next button in the set is presented every second. The CardTest applet uses the CardLayout for layout management and a Thread object to do the animation.

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

public class CardTest extends Applet implements Runnable
{
    // a Thread to act as the timer
    private Thread timer;

    public void init()
    {
        // create a new CardLayout
        setLayout(new CardLayout());

        // create 10 buttons stacked within the CardLayout
        for(int i = 1; i <= 10; i++)
        {
            // the second parameter is a mandatory String representation
            // of the Button to add
            add(new Button("Card " + i), "Card " + i);
        }

        // register this applet as a Thread
        timer = new Thread(this);
    }

    public void start()
    {
    }

    public void stop()
    {
    }
}
```

timer.start();
}

public void stop()
{
    timer = null;
}

// define the run method as prescribed in the Runnable interface
public void run()
{
    CardLayout layout = (CardLayout)getLayout();

    // get a reference to this thread
    Thread t = Thread.currentThread();

    // loop while the thread is active
    while(t == timer)
    {
        layout.next(this);

        // wait one second between updates
        try
        {
            timer.sleep(1000);
        }
        catch(InterruptedException e) { return; }
    }
} // run

} // CardTest

Figure 5.13 shows a sample CardTest applet run.

There are some notable methods contained within the CardLayout class that were not used in the above code listing. The first and last methods display the first and last component in the layout, respectively. In addition to the next method, there is also a previous method, which displays the previous component in the layout. Each of these four methods takes a Container as its parameter; in this case, it’s the applet itself. We will look more at the Container class in the next section.
A fifth CardLayout method worth looking at is the `show` method. It takes a `Container` as its first parameter and a `String` as its second. The `String` should match one of the `String` objects sent as a parameter to the `add` method when creating the layout. For example, when adding buttons to the layout I associated the button labeled “Card 1” with the string “Card 1,” the button labeled “Card 2” was associated with the string “Card 2,” and so on. Therefore, the following code would display the card labeled “Card 5”:

```
layout.show(this, "Card 5");
```

Containers

Thus far, you have mostly looked at component classes and how they can be arranged within an applet. Now let’s look at container classes.

Recall from our definition that a container can contain another component—even another container. Containers are separated into two main categories: those that can be moved freely and those that remain in a fixed position. In Figure 5.14, the left-hand branch of the Container hierarchy (Window and its child classes) contains the containers that can be moved freely; they exist as dedicated windows. The right-
hand branch stemming from Container (Panel and Applet) contains those that are attached to another surface, such as a Web browser or appletviewer.

Let’s extend our discussion of applets by looking at the Panel class before looking at the left-hand side of the Container class hierarchy.

The Panel Class

If you look at the Java documentation, you’ll see that the Applet class is a subclass of the Panel class. Panel, in turn, is a subclass of Container. An important property of the Container hierarchy is the fact that it’s the Container class that actually defines methods such as setLayout. Therefore, you can make the call to setLayout at any level in the hierarchy. This becomes important when you wish to embed several layout managers within the same Applet window.

You’ve already learned how to set up a few different layout managers within your applets. When you set up a layout manager within a panel (such as an applet), it controls the entire panel. So, what if you want to set up several different layouts within the same window?

The answer lies within a very special property of panels. Unlike classes deriving from Window, which can be moved independently, Panel classes must be attached to another surface. The applets that you write are attached to a Web browser or the appletviewer utility. You can actually embed several Panel or Container objects to a parent Container object. Each object that is attached to the parent Container can have its own layout manager. Using this model, you can create some rather elaborate layouts within your applets. Figure 5.15 shows how you can combine panels with different layouts to create a more complex layout.

The following code, PanelTest, illustrates a panel layout similar to that shown in Figure 5.15. Here, two Panel objects are created, each with a different layout. Several Button objects are added to each panel, and then the panels are added to the applet. The applet will use the default FlowLayout manager to display the two panels.

Figure 5.15
Embedding one Container within another
import java.awt.*;
import java.applet.*;

public class PanelTest extends Applet
{
    public void init()
    {
        // set the applet's layout to the default FlowLayout
        setLayout(new FlowLayout());

        // create a Panel with a 2x2 GridLayout and attach 4 buttons
        // to it, then attach the panel to the parent applet
        Panel p1 = new Panel();
        p1.setLayout(new GridLayout(2, 2));
        p1.add(new Button("B1"));
        p1.add(new Button("B2"));
        p1.add(new Button("B3"));
        p1.add(new Button("B4"));
        add(p1);

        // create a second Panel with a BorderLayout and attach 5 buttons
        // to it, then attach the panel to the applet
        Panel p2 = new Panel();
        p2.setLayout(new BorderLayout());
        p2.add(new Button("North"), BorderLayout.NORTH);
        p2.add(new Button("South"), BorderLayout.SOUTH);
        p2.add(new Button("East"), BorderLayout.EAST);
        p2.add(new Button("West"), BorderLayout.WEST);
        p2.add(new Button("Center"), BorderLayout.CENTER);
        add(p2);
    } // init
} // PanelTest

Since the above code does not save references to each button, there is no way to
catch events fired by them. Of course, a real applet would implement ActionListener
and hold references to the buttons as private members; the above code merely
serves as a simple example. The result of the above code is shown within a 150 x 125
window in Figure 5.16.
Creating Custom Components

To get more power out of your applets, you can extend component or container classes to behave any way you want them to. We’ll start out with a simple example that uses a single form, then build up to a more complete example.

Suppose you are developing a role-playing game where the user can customize his or her character. An important aspect of role-playing character development is skill allocation. Given an initial number of “skill points,” your user can allocate points towards different skills. The more skill points an attribute has, the greater that skill is for the character. You’ve undoubtedly seen this before in games you’ve played.

The AttributeTest applet allows the user to allocate 10 skill points among four skill types: Strength, Wisdom, Agility, and Magic. The code for this applet introduces two classes that extend basic AWT components to add functionality to your program: AttributeButton and AttributePanel. The AttributeButton class extends the Button class and adds a way to link itself to an AttributePanel and update its contents. The AttributePanel class contains a description of an attribute along with the points allocated to that attribute. It also contains two AttributeButton objects for reallocating attribute points. The user can “tweak” allocated points among the attributes until he or she is satisfied with the final allocation.

Enough of my babble—go ahead and dig into this somewhat lengthy example, then check out the screenshot in Figure 5.17.

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

// allows an attribute value to be adjusted
class AttributeButton extends Button
{
    // the Panel that owns this button
```
private AttributePanel parent;

public AttributeButton(String label, AttributePanel ap)
{
    super(label);
    parent = ap;
}

// updates the parent attribute's value
public int updatePanel(
    int pointsRemaining // points left to allocate
)
{
    // allocate a point for 'plus' buttons
    if(getLabel().equals("+"))
    {
        // only allocate if there's points remaining
        if(pointsRemaining > 0)
        {
            parent.allocatePoints(1);
            return -1;
        }
        else return 0;
    }

    // otherwise, deallocate a point
    else
    {
        // don't allow negative allocation
        if(parent.getPointsAllocated() > 0)
        {
            parent.allocatePoints(-1);
            return 1;
        }
        else return 0;
    }
class AttributePanel extends Panel {
    // text description of the attribute
    private String attribute;

    // Label holding the points allocated to this attribute
    private Label pointsAllocated;

    public AttributePanel(String attr, ActionListener l) {
        attribute = attr;

        pointsAllocated = new Label("0", Label.CENTER);

        // set the panel layout within a 3x1 grid
        setLayout(new GridLayout(3, 1));

        setBackground(Color.GREEN);

        // add Labels to describe attribute
        add(new Label(attr, Label.CENTER));
        add(pointsAllocated);

        // attach the +/- buttons to the parent ActionListener
        Button incr = new AttributeButton("+", this);
        incr.addActionListener(l);
        Button decr = new AttributeButton("-", this);
        decr.addActionListener(l);

        // add another Panel with the plus/minus buttons
        Panel p = new Panel();
public void allocatePoints(int n) {
    int value = getPointsAllocated() + n;
    pointsAllocated.setText("" + value);
}

public int getPointsAllocated() {
    return Integer.parseInt(pointsAllocated.getText());
}

public String toString() {
    // return a verbose description of the attribute
    return attribute + ": " + getPointsAllocated();
}

private final String[] ATTRS = { "Strength", "Wisdom", "Agility", "Magic" };

public void init() {
    pointsRemaining = new Label("Points remaining: 10", Label.CENTER);
    // set the applet's layout to a FlowLayout
setLayout(new FlowLayout(FlowLayout.CENTER, 5, 10));

// add the components to the layout
for(int i = 0; i < ATTRS.length; i++)
{
    add(new AttributePanel(ATTRS[i], this));
}

add(pointsRemaining);
}

// init

// public void actionPerformed(ActionEvent e)
{
    // get the points left to allocate
    int n = Integer.parseInt(pointsRemaining.getText().substring(18));

    // update the Button's Panel and the main Label
    n += ((AttributeButton)e.getSource()).updatePanel(n);
    pointsRemaining.setText("Points remaining: "+n);
}

} // AttributeTest

Graphically, the AttributeTest applet may not be the most elegant applet ever created, but you get the point. After we talk more about graphics and imaging in Chapters 7 and 8, you'll be able to spruce it up quite nicely. For now, focus on how the AttributeButton and AttributePanel classes extend basic AWT components and how the classes can communicate with one another.

The last section of this chapter extends the AttributeTest applet into a more complete way to define a character in a simple role-playing game.

Figure 5.17

Using custom components in the AttributeTest applet
A Complete Example

It is now time to put what you know about components and containers together and create an applet that can actually do something quite useful.

Our previous example demonstrated how to use basic applet components to create a character profile for a role-playing game. Although it was pretty snazzy for a beginning applet, there are a few things that can be done to improve upon it, such as:

- **Adding more attribute panels.** It might also be nice to allow the user to enter his or her name, choose a profession and gender, as well as review a summary of selected attributes.
- **Increasing flexibility.** If you want to have multiple attribute menus, it might be nice to allow them to extend from a single base class with methods common to all attribute menus. This way you can program more flexibly, which will allow you to add and change your applet more easily.

Okay, great. So let’s improve upon the AttributeTest applet and allow the user to create a more complicated character. By using the improvements stated above, you can create a much more useful application.

The idea for the CharacterBuilder applet is to create a CardLayout that can be traversed by a set of buttons. Each card in the layout will consist of a panel containing options for a single character attribute. The user can visit each card and select his or her preferences, such as name, profession, and gender. The user will also be able to allocate skills given a finite number of allocation points, as seen in the original AttributeTest applet. Figure 5.18 shows the basic layout for this extended layout.

![Figure 5.18](image_url)

*Designing the CharacterBuilder applet*
To add flexibility to the applet, let’s define an abstract `AttributePanel` class to serve as the base class for all panels that display character attributes. To get you started, here’s the source listing for the `AttributePanel` class:

```java
// file: AttributePanel.java
import java.awt.*;

// Panel for holding character attributes
public abstract class AttributePanel extends Panel
{
    // text description of the attribute
    protected String attribute;

    public AttributePanel(String attr)
    {
        attribute = attr;
    }

    public final String getAttribute()
    {
        return attribute;
    }

    // force subclasses to override the toString method
    public abstract String toString();
}
```

The abstract `AttributePanel` class serves two main purposes. First, it allows you to associate several properties to all panels that contain character attribute options. Each defines a `String` attribute to describe the panel, as well as the ability to define a `String` representation of the attribute itself. Another advantage is that you can define `AttributePanel` objects without having to know their final run-time type ahead of time. Remember that the following are legal ways to create `AttributePanel` objects:

```java
AttributePanel[] panels = new AttributePanel[3];
panels[0] = new TextFieldPanel("Name", "Enter your name: ", 20);
panels[1] = new CheckboxPanel("Gender", new String[] {"M", "F" }, "M");
```
Since the `AttributePanel` class is declared as abstract, you can’t instantiate `AttributePanel` objects directly, but that’s okay; it was designed it to be incomplete on purpose so that it can be extended by the needs of specific character attributes.

Now on with the subclasses. As I said, you want to allow the user to define a name, profession, and gender to a character in addition to distributing points among various skills. You can also create a panel containing a summary of defined character attributes. The following class listing, `TextFieldPanel`, creates a `TextField` and a `Label` to allow the user to input his or her name:

```java
// file: TextFieldPanel.java
import java.awt.*;

// holds a String attribute within a Panel
public class TextFieldPanel extends AttributePanel
{
    // the TextField for the attribute
    private TextField textField;

    public TextFieldPanel(String attr, String prompt, int textLength)
    {
        super(attr);

        // add a Label if the prompt is a valid String
        if(prompt != null)
        {
            add(new Label(prompt, Label.LEFT));
        }

        // create and add the TextField to the Panel
        textField = new TextField(textLength);
        add(textField);
    }

    public String toString()
    {
        // return the attribute, a "not specified" message
        if(textField.getText().trim().equals(""))
        {
            // return the attribute, a "not specified" message
            if(textField.getText().trim().equals(""))
            {
```
The `CheckboxPanel` class allows the user to choose a single attribute from a number of selections, such as gender and profession. Here’s the code:

```java
// file: CheckboxPanel.java
import java.awt.*;

public class CheckboxPanel extends AttributePanel
{
    // a CheckboxGroup to hold our Checkboxes
    protected CheckboxGroup cbg;

    // this method overrides the init method from the Applet class
    public CheckboxPanel(String attr, String[] items, String selected)
    {
        super(attr);

        setLayout(new GridLayout(items.length+1, 1, 5, 5));

        add(new Label(attribute, Label.CENTER));

        // create the CheckboxGroup
        cbg = new CheckboxGroup();
        for(int i = 0; i < items.length; i++)
        { // cbg
            add(new Checkbox(items[i], cbg, items[i].equals(selected)));
        }
    }

    public String toString()
    {
        return attribute + " : " + cbg.getSelectedCheckbox().getLabel();
    }
}
```

A Complete Example
Finally, here’s the last listing for attribute selection panels. The `SkillPanel` class allows the user to allocate points toward different skills. It does much the same thing that you did in the `AttributeTest` applet shown earlier.

```java
// file: SkillPanel.java
import java.awt.*;
import java.awt.event.*;

// Represents a button capable of adjusting the value of a skill
class SkillButton extends Button
{
    // Label referencing the points allocated to this skill
    private Label pointsAllocated;

    public SkillButton(String desc, Label label)
    {
        super(desc);
        pointsAllocated = label;
    }

    // parses the value from the Label
    public int getPointsAllocated()
    {
        return Integer.parseInt(pointsAllocated.getText());
    }

    // updates the pointsAllocated label
    private void allocatePoints(int n)
    {
        int value = getPointsAllocated() + n;
        pointsAllocated.setText("" + value);
    }

    // updates the parent attribute's value
    public int update(int pointsRemaining) // overall points left to allocate
    {
        // allocate a point for 'plus' buttons
        if(getLabel().equals("+"))
```
// only allocate if there's points remaining
if(pointsRemaining > 0) {
    allocatePoints(1);
    return -1;
}
}

// otherwise, deallocate a point
else {
    // don't allow negative allocation
    if(getPointsAllocated() > 0) {
        allocatePoints(-1);
        return 1;
    }
}

// de/allocation failed
return 0;
}

// holds numerical values for various character skills
public class SkillPanel extends AttributePanel implements ActionListener {
    // points allocated to each skill
    Label[] pointsAllocated;

    // overall points remaining to allocate
    Label pointsRemaining;

    // the attributes for this applet
    private String[] skills;

    public SkillPanel(String attr, String[] sk, int alloc) {
    }
super(attr);

skills = sk;

// create the pointsRemaining Label
pointsRemaining = new Label("Points remaining: " + alloc, Label.CENTER);

// set the applet's layout to a FlowLayout
setLayout(new FlowLayout(FlowLayout.CENTER, 5, 10));

// add the components to the layout
pointsAllocated = new Label[skills.length];
for(int i = 0; i < skills.length; i++)
{
    pointsAllocated[i] = new Label("0", Label.CENTER);
    addSkill(skills[i], pointsAllocated[i]);
}

add(pointsRemaining);

private void addSkill(String skill, Label label)
{
    Panel p = new Panel();

    // set the panel layout within a 3x1 grid
    p.setLayout(new GridLayout(3, 1));
    p.setBackground(Color.GREEN.darker());

    // add Labels to describe attribute
    p.add(new Label(skill, Label.CENTER));
    p.add(label);

    // attach the +/- buttons to the parent ActionListener
    Button incr = new SkillButton("+", label);
    incr.addActionListener(this);
    Button decr = new SkillButton("-", label);
decr.addActionListener(this);

// add another Panel with the plus/minus buttons
Panel buttonPanel = new Panel();
buttonPanel.add(incr);
buttonPanel.add(decr);
p.add(buttonPanel);

add(p);
}

public String toString()
{
    // return a String containing the allocation for each skill
    String s = "";
    int points = 0;
    for(int i = 0; i < skills.length; i++)
    {
        points = Integer.parseInt(pointsAllocated[i].getText());
        s = s + skills[i] + " (" + points + ")    ";
    }
    return s;
}

public void actionPerformed(ActionEvent e)
{
    // get the points left to allocate
    int n = Integer.parseInt(pointsRemaining.getText().substring(18));

    // update the Button's Panel and the main Label
    n += ((SkillButton)e.getSource()).update(n);
    pointsRemaining.setText("Points remaining: " + n);
}

} // SkillPanel

I also promised you a panel that shows a summary of user input. It contains a reference to each of the main applet's AttributePanel objects and allows their toString method to define the summary text displayed.
public class SummaryPanel extends Panel
{
    // a Label to describe each attribute
    private Label[] summaries;

    // reference to array of AttributePanels for the attributes
    private AttributePanel[] panels;

    public SummaryPanel(AttributePanel[] ap)
    {
        super();

        panels = ap;

        setLayout(new GridLayout(panels.length+1, 1, 5, 5));

        add(new Label("Summary:", Label.CENTER));

        // add the Labels to the Panel
        summaries = new Label[panels.length];
        for(int i = 0; i < panels.length; i++)
        {
            summaries[i] = new Label("", Label.LEFT);
            add(summaries[i]);
        }
    }

    // since we don't know exactly which panel has been updated, let each
    // AttributePanel update its Label
    public void update()
    {
        for(int i = 0; i < panels.length; i++)
        {
            summaries[i].setText(panels[i].toString());
        }
    }
}
You can now put it all together and start building some characters. The following CharacterBuilder applet brings the design shown in Figure 5.18 to life. It defines four attribute selection panels and a summary panel within a CardLayout. Pressing the Back and Next buttons will traverse the layout. Here’s the code for the main CharacterBuilder class:

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

// the CharacterBuilder class consists of a number of Panels arranged within
// a CardLayout along with associated "back" and "next" buttons
public class CharacterBuilder extends Applet implements ActionListener
{
    // selects the next and previous cards in cardPanel
    private Button back;
    private Button next;

    private Panel attributePanel;
    private SummaryPanel summaryPanel;

    // final String arrays representing various attribute selections
    private final String[] GENDERS = new String[] { "Male", "Female" };
    private final String[] SKILLS = new String[]
        { "Strength", "Wisdom", "Agility", "Magic" };
    private final String[] PROFESSIONS = new String[]
        { "Knight", "Ranger", "Archer", "Wizard", "Smith", "Druid" };

    // this method overrides the init method from the Applet class
    public void init()
    {
        // create a GridLayout to hold our Cards and Buttons
        setLayout(new GridLayout(2, 1));

        // get the number of skill points to be allocated to the character
        int skillPoints;
```
try
{
    skillPoints = Integer.parseInt(getParameter("SkillPoints"));
}
catch(NumberFormatException e)
{
    skillPoints = 10;
}

// create an array of panels for our attributes; one for the name, // gender, skills, and profession of the character
AttributePanel[] panels = new AttributePanel[] {
    new TextFieldPanel("Name", "Enter your name: ", 20),
    new CheckboxPanel("Gender", GENDERS, GENDERS[0]),
    new SkillPanel("Skills", SKILLS, skillPoints),
    new CheckboxPanel("Profession", PROFESSIONS, PROFESSIONS[0])
};

// create a Panel to place our CardLayout
attributePanel = new Panel();
attributePanel.setLayout(new CardLayout());

// add the AttributePanels to the main Panel
for(int i = 0; i < panels.length; i++)
{
    attributePanel.add(panels[i], panels[i].getAttribute());
}

// create the SummaryPanel and add it to our CardLayout
summaryPanel = new SummaryPanel(panels);
attributePanel.add(summaryPanel, "Summary");

// add the attributePanel
add(attributePanel);

// create and add our "back" and "next" buttons
Panel p = new Panel();
back = new Button("back");
back.addActionListener(this);
p.add(back);

next = new Button("next");
next.addActionListener(this);
p.add(next);

p.setBackground(Color.BLACK);

add(p);
}

// called when the "back" or "next" button is clicked
public void actionPerformed(ActionEvent e)
{
    CardLayout cardLayout = (CardLayout)attributePanel.getLayout();

    if(e.getSource() == back)
    {
        cardLayout.previous(attributePanel);
    }
    else if(e.getSource() == next)
    {
        cardLayout.next(attributePanel);
    }

    // update the Summary after each change
    summaryPanel.update();
}

}    // CharacterBuilder

Figure 5.19 shows the Profession selection screen from the CharacterBuilder applet.

Whew! This lengthy example demonstrates the major features discussed in this chapter: component and container use, layout managers, listeners, and custom components. The CharacterBuilder applet is the longest we’ve seen yet. Just take it line by line and visualize what each step is trying to accomplish.

There is one more point I’d like to mention. Notice at the top of the init method under the CharacterBuilder class the use of the getParameter method. Parameters can
be sent to an applet much like they can to console programs run on the command line. This is useful when you want to send information to an applet without having to recompile. The above code listing reads one parameter representing the number of skill points to allocate. The following code shows how to include parameters from within your .html documents:

```html
<applet code=CharacterBuilder.class width=350 height=160>
    <param name="SkillPoints" value="15">
</applet>
```

To access the “SkillPoints” parameter from within your applet, simply use

```java
int skillPoints = Integer.parseInt(getParameter("SkillPoints"));
```

Since the point value is an integer, I embedded the above code within a try/catch block just in case some wise guy tries to insert invalid data into the Web document. Applet parameters are a great way to make quick changes to your code without wasting time having to recompile. They are also handy for times when you want the end user to be able to define parameters for the applet quickly and easily. One of this chapter’s exercises asks you to define an applet parameter for the default character name.

**Conclusion**

Remember, a Java applet is simply a piece of software that is embedded within another application. Applets are a very convenient and popular way to deliver content to your users.

Not all of the AWT components were presented in this chapter. I wanted to use this chapter mainly as an introduction to components and containers; we’ll pick up the slack with topics such as the Graphics class coming up in the next chapter. Using AWT components is a very quick and easy way to present interactive content to the
user. Components such as buttons, check boxes, and labels can be attached to your applets with just a few lines of code. However, I feel the AWT does come with a few restrictions. Although it is great for designing software independent of window resolution, I would like to have a bit more control over the placement of my components. I would also like a way to create more attractive components that fit within the “feel” of my games rather than using the default windowing and drawing styles. But don’t discount the native AWT components entirely; their flexibility and ease of use will surely come in handy in the future.

You have had just a glimpse of the power of the Applet class. With just a few lines of code, you can embed buttons, text fields, and labels to an applet window. But it doesn’t end there. In Chapter 7, we’ll look at ways to draw lines, shapes, and text within our applets.

Exercises

5.1 Predict the output of the following code snippet:

```java
class B extends Applet {
    public static void main(String[] args) {
        System.out.println("I am a Java guru!");
    }
} // B
```

5.2 Why is it important for Applet classes to implement EventListener interfaces when using components such as Button objects? If you’re not sure, look at the prototype for the Button addActionListener method and it should become clearer.

5.3 Describe the differences and similarities between a component and a container.

5.4 Modify the ButtonTest applet so that it contains a separate Button for each color in the Color array. The Button associated with the color red should be labeled “Red,” the blue one labeled “Blue,” and so on. A simple way to do this would be to create parallel arrays all indexed by the currentColor variable. A more robust way would be to extend the Button class so that it changes the color internally. You can associate a Color with your custom Button in its constructor and have the actionPerformed method allow the Button to update the window.

5.5 Change the AudioChoiceTest applet so that it implements a List object to hold the audio file names. Your list should be capable of having multiple selections at any time, so be sure to adjust your audio control code accordingly.
5.6 Describe how multiple `Panel` objects can be used to create more elaborate layouts.

5.7 What is the air-speed velocity of an unladen swallow?

5.8 Sketch how a group of five buttons can be arranged given the following layout managers: `FlowLayout`, `GridLayout`, `CardLayout`, `BorderLayout`.

5.9 Write an applet that displays the current date and time within a `Label` object. The applet should update itself regularly so that the date and time display looks continuous.

5.10 Modify the `CharacterBuilder` class so that it reads an applet parameter describing the default character name. You’ll have to either adjust the constructor of the `TextFieldPanel` class or add a method to set the text field’s value. You’ll also have to add a parameter tag to your `.html` file like the following:

```html
<param name="DefaultName" value="Merlin"/>
```